ECRYPT Workshop on Lightweight Cryptography.

November 28-29, 2011, Louvain-la-Neuve, Belgium.

Gregor Leander and François-Xavier Standaert (Ed.)

November 28 – 29, 2011.

Université catholique de Louvain.
Louvain-la-Neuve, Belgium.

ECRYPT II

Program Committee :

- Frederik Armknecht (Universität Mannheim, Germany).
- Jean-Philippe Aumasson (Nagravision, Switzerland).
- Gildas Avoine (Université catholique de Louvain, Belgium).
- Thomas Baignères (CryptoExperts, France).
- Lejla Batina (Radboud University Nijmegen, The Netherlands and KU Leuven, Belgium).
- Guido Bertoni (ST Microelectronics, Italy).
- Jean-Luc Beuchat (University of Tsukuba, Japan).
- Andrey Bogdanov (KU Leuven, Belgium).
- Christophe De Cannière (Google, Switzerland).
- Josep Domingo-Ferrer (Universitat Rovira i Virgili, Catalonia).
- Emmanuelle Dottax (Oberthur Technologies, France).
- Orr Dunkelman (University of Haifa, Israel).
- Benoît Gérard (Université catholique de Louvain, Belgium).
- Henri Gilbert (ANSSI, France).
- Thomas Gross (University of Newcastle, UK).
- Tim Güneysu (University of Bochum, Germany).
- Tanja Lange (Technische Universiteit Eindhoven, The Netherlands).
- Stefan Mangard (Infineon Technologies, Germany).
- María Naya-Plasencia (Université de Versailles, France).
- David Naccache (Ecole Normale Supérieure, France).
- Thomas Peyrin (Nanyang Technological University, Singapore).
- Axel Poschmann (Nanyang Technological University, Singapore).
- Francisco Rodriguez-Henriquez (Centro de inv. y de Estudios Avanzados del IPN, Mexico).
- Kazuo Sakiyama (University of Electro Communications, Japan).
- Serge Vaudenay (EPFL, Switzerland).
- Erik Zenner (Technical University of Denmark).

Program co-Chairs :

- Gregor Leander (Technical University of Denmark).
- François-Xavier Standaert (Université catholique de Louvain).

Preface.

This document contains the final versions of the papers accepted for presentation at the ECRYPT Workshop on Lightweight Cryptgraphy, that was held in Louvain-la-Neuve, Belgium, in November 2011. The focus of the workshop was on all aspects related to low-cost cryptography, mixing symmetric and asymmetric techniques, algorithms and protocols, as well as hardware and software implementation issues. As program co-chairs, we hope that these proceedings will trigger further work towards the better understanding of solutions for securing small embedded devices (such as smart cards, RFIDs, sensor nodes, ...), including their underlying mathematical foundations and practical applications. We would also like to acknowledge the people having helped making this workshop a successful event. In particular, we thank authors who submitted their works, the program committee for very professional reviews, and the attendees of the workshop. The assistance of Sylvie Baudine (for the practical organization details) and Olivier Pereira (for maintaining the website) was highly appreciated. Eventually, we are grateful to our three invited speakers who accepted to share their experience about important topics, namely Joan Daemen (ST Microelectronics, Belgium) for his talk on "*Challenges in Embedded Cryptography",* Pascal Junod (HEIG-VD, Switzerland) for his talk entitled "*Bridging Theory and Practice in Cryptography*" and Matt Robshaw (Orange, France) for his talk on "Cryptography for RFIDs".

November 2011.

Gregor Leander
François-Xavier Standaert

TABLE OF CONTENTS.

Session 1: *Cryptanalysis.*

Differential Cryptanalysis of PUFFIN and PUFFIN2 1
Céline Blondeau and Benoît Gérard

Accelerated Key Search for the KATAN Family of Block Ciphers 25
Simon Knellwolf

Some Preliminary Studies on the Differential Behavior of the Lightweight Block Cipher LBlock 35
Marine Minier and María Naya-Plasencia

Session 2: *Implementation issues.*

Compact Hardware Implementations of the Ultra-Lightweight Blockcipher Piccolo 49
Harunaga Hiwatari, Kyoji Shibutani, Takanori Isobe, Atsushi Mitsuda, Toru Akishita and Taizo Shirai

Compact Implementation and Performance Evaluation of Block Ciphers in ATtiny Devices 71
Thomas Eisenbarth, Zheng Gong, Tim Güneysu, Stefan Heyse, Sebastiaan Indesteege, Stéphanie Kerckhof, François Koeune, Tomislav Nad, Thomas Plos, Francesco Regazzoni, Francois-Xavier Standaert and Loic Van Oldeneel

High Speed Implementation of Authenticated Encryption for the MSP430X Microcontroller 87
Conrado P. L. Gouvea and Julio López

Session 3: *LPN.*

An Efficient Authentication Protocol Based on Ring-LPN 104
Stefan Heyse, Eike Kiltz, Vadim Lyubashevsky, Christof Paar and Krzysztof Pietrzak

The Cryptographic Power of Random Selection 122
Matthias Krause and Matthias Hamann

Session 4: *New designs.*

TWINE: A Lightweight, Versatile Block Cipher 146
Tomoyasu Suzaki, Kazuhiko Minematsu, Sumio Morioka and Eita Kobayashi

SPONGENT: The Design Space of Lightweight Cryptographic Hashing 170
Andrey Bogdanov, Miroslav Knezevic, Gregor Leander,
Deniz Toz, Kerem Varici and Ingrid Verbauwhede

Session 5: *Authentication.*

A new generic protocol for authentication and key agreement in lightweight systems 192
Naïm Qachri, Olivier Markowitch and Frédéric Lafitte

Relation among the Security Models for RFID Authentication Protocol 211
Daisuke Moriyama, Shin'Ichiro Matsuo and Miyako Ohkubo

CANAuth: A Simple, Backward Compatible Broadcast Authentication Protocol for CAN bus 229
Anthony Van Herrewege, Dave Singelee and Ingrid Verbauwhede

Session 6: *Implementation issues.*

The Technology Dependence of Lightweight Hash Implementation Cost 236
Xu Guo and Patrick Schaumont

Enabling Standardized Cryptography on Ultra-Constrained 4-bit Microcontrollers 255
Tino Kaufmann and Axel Poschmann

Elliptic Curve Cryptography in JavaScript 277
Laurie Haustenne, Quentin De Neyer and Olivier Pereira

Differential Cryptanalysis of PUFFIN and PUFFIN2

Céline Blondeau[1] * and Benoît Gérard[2]**

[1] Aalto University School of Science, Department of Information and Computer Science

[2] Université catholique de Louvain, UCL Crypto Group, ICTEAM Institute.

celine.blondeau@aalto.fi; ben.gerard@uclouvain.be

Abstract. A sound theoretical framework for analyzing multiple differential cryptanalysis was introduced and applied to reduced-round PRESENT at FSE 2011. We propose here to apply it to the cryptanalysis of another lightweight block cipher namely PUFFIN. This cipher security has already been undermined by Leander for a quarter of the keys. In this paper we claim that both PUFFIN and its patched version PUFFIN2 can be broken by differential cryptanalysis faster than by exhaustive search and using less than the full code-book. We also improve the complexities of these attacks using multiple differentials. Particularly, we propose an attack on PUFFIN2 that recovers the 80-bit key in $2^{74.78}$ operations using $2^{52.3}$ chosen plaintexts.

Keywords: multiple differential cryptanalysis, lightweight cryptography, PUFFIN.

1 Introduction

Security and privacy in constrained environment is a challenging topic in cryptography. In this prospect many lightweight ciphers have been designed in the last few years. The most studied one is PRESENT that was proposed at CHES 2007 [1]. The popularity of this cipher may come from its very simple structure (the Substitution Permutation Network or SPN) together with the simplicity of its permutation layer that only consists in wire crossings. This cipher has been extensively studied and its security against statistical cryptanalyses has not been threatened yet. Unfortunately, this is not the case of all proposed lightweight ciphers. Indeed, some specifications include incorrect security analysis claiming resistance against classical statistical attacks namely Matsui's linear cryptanalysis [2] and Biham and Shamir's differential cryptanalysis [3]. A typical illustration is the PUFFIN cipher family. PUFFIN [4] and PUFFIN2 [5] are two recently proposed ciphers operating on 64-bit messages with respective key lengths 128 and 80. An extension of linear cryptanalysis has been applied to PUFFIN by Leander [6] and we propose in this paper to consider the weakness of both ciphers against differential cryptanalysis.

---

* This work was produced while at INRIA project-team SECRET, France

** Postdoctoral researcher supported by Walloon region MIPSs project.

The main source of incorrectness in the security analysis of PUFFIN and PUFFIN2 comes from the fact that advances in both linear and differential cryptanalysis have not been taken into account. For instance, at EUROCRYPT 2011, Leander [6] presented a new method to attack block ciphers considering the linear hull effect. Applying this method to PUFFIN he obtained an attack that recovers the master key with a data complexity of 2^{58} for a quarter of the keys while the designers claim that obtaining a gain over exhaustive search at least requires 2^{64} plaintext/ciphertext pairs. The same holds for differential attacks since only attacks using one differential are considered and since the probabilities of such differentials are underestimated by only looking at the most probable differential trail. Moreover, it is implicitly considered that the classical last-round attacks will use differentials on $r-1$ rounds (that is 31 out of 32 for PUFFIN and 33 out of 34 for PUFFIN2) while it turns out that we are able to perform attacks with differentials for $r-4$ and $r-5$ rounds of the cipher due to the slow diffusion of both the permutation layer and the key-schedule algorithm.

From a cryptanalytic point of view, the proposed attacks are very similar to the multiple differential cryptanalyses of PRESENT [7, 8]. The main difference comes from the fact that more than the two last rounds are partially inverted. This implies that the key-schedule may be exploited when partially deciphering samples to keep the time complexity small enough. This consideration is the main contribution of this paper regarding the state of the art in differential cryptanalysis. It also raises the question of using algebraic techniques at this point of the attack. This problem is closely related to the use of an algebraic wrong-pair detection as investigated in [9].

Using the theoretical framework presented in [8], we propose different attacks on both ciphers. For instance, we propose parameters to attack PUFFIN which is parametrisez by a 128-bit key with time complexity $2^{108.84}$ and data complexity $2^{49.42}$. More importantly, we also propose an attack on the patched PUFFIN2 version that recovers the 80-bit key in time $2^{74.78}$ using $2^{52.3}$ chosen plaintexts contradicting the security claims of the designers.

The remaining sections are organized as follows. First, in Section 2 we provide a detailed description of both PUFFIN and PUFFIN2 then, in Section 3 we discuss techniques for recovering key bits in a last-round attack. We also discuss techniques for performing this key-recovery part of the attack efficiently when many active S-boxes have to be considered and propose an estimate for the resulting time complexity. In Section 4 we present tools we use to analyze multiple differential attack complexities and provide some lower bounds for differential probabilities on PUFFIN and PUFFIN2. Finally, we detail our choice of parameters for the proposed attacks and provide the corresponding complexities in Section 5 before concluding in Section 6.

2 Description of PUFFIN and PUFFIN2

The lightweight cipher PUFFIN was introduced in [4] then upgraded to PUFFIN2 in [5]. Both ciphers are 64-bit SPN ciphers with round functions composed of a key addition, an S-box layer and a permutation. The order of the different components differs from a version to another but both the S-box and the permutation are the same for PUFFIN and PUFFIN2. The particularity of this cipher is that the permutation and the substitution layer are involutions, meaning that the same primitive is used for both encryption and decription process.

The initial number of key bits in PUFFIN was 128, it has been reduced to 80 in PUFFIN2 while the number of rounds has been increased from 32 to 34. The key-schedule also has been improved since it was linear in PUFFIN and is now highly non-linear in PUFFIN2. Notice that in addition to the 32 rounds of PUFFIN, a sub-key addition and a permutation are performed at the beginning. In PUFFIN2, the S-box layer is applied at the end of the 34 rounds (see Fig. 1). The round functions and key-schedules are detailed in the following subsections.

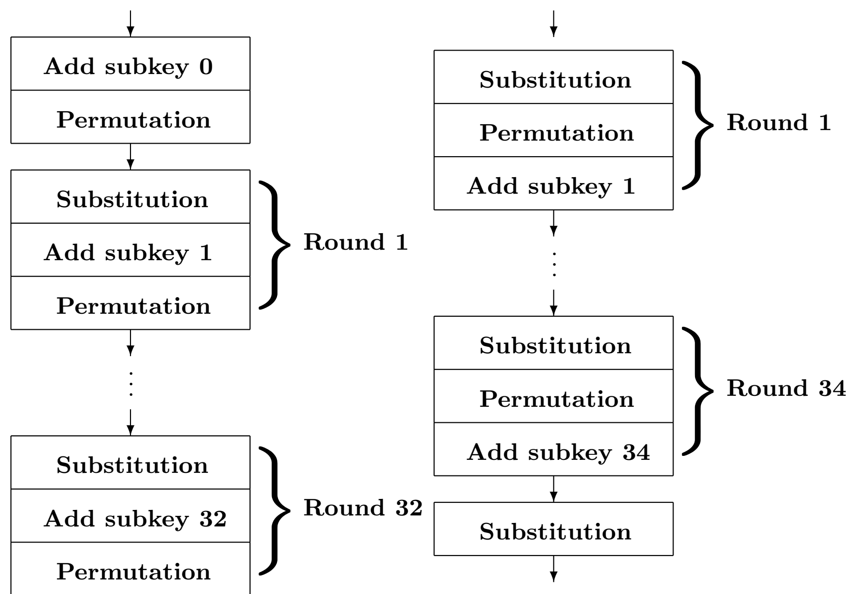


Fig. 1. PUFFIN (left) and PUFFIN2 (right) ciphers.

2.1 Round functions of PUFFIN and PUFFIN2

As mentioned earlier, the round functions are composed of a key addition, an S-box layer and a permutation layer. While in both PUFFIN and PUFFIN2 the state first passes through the S-box layer, the state is exclusively-ORed to

the round sub-key before being permuted in PUFFIN (SAP) when it is first permuted in PUFFIN2 (SPA). Both ciphers are depicted in Fig. 1.

For both versions of the cipher, the substitution layer is composed of 16 applications of a 4x4 S-box given by Table 6 in Appendix B. The permutation layer P is given by Table 7 in Appendix B. A round of PUFFIN is depicted in Fig. 2.

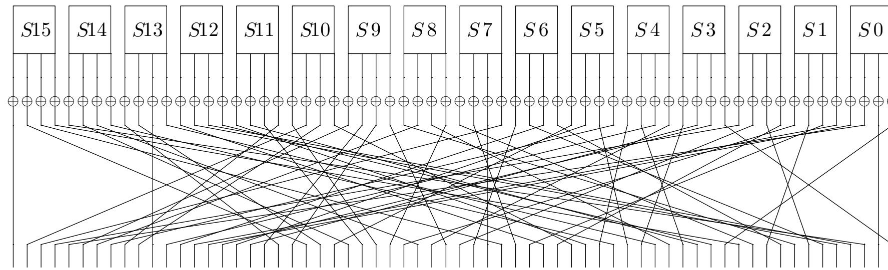


Fig. 2. One round of PUFFIN.

2.2 PUFFIN key-schedule

The key-schedule of PUFFIN is linear and operates on 128-bit master keys to generate 33 round-subkeys as follow.

1. First, a 128-bit state is initialized with the master key.
2. The 64-bit first-round subkey is extracted from the state using the selection table given in Table 8.
3. Steps 4 to 6 are iterated to obtain the remaining subkeys.
4. The state is updated using the permutation given in Table 9.
5. Bits $0, 1, 2$ and 4 are inverted excepted for rounds $2, 5, 6$ and 8.
6. A 64-bit round subkey is extracted from the state using Table 8.

2.3 PUFFIN2 key-schedule

The key-schedule in PUFFIN2 is not linear anymore but nevertheless remains simple since its structure follows the one of an SPN (without key addition of course). The key-size has been reduced here from 128 in PUFFIN to 80 bits. Then the state is processed as follows to generate 34 round sub-keys.

– First, an 80-bit state is initialized with the master key and passed through an S-box layer which consist in an application of the cipher S-box to each group of nibbles of the 80 bits.
– Then the following steps are iterated to obtain the 34 round-subkeys.

1. A 64-bit subkey is extracted by selecting the 64 leftmost bits or the 64 rightmost bits of the state depending on the round number (see Table 1).
2. The 64 bits of the state used to generate the key are passed through the cipher permutation P(64 leftmost or 64 rightmost depending on the round number).
3. The whole 80-bit state is non-linearly transformed by passing through a layer of S-boxes: adjacent bits are grouped by nibbles (4 bits) and the cipher S-box is applied to each of the 20 groups.

	Round numbers
64 leftmost bits	1,2,7,8,11,12,15,16,19,20,23,24,27,28,33,34
64 rightmost bits	3,4,5,6,9,10,13,14,17,18,21,22,25,26,29,30,31,32

Table 1. Bits of the state considered in the key-schedule.

3 Key-recovery in differential cryptanalysis

3.1 Last-round attacks: the differential case

Last-round attacks recover some key bits by peeling off one or more rounds (say r') of the cipher. The idea is to compute a statistic linked to the behavior of the cipher over r rounds thus targeting $r + r'$ rounds. This is done by partially deciphering some relevant part of the available ciphertexts over r' rounds. Hence, for each possible value of the key bits involved into this process, a statistic is obtained. These statistics translate into probabilities or likelihood values that induce at their turn an ordering of the subkey candidates from the most probable value for the correct subkey to the least one.

The main novelty of the attacks proposed in this paper lies in the fact that, by contrast with differential attacks on PRESENT, the number of peeled off rounds is larger (r' can be up to 5 rounds in the PUFFIN's case while it is equal to 2 in differentials attacks on PRESENT [7, 8]). The partial deciphering hence has a huge cost in time if no trick is used to reduce it. The context of differential cryptanalysis is particular in this prospect hence, as mentioned in the title of the section, we will focus on this case. We are first going to briefly recall the flow of differential cryptanalysis to clearly express the problem. To ease the understanding, we restrict ourselves to the particular setting where only one differential is used. The problem is quite similar when using more differentials hence we will discuss the main differences later on.

The cipher is divided into two parts: $E_{r+r'} = E_{r'} \circ E_r$ that is the first r rounds E_r that are considered when searching a good differential and the last r' rounds $E_{r'}$ that are partially inverted during the attack. The functions E have two variables: the first is the key used and the second one the message/internal

state. Pairs available to the attacker have been obtained using the correct key k_* that the attacker aims at recovering: $C = E_{r+r'}(k_*, P)$. The partial decryption using a candidate k is denoted by $E_{r'}^{-1}(k, C)$. For a given differential (δ_0, δ_r), the attack is detailed in Algorithm 1.

Algorithm 1: Last-round differential attack.

Input: a differential (δ_0, δ_r), plaintext/ciphertext pairs $(P, C = E_{r+r'}(k_*, P))$
Output: an ordered list of subkey candidates
Initialize a table D of size 2^{n_k} to 0, $D(k)$ corresponds to the counter for the subkey candidate k;
foreach *plaintext pair* (P, P') *such that* $P \oplus P' = \delta_0$ **do**
  foreach *subkey candidate* k **do**
    if $E_{r'}^{-1}(k, C) \oplus E_{r'}^{-1}(k, C') = \delta_r$ **then**
      $D(k) \longleftarrow D(k) + 1$;
return *candidates ordered according to* $D(k)$*;*

3.2 PUFFIN permutation layer diffusion

The permutation layer of PUFFIN has been designed to be hardware efficient (it uses only wire crossings) and to be an involution. This last point is the main difference with the permutation layer of PRESENT and has a major counterpart that is its slow diffusion. The full-diffusion of a cipher is reached when a bit-flip at any position of the input will influence all output bits[3]. In general, the smaller the number of rounds required to reach full-diffusion is, the more resistant the cipher should be (particularly against last-round attacks and impossible differential cryptanalysis). While using PRESENT permutation full-diffusion is reached after 3 rounds (which is optimal for a bit permutation and 16 4-bit S-boxes), 5 rounds of PUFFIN are required to obtain this property (see Fig. 3 for an example). The danger for last-round attacks is that a bad diffusion implies a small number of active S-boxes when inverting the last rounds. Hence, the attacker is able to increase r' that is decreasing r for the same number of targeted rounds, which in turns, implies the use of better statistical characteristics.

In this paper, we present attacks where only S-box $S11$ is active in the $r+1$-th round. It turns out that best differentials are the ones having only one active S-box in the last round. More particularly, output differences activating S-boxes $S3, S11$ and $S12$ have pretty good probabilities. Then, we looked at the corresponding diffusion patterns and chose $S11$ for the attack. The diffusion pattern for $S11$ is depicted in Fig. 3.

We want to point out that the larger r' is, the larger the differential probabilities for the r-round part are (since covering less rounds) inducing a smaller

[3] Notice that this does not mean that there is not any bias in the distribution of the output differences.

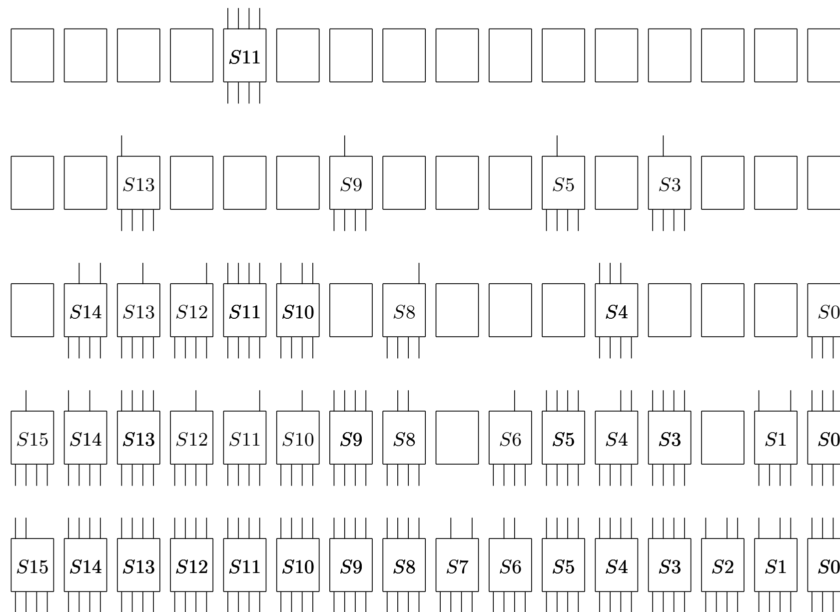


Fig. 3. Diffusion of differences activating only $S11$ over 5 rounds of PUFFIN and PUFFIN2.

data complexity. The counterpart of choosing a larger r' is that the number of key-bits involved increases. The partial decryption phase as presented in Algorithm 1 has a time complexity of $\Theta\left(N_s\, 2^{n_k}\right)$ where N_s is the number of pairs used and n_k the number of involved key bits. Hence, r' should be chosen such that $\log_2(N_s) + n_k$ is smaller than the master-key length.

It is actually possible to speed up Algorithm 1. Many techniques can be found in the literature but here we need to push them beyond their typical use since the number of rounds we invert is large (and so is the number of active S-boxes). Moreover, the analysis of the fastened Algorithm 1 becomes really tricky when using such advanced techniques. Therefore, we propose to look at key-recovery as a tree-traversal problem. Such representation will help in both the understanding of the problem and the derivation of an estimate of the complexity. This is precisely the two points we focus on in the next subsections.

3.3 Using a tree-based description for key-recovery

For a fixed pair of ciphertexts, the problem of incrementing the key-candidate counters can be seen as a tree traversal. Using this expression of the problem makes things clearer and might be helpful for the analysis of the attack complexity.

Counter incrementation as a tree traversal.
The attack uses a differential (δ_0, δ_r) over r rounds of the cipher. For each ciphertext pair (C, C') obtained from a plaintext pair having a difference δ_0, we have to increment by one each candidate counter corresponding to a value k such that $E_{r'}^{-1}(k, C) \oplus E_{r'}^{-1}(k, C') = \delta_r$. Not all the bits of $E_{r'}^{-1}$ are required to increment the counters. Only active S-boxes (as depicted in Fig. 3) need to be inverted. Let us denote by d the number of active S-boxes (maximal number of active S-boxes on the r' rounds for a pair of ciphertexts fulfilling $E_{r'}^{-1}(k, C) \oplus E_{r'}^{-1}(k, C') = \delta_r$.). The tree $\mathcal{T}_{C,C'}$ corresponding to the problem is a tree of height d and branching factor 2^4. This tree has 2^{4d} leaves corresponding to all the possible values for the key-bits XORed before inverting active S-boxes. At some depth in the tree, all the nodes are corresponding to the same active S-box. For each leave, there is a difference δ that results from the partial decryption of C and C' using the key-bits determined by the path of the leave. Note that not all the $4d$-bit values correspond to a master key. Indeed, these bits may be linked by the key-schedule algorithm. Incrementing counters boils down to detecting the leaves for which the $4d$ bits correspond to a real key (there are only 2^{n_k} such leaves) and for which the difference after deciphering is equal to δ_r.

We propose to implement the key-recovery part of the attack as the traversal of $\mathcal{T}_{C,C'}$. Then, to be efficient, we have to take care of not wasting time traversing *useless branches*. What we refer as a *useless branch* is a subtree that does contain no leave leading to a counter incrementation. Such branches may be the result of an inconsistence in the key-schedule (only 2^{n_k} out of 2^{4d} candidates correspond to a real key) or may simply come from the fact that not all the candidates are incremented by a given ciphertext pair.

Using a first filter.
An important remark here is that not all the 2^{n_k} leaves corresponding to valid candidates will produce difference δ_r after decryption. Let us introduce some formalism to express this. Let φ_{δ_r} be the function defined as follows where m is the message-length.

$$\begin{aligned} \varphi_{\delta_r} : \mathbb{F}_2^m \times \mathbb{F}_2^{n_k} &\to \mathbb{F}_2^m \times \mathbb{F}_2^m \\ (x, k) &\mapsto (y, y') = (E_{r'}(k, x), E_{r'}(k, x \oplus \delta_r)) \end{aligned}$$

The set $\mathrm{Im}(\varphi)$ is the set of ciphertext pairs that can be obtained after r' rounds of encryption when starting from a difference δ_r. The first main tool to speed up Algorithm 1 is to use a filter that is a subset $\mathcal{F} \subseteq \mathbb{F}_2^m \times \mathbb{F}_2^m$ such that $\mathrm{Im}(\varphi) \subseteq \mathcal{F}$. Indeed, no candidate counter will be incremented by a ciphertext pair not in $\mathrm{Im}(\varphi)$ thus processing such a pair is useless and time consuming. Such filter $\mathcal{F}$ is optimal when equal to $\mathrm{Im}(\varphi)$. Typically, filters consist of a set of reachable differences or potentially active key-bits obtained using techniques of truncated differential cryptanalysis or impossible differential cryptanalysis. The usefulness of algebraic techniques for this purpose is still an open question [10, 9].

Using round filters.
In the attack on PRESENT proposed in [7] and where r' is equal to 2, the author proposed to use an intermediate filtering step. For a given pair of ciphertexts, the attacker inverts the active S-boxes in the last round for all possible corresponding round-key bit values. He goes on inverting the penultimate round only if the difference obtained after the round inversion belongs to the set of possible differences. Such an additional filtering step has an important impact on the time complexity of the attack.

Using S-box filters.
The number of active S-boxes in the last round is too large in our context to use only round filters. Indeed, inverting the last round for all pairs that have not been discarded is far too time consuming. Hence, the natural idea is to apply a filter after each S-box inversion. This is precisely what we propose in this paper and we will see that it allows us to take large values for r' (up to 5). Using efficient filters drastically fastens the attack but is difficult to analyze. Indeed, ad hoc techniques used up to now becomes more tricky to apply for many reasons and particularly because dependencies between different filters may appear.

Invalid key values.
Considering S-boxes successively, we have to take care of detecting invalid values for the $4d$ key-bits early in the traversal. In Algorithm 1, each of the 2^{n_k} candidate was expanded to a $4d$-bit value then used for partial decryption. Using S-box filters, we try to avoid decrypting using all the candidates and hence we have to check for key-bit consistency during the traversal. This can be done by using r' states corresponding to round subkeys and by updating them relatively to the key-schedule each time key-bits are guessed. Then, some key-bits of previous rounds will be fixed in advance reducing the number of children of the next level of the tree.

Analyzing key-recovery complexity.
We presented techniques to speed up Algorithm 1 and we aim now at analyzing the resulting complexity of the attack. We tried to use the same kind of techniques that can be found in the literature but we faced many difficulties because of all the inherent dependencies between bits when considering more than 2 rounds of a cipher. A typical problem encountered is that the diffusion pattern provides a list of potentially active S-boxes that have to be inverted but, for a fixed ciphertext pair, some of these S-boxes may be inactive. We have to invert them to perform the r'-round decryption but we should not start with them since the corresponding filters will validate all the 4-bit values for the key. Hence, to be efficient, the order of S-boxes may depend on the ciphertext pair. It is not tractable to take this into account using aforementioned evaluation techniques and fixing the same order for all pairs would result in very high complexity. That

is the reason why we propose to analyze the complexity of this part of the attack in an ideal context using the tree-representation of the problem.

3.4 Expected complexity of the fast counter incrementation

In this subsection, we derive an asymptotic estimate for the key-recovery complexity under the strong assumption that we are able to detect all useless branches as soon as possible (that is detecting a useless branch when reaching its root) in constant time. This assumption is optimistic but we will discuss in Appendix A the fact that techniques used to speed up Algorithm 1 take constant time at the cost of table pre-computations and storage and that they allow a quasi-optimal tree traversal.

Now assuming that this hypothesis holds, then the complexity of one tree traversal is $\Theta(d\alpha)$ where α is the number of counters incremented by the processed pair. Let N be the number of available plaintext/ciphertext pairs, then we can form $N_s = N/2$ plaintext pairs and obtain the corresponding ciphertext pairs. We denote by $A_1, \ldots, A_{N_s}$ the number of incremented counters corresponding to the N_s ciphertext pairs. Then, we are interested in computing the complexity of the attack: $\sum_{i=1}^{N_s} d\, A_i$. Since we know very few about the distribution of A_i, we propose to estimate the expected value of the time complexity of the attack. Using the linearity of the expected value, we easily derive

$$\mathbb{E}\left(\sum_{i=1}^{N_s} d\, A_i\right) = N_s \cdot d \cdot \mathbb{E}(A).$$

Supposing that ciphertext pairs are uniformly distributed[4] over $\mathbb{F}_2^m \times \mathbb{F}_2^m$, then, the expected value of the cardinal of $\varphi_{\delta_r}^{-1}(y, y')$ is the ratio of the input by the output space cardinalities:

$$\mathbb{E}\left(\#\varphi_{\delta_r}^{-1}(y, y')\right) = \frac{2^{m+n_k}}{2^{2m}} = 2^{n_k - m}.$$

Hence, the expected value of the time complexity of the fastened version of Algorithm 1 using the tree-based approach is

$$\Theta(N_s\, d\, 2^{n_k - m}). \tag{1}$$

Notice that the N available plaintexts have to be read to form the pairs. Hence, it has to be taken into account when estimating the complexity of the attack. Moreover, the first filter can be applied at the same time as forming pairs using a hash table. The value provided by (1) may be smaller than N but this is due to the fact that after the first filter only very few pairs remain.

We presented a new technique to estimate the complexity of a simple differential cryptanalysis. In the next section, we present tools for analyzing the use of several differentials.

[4] This assumption is realistic since they are obtained by using $r + r'$ rounds of the cipher: if not true, it will induce an attack on $r + r' + 1$ rounds of the cipher.

4 Multiple differential cryptanalysis

In the previous section, we focused on techniques for incrementing candidate counters for a given ciphertext pair. All the statements were instantiated in the particular case of single differential cryptanalysis. Attacks presented here are multiple differential cryptanalyses that is attacks using more than one differential. The optimal way for combining information from many differentials is a work in progress and may hardly depend on the accuracy of differential probability estimates as it is the case for linear cryptanalysis [11–13]. Hence, we chose here a classical approach that consists in combining counters obtained from different differentials using addition. Such attacks have been formalized and studied in [8]. Before providing results on the differential probabilities of the best characteristics on PUFFIN and PUFFIN2, we briefly discuss the influence of the use of many differentials on the complexity given in (1) and recall the results given in the aforementioned paper.

4.1 Time cost of the use of many differentials

The asymptotic time complexity of the key recovery part given by formula (1) have been obtained for a single differential. In differential cryptanalysis, the attacker uses a set Δ of differentials. The set of input differences contained in Δ is denoted by Δ_0. To a given input difference $\delta_0^{(i)}$ in Δ_0 corresponds a set of differentials of the form $(\delta_0^{(i)}, \delta_r^{(i,j)})$. We denote by $\Delta_r^{(i)}$ the set of output differences corresponding to a given input difference $\delta_0^{(i)}$. Then, from well-chosen[5] N plaintext/ciphertext pairs, the attacker can form $N/2$ couples of plaintext for each input difference hence obtaining a total of $N_s = \frac{|\Delta_0| \cdot N}{2}$ samples.

Forming all the samples should have time complexity $\Theta(N_s)$ but it is possible to combine the formation of plaintext pairs to the first filter using a hash table (as mentioned in Section 3.4). It turns out that if the filter is efficient enough, the number of remaining pairs will smaller than N. In the other case, forming pairs will be more time consuming than $\Theta(N)$ but this complexity will be negligible compared to the one of the corresponding key-recovery. Hence, it is not abusive to consider than the time complexity of forming samples is the one of reading available plaintext/ciphertext couples that is $\Theta(N)$.

Then, for a given ciphertext pair, the input difference $\delta_0^{(i)}$ is fixed. This difference has a corresponding set of output differences $\Delta_r^{(i)}$. Since we combine differentials by summing counters, the only difference with Algorithm 1 is that a candidate counter will be incremented if the obtained difference belongs to the set $\Delta_r^{(i)}$. Hence, for a fixed input difference, the number of incremented counters will be multiplied by $|\Delta_r^{(i)}|$ influencing the complexity of a tree traversal. Summing over all possible samples, we obtain a global complexity of

[5] That is choosing plaintexts using the so-called *structures*.

$$\Theta\left(N_s\, d\, 2^{n_k - m}\, \frac{\sum_{i=1}^{|\Delta_0|} |\Delta_r^{(i)}|}{|\Delta_0|}\right). \tag{2}$$

Note that since $N_s = \frac{|\Delta_0| \cdot N}{2}$, the term $N_s \frac{\sum_{i=1}^{|\Delta_0|} |\Delta_r^{(i)}|}{|\Delta_0|}$ actually corresponds to $2N\,|\Delta|$.

4.2 Theoretical framework used

Let us first begin with the definition of some notation from the framework developed in [8].

Notation.
The attacker chooses a set Δ of differentials with probabilities $p^{(1)}, \ldots, p^{(|\Delta|)}$. The output differences of the differentials determine the set of active S-boxes involved in the partial deciphering process. The corresponding number of required key bits is denoted by n_k hence the attacker will have to distinguish the correct subkey among 2^{n_k}.

The cornerstone of the theoretical analysis in [8] is an estimate for the subkey-counter cumulative functions that is given in [8, Proposition 1]. This estimate is denoted by G and is parameterized by the number of samples. We will hence denote it by G_{N_s}. The function $G_{N_s}(\tau, p_*)$ is the estimate for the correct-key counter distribution and $G_{N_s}(\tau, p)$ is the one for the wrong keys. Values for p_* and p are equal to

$$p_* = \frac{\sum_i p_*^{(i)}}{|\Delta_0|} \quad \text{and} \quad p = \frac{|\Delta|}{2^{64}|\Delta_0|}.$$

We now recall the main two results that can be found in [8]. The first one is an estimate of the data complexity required for the correct key to be ranked among the ℓ most likely candidates with probability close to one half.

Corollary 1. *[8, Corollary1] Let ℓ be the size of the list of the remaining candidates and let n_k be the number of bits of the key we want to recover. Using the previous notations, the data complexity of a multiple differential cryptanalysis with success probability close to 0.5 can be estimated by*

$$N' = -2 \cdot \frac{\ln(2\sqrt{\pi}\ell\, 2^{-n_k})}{|\Delta_0| D(p_*||p)}, \tag{3}$$

where $D(p_*||p)$ denote the Kullback-Leibler divergence:

$$D(p_*||p) = p_* \log\left(\frac{p_*}{p}\right) + (1 - p_*) \log\left(\frac{1 - p_*}{1 - p}\right)$$

In this result the statement "success probability close to 0.5" may seem unclear and imprecise[6]. The point is that the success probability corresponding to

[6] For a complete understanding, to please refer to [14]

the data complexity N' given in this Corollary may vary a bit around this value. In the case of differential or multiple differential cryptanalysis, our experiments show that using the value of N' given in Corollary 1 leads to a success probability between 0.35 and 0.50[7]. This formula provides an intuition on the impact of the use of many differentials on the data complexity through the denominator $|\Delta_0|D(p_*||p)$.

Then, to precisely adjust the number of samples to reach a given success probability, another result of [8] should be use. This one, presented in Corollary 2, is a tight formula for computing the success probability of an attack when only the ℓ most likely keys are tested and when N_s samples are available.

Corollary 2. *[8, Corollary2] Under the previous notations, the success probability, P_S, of a multiple differential cryptanalysis is given by*

$$P_S \approx 1 - G_{N_s}\left[G_{N_s}^{-1}\left(1 - \frac{\ell - 1}{2^{n_k} - 2}, p\right) - 1, p_*\right] \tag{4}$$

where the pseudo-inverse of G_{N_s} is defined by $G_{N_s}^{-1}(y) = \min\{x | G_{N_s}(x) \geq y\}$.

The tightness of formula (4) have been empirically tested on a reduced version of PRESENT in [8] with very convincing results.

4.3 The differential probabilities

In the first analysis of the security of PUFFIN [4] authors claim that since the best trail on 31 rounds (over the 32 rounds) has a probability equal to 2^{-62}, the cipher is secure against differential cryptanalysis. On the one hand and according to PUFFIN design -i.e. similar to the one of PRESENT- a differential will be composed of so many trails that the probability of a differential is dramatically underestimated when only considering the best differential trail. On the other hand, authors' analysis of the cipher security is based on the assumption that the attacker will only recover key bits from the last round (that is $r' = 1$) while attacks proposed in this paper actually use values for r' up to 5.

Using a Branch and Bound algorithm (see [15–17] for instance), we are able to compute the probabilities of the best differential trails. Combining these trails, we obtain lower bounds on the probabilities of the best differentials. As mentioned in Section 3, we have observed that the best differential trails have a single active S-box in the input and the output difference. Therefore we chose to select differentials such than only one S-box is active in the output difference. Diffusion properties of PUFFIN and PUFFIN2 suggest the use of $S11$ as active S-box among those having good differential properties.

We now provide some results about the differential probabilities for PUFFIN and PUFFIN2. The data complexity of the attack will depend on the number r

[7] Experiments from [14] shown a range from 0.52 to 0.65 in differential settings.

of rounds targeted by differentials, it will also depend on the differentials we will use. Moreover, the estimation of these probabilities is a critical point since a too pessimistic analysis will lead to the underestimation of the attack performances. In order to compare the different values for r' we looked for the best differentials over $r = 27, 28, 29, 30$ and 31 rounds. According to Section 3.2, we focused on r-round differentials (δ_0, δ_r) such that

$$\delta_r \in \Delta_{out} = \{\texttt{0x0000Y00000000000} | Y \in \{\texttt{0x1}, \cdots, \texttt{0xF}\}\}.$$

The best differentials - corresponding to our criteria - we found are given in Table 2. Notice that we did not consider the additional operations performed before the first round in PUFFIN and after the last round in PUFFIN2 since they do not alter the probabilities of differentials.

Table 2. Best differentials on r rounds with output difference activating only $S11$ and their probabilities.

r	δ_0	δ_r	p_*
27	0x000000000000a000	0x0000400000000000	$2^{-49.71}$
28	0x000b400000000000	0x0000400000000000	$2^{-52.07}$
29	0x0000000000400000	0x0000400000000000	$2^{-53.59}$
30	0x0000400000000000	0x0000400000000000	$2^{-56.35}$
31	0x0000000000007000	0x0000400000000000	$2^{-57.9}$

5 Parameters and performances of proposed attacks

Let us now move to the choice of parameters for the attacks we propose. We want to precise that the attacks mentioned here are not claimed to be optimal since some of our choices relied on heuristics (for instance the choice of $S11$ may not lead to the best possible attack).

5.1 Formulas of the attack complexities

Simple formulas for complexities aim at easing the choice of parameters to balance them. More precisely, in the case of multiple differential cryptanalysis, there are three different steps to consider for the time complexity.

1. Obtaining plaintext/ciphertext pairs: $\Theta(N)$.
2. The key-recovery part with complexity given by formula (2).
3. The final exhaustive search that takes $\Theta(\ell\, 2^{n-n_k})$ where n is the master-key length and ℓ the maximum number of candidates to test.

Then, the time complexity of a multiple differential attack is

$$\Theta\left(N + \ell\, 2^{n-n_k} + N_s\, d\, 2^{n_k-m}\, \frac{\sum_{i=1}^{|\Delta_0|} |\Delta_r^{(i)}|}{|\Delta_0|}\right) \tag{5}$$

For a given set Δ of differentials, fixing r' will determine the values for d the number of active S-boxes and n_k the number of involved key-bits. Then, there are tight links between the data complexity N, the success probability P_S and the number of candidates to test ℓ. This relationship is represented by the formula (3). Since we aim at proposing attacks with success probabilities greater than 0.5, we propose to multiply this formula by a factor 1.5. Experiments in [14] show that for such a value, this probability gets around 0.8.

$$1.5 \cdot N' = -3 \cdot \frac{\ln(2\sqrt{\pi}\ell\, 2^{-n_k})}{|\Delta_0| D(p_*||p)}. \tag{6}$$

This is asymptotic in nature: the estimate is negative for values of ℓ close to 2^{n_k} but it tends toward the correct value of N as $\frac{\ell}{2^{n_k}}$ decreases.

This formula will be of great use since we can substitute it into the aforementioned complexities removing one parameter. Then, for a given set Δ and a fixed r', the problem boils down to balance these complexities using the parameter ℓ.

5.2 First approach: using a single differential

As both time and data complexity depend on the set of differentials, we chose, as a first approach, to study the complexities of differential cryptanalyses using a single differential. For the best differentials (δ_0, δ_r) described in Table 2, we propose to compute the complexities obtained for different values of r' ($r + r' = 32$ for PUFFIN and 34 for PUFFIN2). Once ℓ is fixed, the success probability will accurately be estimated using (4). As it is supposed to, it will vary in a reasonable range around 0.5.

Table 3. PUFFIN: Parameters of simple differential cryptanalyses using the formula for N given by (6).

r'	d	n_k	p_*	ℓ	N	Time C.	P_S
3	13	43	$2^{-53.59}$	$2^{00.1}$	$2^{57.49}$	$2^{85.10}$	0.75
4	27	81	$2^{-52.07}$	$2^{24.6}$	$2^{56.04}$	$2^{76.84}$	0.79
5	43	109	$2^{-49.71}$	$2^{78.6}$	$2^{52.45}$	$2^{101.95}$	0.85

Results in Table 3 and Table 4 confirm the fact that the formula (6) leads to a success probability close to 0.8. The parameters presented in Table 3 and Table 4 show that a differential attack using a single differential is enough to break both PUFFIN and PUFFIN2.

Table 4. PUFFIN2: Parameters of simple differential cryptanalyses using the formula for N given by (6).

r'	d	n_k	p_*	ℓ	N	Time C.	P_S
3	13	35	$2^{-57.90}$	$2^{12.4}$	$2^{61.25}$	$2^{61.35}$	0.75
4	27	59	$2^{-56.35}$	$2^{35.0}$	$2^{59.47}$	$2^{60.07}$	0.63
5	43	74	$2^{-53.59}$	$2^{61.1}$	$2^{55.60}$	$2^{70.21}$	0.87

One of the problems left as an open question for multiple differential cryptanalysis is the optimal choice of the parameters of the attack (typically the set of differences used). Results given here when using a single differential emphasize the fact that this notion of *optimal choice* may hardly depend on the context (Is the full code-book available?, Is the computational power the limiting factor?). Indeed, when moving from $r' = 4$ to $r' = 5$, samples are exchanged for computational effort. Nevertheless, there are parameters that are clearly sub-optimal as $r' = 3$ that, here, leads to an attack outperformed in both time and data complexities by other parameters.

5.3 Proposed attacks

Table 3 and Table 4 show that we can break both PUFFIN and PUFFIN2 using simple differential cryptanalysis. Nevertheless, complexities of these attacks can be improved using several differentials. The set of differential we propose for the attacks is big, that we present in Appendix in Table 10 only the parameters for the attacks with the smallest used differentials. We propose a multiple differential attack on PUFFIN where $r' = 4$ that outperforms the simple attack for $r' = 5$ i.e. that have smaller complexities (for both time and data). We also propose a multiple attack on PUFFIN2 with $r' = 4$ and attacks on both versions for $r' = 5$ to illustrate that the choice of differential sets and values for r' allow trading data complexity for time complexity.

The attacks proposed are summarized in Table 5 and detailed below.

Table 5. Attacks on the PUFFIN ciphers.

version	key bits	rounds	Data C.	Time C.	Success P.	
PUFFIN	128	32	2^{58}	2^{124}	> 0.25	[6]
PUFFIN	128	32	$2^{52.16}$	$2^{95.40}$	0.77	this paper
PUFFIN	128	32	$2^{49.42}$	$2^{108.84}$	0.59	this paper
PUFFIN2	80	34	$2^{55.58}$	$2^{64.66}$	0.58	this paper
PUFFIN2	80	34	$2^{52.30}$	$2^{74.78}$	0.78	this paper

Attacks on PUFFIN.
In the previous section, we presented a simple differential attack on PUFFIN with

data complexity $2^{52.45}$ and time complexity $2^{101.95}$ (case where $r' = 5$). Using a set of 830 differentials having $|\Delta_0| = 359$ input differences on $r = 28$ rounds (that is $r' = 4$), we obtain a multiple differential cryptanalysis ($p_* = 2^{-56.7178}$) which requires $2^{52.16}$ chosen plaintexts that has a time complexity of $2^{95.40}$ and success probability of 0.77 ($\ell = 2^{48.40}$). Complexities of this attack outperform the complexities of the simple differential cryptanalysis of PUFFIN we propose.

Nevertheless, if the bottleneck of the attacker resources is the data complexity and not the computational power, then we may use another set of parameters. For instance, there is a set of 954 differentials over 27 rounds ($r' = 5$) such that $|\Delta_0| = 318$ and that provides a probability $p_* = 2^{-54.6862}$. Taking ℓ equal to $2^{85.9}$, we obtain an attack that requires $2^{49.42}$ chosen plaintexts and that can be performed in time $2^{108.84}$ with a success probability of 0.59. If we compare this attack to the simple differential attack described in Table 3, the data complexity is divided by a factor $2^{3.03}$ while the time complexity is multiplied by $2^{6.89}$.

Examples of parameters we present show that depending on the attack requirements, different trade-offs for the complexities are possible.

Attacks on PUFFIN2.
For PUFFIN2, we also propose two kinds of multiple differential attacks, one that improves the data complexity with a small cost in time complexity (compared to the simple differential attack proposed in Table 4) and another that tries to minimize the data complexity of the attack.

We recall that in the case of $r' = 5$, the simple differential attack proposed in Table 4, can be performed using $2^{55.60}$ plaintexts in $2^{70.21}$ operations. Now, using a set of 115 differentials over 30 rounds (that is $r' = 4$) with $|\Delta_0| = 95$ input differences, we propose a multiple differential cryptanalysis on PUFFIN2 with time complexity $2^{64.66}$, data complexity $2^{55.58}$ and success probability 0.58 ($\ell = 2^{44.60}$ and $p_* = 2^{-59.1178}$). Differentials used for this attack are given in Table 10 for the interested reader to be able to check our results.

We can also perform a multiple differential attack on PUFFIN2 using differentials on 29 rounds ($r' = 5$). Using 210 differentials having $|\Delta_0| = 137$ different input differences, we obtain a probability $p_* = 2^{-57.7949}$ which lead to an attack with data complexity $2^{52.30}$, time complexity $2^{74.78}$ and success probability 0.78 ($\ell = 2^{66.5}$).

6 Conclusion

This work aims at illustrating the impact of the flaws that can be found in the security analysis of recent ciphers. We propose attacks on PUFFIN and PUFFIN2 that are low-cost ciphers designed to be more efficient than PRESENT (and particularly to be involutions). It turns out that they are also less secure since a simple differential cryptanalysis allows to break them. Using a new approach to estimate the cost of key-recovery in differential cryptanalyses, we proposed two attacks on these ciphers. The attack on PUFFIN recovers the full 128-bit key in $2^{95.40}$ operations with probability 0.77 when using $2^{52.16}$ chosen plaintexts and

the attack on PUFFIN2 recovers the full 80-bit key in $2^{65.66}$ operations with probability 0.58 when using $2^{55.58}$ chosen plaintexts.

While PUFFIN and PRESENT are similar, these differential attacks and the linear attack proposed by Leander[6] on PUFFIN and PUFFIN2 do not threaten the security of PRESENT. Differences can be explained by both the substitution and the permutation layers. Indeed, the fact that in PUFFIN, the S-box have properties that allow one-bit input and output differences implies there exists differential trails with one active S-box for each round what is not possible for PRESENT (at least two active S-boxes for one out of two rounds). This property can be removed using a linear transformation of the S-box.

Concerning the permutation layer, it turns out that the optimality of the PRESENT bit-permutation cannot be obtained with an involution. The permutation proposed in PUFFIN only reaches full diffusion after 5 rounds. This has implications in both the trail probabilities and the number of rounds that can be inverted by the attacker. Using involution permuation layer with diffusion on 4 rounds may improve the security of the cipher.

Impact of the use of "better" involutions as basic components for an SPN is an interesting scope for further research.

References

1. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.H.: PRESENT: An ultra-lightweight block cipher. In Paillier, P., Verbauwhede, I., eds.: Cryptographic Hardware and Embedded Systems - CHES 2007. Volume 4727 of LNCS., Springer (2007) 450–466
2. Matsui, M.: Linear cryptanalysis method for DES cipher. In Helleseth, T., ed.: Advances in Cryptology - EUROCRYPT 1993. Volume 765 of LNCS., Springer (1994) 386–397
3. Biham, E., Shamir, A.: Differential cryptanalysis of DES-like cryptosystems. Journal of Cryptology **4** (1991) 3–72
4. Cheng, H., Heys, H.M., Wang, C.: PUFFIN: A novel compact block cipher targeted to embedded. In Fanucci, L., ed.: Conference on Digital System Design: ARchitectures, Methods and Tools - DSD 2008, IEEE (2008) 383–390
5. Wang, C., Heys, H.M.: An ultra compact block cipher for serialized architecture implementations. In: Canadian Conference on Electrical and Computer Engineering - CCECE 2009, IEEE (2009) 1085–1090
6. Leander, G.: On linear hulls, statistical saturation attacks, PRESENT and a cryptanalysis of PUFFIN. In Paterson, K., ed.: Advances in Cryptology - EUROCRYPT 2011. Volume 6632 of LNCS., Springer (2011) 303–322
7. Wang, M.: Differential cryptanalysis of reduced-round PRESENT. In Vaudenay, S., ed.: Progress in Cryptology - AFRICACRYPT 2008. Volume 5023 of LNCS., Springer (2008) 40–49
8. Blondeau, C., Gérard, B.: Multiple differential cryptanalysis: Theory and practice. In Joux, A., ed.: Fast Software Encryption - FSE 2011. Volume 6733 of LNCS., Springer (2011) 35–54
9. Wang, M., Sun, Y., Mouha, N., Preneel, B.: Algebraic techniques in differential cryptanalysis revisited. In Parampalli, U., Hawkes, P., eds.: Information Security and Privacy - ACISP 2011. Volume 6812 of LNCS., Springer (2011) 120–141

10. Albrecht, M., Cid, C.: Algebraic techniques in differential cryptanalysis. In Dunkelman, O., ed.: Fast Software Encryption - FSE 2009. Volume 5665 of LNCS., Springer (2009) 193–208
11. Hermelin, M., Cho, J.Y., Nyberg, K.: Multidimensional extension of Matsui's algorithm 2. In Dunkelman, O., ed.: Fast Software Encryption - FSE 2009. Volume 5665 of LNCS., Springer (2009) 209–227
12. Hermelin, M., Cho, J.Y., Nyberg, K.: Statistical tests for key recovery using multidimensional extension of Matsui's algorithm 1. Advances in Cryptology - EUROCRYPT 2009 POSTERSESSION (2009)
13. Cho, J.Y.: Linear cryptanalysis of reduced-round PRESENT. In Pieprzyk, J., ed.: Topics in Cryptology - CT-RSA 2010. Volume 5985 of LNCS., Springer (2010) 302–317
14. Blondeau, C., Gérard, B., Tillich, J.P.: Accurate estimates of the data complexity and success probability for various cryptanalyses. Design, Codes and Cryptography **59** (2011) 3–34
15. Biryukov, A., De Cannière, C., Quisquater, M.: On multiple linear approximations. In Desmedt, Y., ed.: Advances in Cryptology - CRYPTO 2004. Volume 3152 of LNCS., Springer (2004) 1–22
16. Collard, B., Standaert, F.X., Quisquater, J.J.: Improved and multiple linear cryptanalysis of reduced round Serpent. In: Inscrypt'07. Volume 4990 of LNCS., Springer–Verlag (2007) 51–65
17. Blondeau, C., Gérard, B.: Links between theoretical and effective differential probabilities: Experiments on PRESENT. In: TOOLS'10. (2010) http://eprint.iacr.org/2010/261.

A More details on the tree-traversal

In Section 3.3, we modeled the key-recovery part of a differential attack as a tree traversal. Nodes at the same depth in the tree correspond to one of the active S-boxes in the diffusion path. There are 2^4 children corresponding to the possible values for the 4 key bits XORed to the output value of the S-box corresponding to the node. When traversing the tree, passing along an edge refers to XORing the corresponding 4-bit key value and inverting the S-box. When reaching a node, it is essential to detect the current branch (this node and its descendants) as useless if so. A useless branch is a branch where all leaves at depth d do not correspond to counter incrementation. Such branches may appear for different reasons. If the current node is inconsistent with the previous guessed key bits for instance (this case is really easy to detect). The other source of useless branches comes from the fact that the number of incremented counter is far smaller than the 2^{n_k} candidates. Such branches are more difficult to detect even using many filters.

When analyzing this tree traversal in Section 3.4 we made the strong hypothesis that we were able to detect useless branches at the top level (that is as soon as possible). This is a bit optimistic according to what has just been said but using the tricks we are to detail, the proportion of non-discarded useless branches can be kept small. A more precise study of the complexity taking this proportion into account may be of interest here and is left as an open question for further work.

We detail here the algorithm we have in mind for constructing and traversing the tree efficiently (that is detecting most of the useless branches in constant time).

Constructing the tree.
Obviously we are not going to construct the tree since we aim at traverse the tree avoiding useless branches. Nevertheless, there is one degree of freedom in the definition we gave for the tree $\mathcal{T}_{C,C'}$: the order of S-box inversions. Modifying the order of S-boxes will not necessarily help in improving useless branch detection technique but using an *efficient* S-box order, useless branches may be met earlier in the tree traversal. This would reduce the constant hidden in the Θ notation and hence may be carefully looked at when implementing a practical attack but can be omitted for analyzing an attack.

Detecting key-bit inconsistency.
As already mentioned, this can be done by updating a $4d$ state following the key-schedule algorithm. Then, when reaching a new node, the set of child nodes considered is restricted if some key bits have already been guessed. In this context changing S-box order may also be of interest. Indeed, for PUFFIN2, the only non-linear part of the key-schedule is the application of the substitution layer to 64 bits of the current key state. Hence, some key bits directly correspond from one subkey to the other. Starting by guessing those bits may induce more efficient filters at the top of the tree.

Round filtering.
The round filtering process as detail in [7] consist in detected some useless branch after deciphering all active S-boxes in one round. This one may not be optimal due to memory limitation. Hence, they may be reduced to a set of reachable differentials. Applying such a filter can be done efficiently since it will consist in at most 2^{63} differences which can be stored in a relevant structure with search cost logarithmic in its size. Such sieves will be applied only at some depth of the tree hence it is not abusive to consider that this cost is constant in nature. Notice that the first sieve as to be applied to all the N_s samples hence the total cost of the key-recovery step may not be smaller than N_s.

S-box filtering.
An other kind of filters are S-box filters. Instead of applying such filters after having inverted the S-box using all the possible values of key bits, it is more efficient to pre-compute a $2^4 \times (2^4 - 1)$ table containing, for each pair of outputs, the list of keys that lead to a correct input difference. The memory cost of this technique for b-bit S-boxes is $\Theta(d\, 2^{3b})$ hence is negligible compared to the memory used for counters (2^{n_k}) when inverting a large number of S-boxes. Again, the cost of such filtering is constant regarding parameters of the attack.

B Components of PUFFIN

This appendix section contains the details for PUFFIN and PUFFIN2 specification.

B.1 Common components

In Table 6 the S-box used in both PUFFIN and PUFFIN2 is given using hexadecimal values.

Table 6. PUFFIN/PUFFIN2 S-box in hexadecimal, `S1(0x0)` = `0xD`.

input	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
output	D	7	3	2	9	A	C	1	F	4	5	E	6	0	B	8

Then, Table 7 specifies the bit-permutation P. The table has to be read as: the input bit at position $a \cdot 8 + b$ will be sent to the position given in the cell at line a and column b in the output. For instance, the 19-th input bit will correspond to the 21-th output bit. As shown in Fig. 3, we number S-boxes (hence bits) from the right to the left (little endian representation).

Table 7. 64-bit permutation P: $input = row \cdot 8 + column$.

$a \backslash b$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	12	1	59	49	50	26	9	35
1	24	6	31	60	0	48	46	18
2	33	52	15	21	56	19	47	40
3	8	51	5	30	61	29	27	10
4	36	16	57	7	32	43	45	58
5	23	54	62	37	55	38	14	22
6	13	3	4	25	17	53	41	44
7	20	34	39	2	11	28	42	63

B.2 Components used in PUFFIN key-schedule

As for Table 7, Table 9 and Table 8 have to be read in the following way. The value in the row a and column b is the output position of the input bit number $16a + b$ or $8a + b$ depending on the table.

Table 8. The 64-bit selection used in the PUFFIN key-schedule: $input = row \cdot 8 + column$.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	2	122	14	57	88	35	97	51
1	56	62	99	69	45	70	93	50
2	82	13	3	21	31	113	83	100
3	11	22	30	64	40	95	119	49
4	44	53	111	121	28	80	29	120
5	96	54	25	63	23	116	18	8
6	110	17	43	85	15	94	41	71
7	1	90	117	123	37	47	42	38

Table 9. The 128-bit permutation used in the PUFFIN key-schedule: $input = row \cdot 16 + column$.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	21	120	125	109	78	80	115	54
1	112	20	28	19	55	75	40	111
2	44	108	94	86	93	43	67	7
3	114	68	5	74	82	4	53	69
4	22	60	105	102	84	123	110	51
5	118	31	99	16	14	33	127	90
6	57	98	119	66	30	97	52	70
7	91	24	37	92	64	1	36	27
8	23	81	87	13	95	117	0	8
9	124	26	126	17	3	9	101	6
10	34	104	47	62	29	76	71	49
11	107	72	11	18	106	10	25	83
12	46	96	116	48	45	32	15	41
13	38	56	113	61	122	100	79	12
14	50	121	63	88	42	59	39	2
15	85	89	58	73	77	103	35	65

B.3 Differentials used to attack PUFFIN2

The following table contains differentials we use in the attack on PUFFIN2 with r' equal to 4. Since the only active box in the outpu difference is the 11-th one, we only mention the corresponding 4-bit output difference in the column $\delta_r|_{S11}$.

Table 10. Differentials used for attacking PUFFIN2 with $r = 30$ and $r' = 4$.

δ_0	$\delta_r\|_{S11}$	p_*	δ_0	$\delta_r\|_{S11}$	p_*
0x0000000000040003	0x4	$2^{-55.55}$	0x0000c00000000000	0x4	$2^{-60.11}$
0x0000000000040003	0x8	$2^{-58.28}$	0x000000000000000d	0x4	$2^{-60.14}$
0x0000000000040003	0x6	$2^{-58.39}$	0x0000000000090001	0x4	$2^{-60.18}$
0x0000000000040003	0x2	$2^{-58.52}$	0x0000000000040001	0x4	$2^{-60.28}$
0x0000000000040003	0xa	$2^{-59.69}$	0x00009a0000000000	0x4	$2^{-60.29}$
0x0000000000040000	0x4	$2^{-55.72}$	0x000100000000000d	0x4	$2^{-60.29}$
0x0000000000040000	0x8	$2^{-58.25}$	0x0001400000000000	0x4	$2^{-60.33}$
0x0000000000040000	0x2	$2^{-58.42}$	0x00000000000d0001	0x4	$2^{-60.42}$
0x0000000000040000	0x6	$2^{-58.55}$	0x0000400000040001	0x4	$2^{-60.44}$
0x0000000000040000	0xa	$2^{-59.65}$	0x000a000000000006	0x4	$2^{-60.45}$
0x0003600000000000	0x4	$2^{-56.79}$	0x00a0000000000000	0x4	$2^{-60.56}$
0x0003600000000000	0x8	$2^{-59.56}$	0x0001d00000000000	0x4	$2^{-60.64}$
0x0003600000000000	0x2	$2^{-59.72}$	0x0000100000050000	0x4	$2^{-60.72}$
0x0003600000000000	0x6	$2^{-59.83}$	0x0000c000000a0000	0x4	$2^{-60.74}$
0x0000000000000001	0x4	$2^{-57.96}$	0x0000000000003000	0x4	$2^{-60.74}$
0x0000000000000001	0x8	$2^{-60.77}$	0x0000500000000000	0x4	$2^{-60.80}$
0x0000000000000001	0x2	$2^{-60.95}$	0x0003400000040000	0x4	$2^{-60.82}$
0x0000000000000001	0x6	$2^{-60.97}$	0x000100000000000a	0x4	$2^{-60.83}$
0x000d400000000000	0x4	$2^{-58.11}$	0x000d000000000000	0x4	$2^{-60.83}$
0x0003000000000006	0x4	$2^{-58.40}$	0x00b0000000000000	0x4	$2^{-60.87}$
0x000300000000000a	0x4	$2^{-58.44}$	0x0000000000004000	0x4	$2^{-60.94}$
0x000b000000000000	0x4	$2^{-58.46}$	0x0000000000040006	0x4	$2^{-60.96}$
0x000a000000000001	0x4	$2^{-58.52}$	0x00001e0000000000	0x4	$2^{-60.99}$
0x0000000000000008	0x4	$2^{-58.63}$	0x0001300000000000	0x4	$2^{-61.08}$
0x000a400000000000	0x4	$2^{-58.74}$	0x0000800000000000	0x4	$2^{-61.09}$
0x0000000000030000	0x4	$2^{-58.78}$	0x0003000000000001	0x4	$2^{-61.11}$
0x0006000000000001	0x4	$2^{-58.88}$	0x0000000000030001	0x4	$2^{-61.12}$
0x0000100000000000	0x4	$2^{-58.92}$	0x0050000000000000	0x4	$2^{-61.13}$
0x0000100000000000	0x8	$2^{-61.62}$	0x00a00000000a0000	0x4	$2^{-61.13}$
0x0000100000000000	0x2	$2^{-61.75}$	0x0000000000060000	0x4	$2^{-61.14}$
0x0000100000000000	0x6	$2^{-61.95}$	0x0080900000000000	0x4	$2^{-61.16}$
0x000a000000000003	0x4	$2^{-58.99}$	0x0001000000040000	0x4	$2^{-61.24}$
0x0001000000000003	0x4	$2^{-59.18}$	0x0050100000000000	0x4	$2^{-61.25}$
0x000100000000000b	0x4	$2^{-59.22}$	0x0000000000000003	0x4	$2^{-61.26}$
0x00000000000e0000	0x4	$2^{-59.33}$	0x00a0300000000000	0x4	$2^{-61.36}$
0x0000200000000000	0x4	$2^{-59.38}$	0x00a0800000000000	0x4	$2^{-61.39}$
0x0000200000000000	0x8	$2^{-61.79}$	0x0000890000000000	0x4	$2^{-61.40}$
0x0000200000000000	0x2	$2^{-61.90}$	0x0000000000000006	0x4	$2^{-61.40}$
0x0000200000000000	0x6	$2^{-62.26}$	0x00d0000000000000	0x4	$2^{-61.44}$
0x0000700000000000	0x4	$2^{-59.54}$	0x0080800000000000	0x4	$2^{-61.46}$
0x0000000000000004	0x4	$2^{-59.61}$	0x0000400000000003	0x4	$2^{-61.46}$
0x00006a0000000000	0x4	$2^{-59.62}$	0x0000400000040003	0x4	$2^{-61.50}$
0x000d000000000001	0x4	$2^{-59.77}$	0x000d000000040000	0x4	$2^{-61.51}$
0x0000000000090000	0x4	$2^{-59.85}$	0x000b000000000001	0x4	$2^{-61.60}$
0x0006400000000000	0x4	$2^{-59.95}$	0x0000000900000000	0x4	$2^{-61.63}$

δ_0	$\delta_r\|_{S11}$	p_*	δ_0	$\delta_r\|_{S11}$	p_*
0x0000030000000000	0x4	$2^{-61.64}$	0x0001000000030000	0x4	$2^{-61.89}$
0x0000d00000080000	0x4	$2^{-61.65}$	0x0000a000000a0000	0x4	$2^{-61.92}$
0x00004000000a0000	0x4	$2^{-61.67}$	0x000600000000000a	0x4	$2^{-61.94}$
0x0001000000000001	0x4	$2^{-61.70}$	0x0008000000000006	0x4	$2^{-61.99}$
0x0000ba0000000000	0x4	$2^{-61.71}$	0x0001400000030000	0x4	$2^{-62.00}$
0x0001300000040000	0x4	$2^{-61.72}$	0x000b000000040001	0x4	$2^{-62.01}$
0x0300000000090000	0x4	$2^{-61.73}$	0x00a0900000000000	0x4	$2^{-62.01}$
0x0050200000000000	0x4	$2^{-61.78}$	0x0005000000000004	0x4	$2^{-62.01}$
0x0000680000000000	0x4	$2^{-61.80}$	0x0000c80000000000	0x4	$2^{-62.05}$
0x0003100000000000	0x4	$2^{-61.83}$	0x00008a0000000000	0x4	$2^{-62.05}$
0x0000600000050000	0x4	$2^{-61.84}$	0x0000000000003001	0x4	$2^{-62.05}$
0x0000000600000000	0x4	$2^{-61.85}$	0x0000600000040003	0x4	$2^{-62.08}$
0x0000400000060000	0x4	$2^{-61.87}$			

Accelerated Key Search for the KATAN Family of Block Ciphers

Simon Knellwolf

ETH Zurich, Switzerland
FHNW, Switzerland

Abstract. The lightweight block cipher family KTANTAN has been shown vulnerable to meet-in-the-middle key recovery attacks. Its sister family KATAN has a different, notably linear, key expansion, which prevents a straightforward application of the attacks. In this paper, we adapt the meet-in-the-middle strategy to the linear key expansion. As a result we show that KATAN does neither provide ideal security against meet-in-the-middle attacks. For the full KATAN32, the key space can be searched at the cost of $2^{79.3}$ instead of the expected 2^{80} cipher evaluations. For round-reduced variants, more significant speed-up factors are obtained. For example for 127 (of 254) rounds, the complexity is $2^{78.1}$. The best previous attack in the single-key scenario was for 78 rounds.

Keywords: KATAN, key recovery, meet-in-the-middle

1 Introduction

In the last few years a number of lightweight cryptographic primitives have been proposed. A prominent example is the KATAN / KTANTAN family of hardware-oriented block ciphers designed by De Cannière, Dunkelman, and Knežević in 2009. The family consists of six ciphers with block sizes $n = 32, 48$, and 64, each variant coming in two flavours that only differ by the key expansion algorithm. For implementations on constrained devices they provide a very attractive alternative to dedicated stream ciphers such as Grain and Trivium. Additionally, they benefit from a vast number of well established techniques for evaluating the security of block ciphers.

The security against differential cryptanalysis has been studied in [10] and all the six ciphers have shown a comfortable security margin in the single-key scenario. For KATAN this seems also to be true in the related-key scenario [11], whereas for KTANTAN surprisingly effective key recovery attacks have been reported [2]. The latter attacks are based on an unexpected weakness in the key expansion algorithm which was already exploited for meet-in-the-middle attacks in [6,13]. The key expansion of KTANTAN is an irregular hardwired sequence of the original key bits. It was designed to minimize the number of gates for hardware implementations. In contrast, KATAN uses a linear recursion for the key expansion. This prevents a straightforward application of the technique from [6,13], but in this paper we show that the linear key expansion does neither

provide ideal security against meet-in-the-middle key recovery attacks. Our analysis focuses on KATAN, but it applies to other ciphers with linear key expansion as well.

Table 1 provides an overview of known results complemented by the results presented in this paper.

Table 1. Cryptanalytic results for KATAN / KTANTAN. All attacks recover at least parts of the key. Data complexities are given as the number of known or chosen plaintext / ciphertext pairs. Time complexities are given as the number of cipher evaluations. If no time complexity is indicated it is essentially dominated by the query complexity. Memory complexities are very small for all attacks. The full number of rounds is 254 for all ciphers.

Scenario	Cipher	n	Rounds	Data	Time	Reference
single-key	KATAN /	32	78	2^{22}	-	[10]
	KTANTAN	48	70	2^{34}	-	
		48	68	2^{35}	-	
single-key	KATAN	32	127	3	$2^{78.1}$	this paper
		32	127	2^{32}	$2^{77.6}$	
		32	full	3	$2^{79.3}$	
		48/64	127	2	$2^{78.8}$	
		48/64	full	2	$2^{79.5}$	
single-key	KTANTAN	32	full	4	$2^{72.9}$	[13]
		48	full	4	$2^{73.8}$	
		64	full	4	$2^{74.4}$	
related-key	KATAN	32	120	2^{31}	-	[11]
		48	103	2^{25}	-	
		64	90	2^{27}	-	
related-key	KTANTAN	32	full	30	$2^{28.5}$	[2]
		48	full	35	$2^{31.8}$	
		64	full	30	$2^{32.8}$	

The first use of a meet in the middle technique in a cryptanalytic context was given by Diffie and Hellman [8]. Only recently, meet in the middle became the main technique for preimage attacks on hash functions. Starting with Aoki and Sasaki [4] it has been improved and refined in a series of papers. Important refinements that we use in this paper are partial matching [4], indirect partial matching [3], splice and cut [4], and initial structure [12] resp. bicliques [9]. Our analysis is based on ideas introduced in [1] and has similarities with the recent key recovery attacks on AES [5].

The paper is organized as follows. Section 2 describes KATAN. Section 3 introduces the accelerated key search. The technique is optimized in Section 4 by exploiting properties of the round function. Finally, Section 5 concludes the paper.

2 Description of KATAN

KATAN is a family of lightweight block ciphers designed by De Cannière, Dunkelman, and Knežević. A full specification can be found in [7]. The family consists of three ciphers denoted by KATANn for $n = 32, 48, 64$ indicating the block size. All instances accept a 80-bit key. KATANn has a state of n bits consisting of two non-linear feedback shift registers. For $n = 32$, the registers have lengths 13 and 19, respectively. They are initialized with the plaintext:

$$(s_0, \ldots, s_{18}) \leftarrow (p_0, \ldots, p_{18})$$
$$(s_{19}, \ldots, s_{31}) \leftarrow (p_{19}, \ldots, p_{31}).$$

The key is expanded to 508 bits according to the linear recursion

$$k_{i+80} = k_i + k_{i+19} + k_{i+30} + k_{i+67}, \quad 0 \leq i < 428,$$

where $k_0, \ldots, k_{79}$ are the bits of k. At each round of the encryption process two consecutive bits of the expanded key are used. The round updates further depend on a bit c_i. The sequence of c_i is produced by an 8-bit linear feedback shift register which is used as a counter. It is initialized by $(c_0, \ldots, c_7) = (1, \ldots, 1, 0)$ and expanded according to $c_{i+8} = c_i + c_{i+1} + c_{i+3} + c_{i+5}$.

Round i corresponds to the following transformation of the state:

$$\begin{aligned} t_1 &\leftarrow s_{31} + s_{26} + s_{27}s_{24} + s_{22}c_i + k_{2i} \\ t_2 &\leftarrow s_{18} + s_7 + s_{12}s_{10} + s_8s_3 + k_{2i+1} \\ (s_0, \ldots, s_{18}) &\leftarrow (t_1, s_0, \ldots, s_{17}) \\ (s_{19}, \ldots, s_{31}) &\leftarrow (t_2, s_{19}, \ldots, s_{30}) \end{aligned}$$

After 254 rounds, the state is output as the ciphertext. All three members of the KATAN family use the same key expansion and the same sequence of c_i. The algebraic structure of the non-linear update functions is the same. They differ in the length of the registers and the tap positions. All members perform 254 rounds, but for KATAN48 the non-linear registers are updated twice per round and for KATAN64 even thrice (using the same c_i and k_i for all updates at the same round).

3 Accelerated Key Search

In this section we describe a technique that exploits the linear key expansion based on a meet in the middle strategy. It provides a (typically small) speed-up of the naive exhaustive search by not recomputing large parts of the cipher for specifically related keys. The key space is partitioned into affine subsets of size 2^{2d} such that each set can be tested at a cost lower than 2^{2d} cipher evaluations.

The technique is derived from accelerated preimage search introduced in [1]. A similar idea underlies the technique *matching with precomputation* recently used for key recovery attacks on AES [5].

3.1 Partitioning the Key Space

Let $f : \{0,1\}^{80} \to \{0,1\}^{2N}$, for $0 \le N \le 254$, be the linear map describing the key expansion algorithm of KATAN, where N is the number of rounds. $K = \mathrm{im}(f)$ is a vector space of dimension 80 which is naturally identified with the key space. For $s \le N$, we define two linear subspaces $U, V \subset K$ as follows:

$$U = \{k \in K \mid k_0 = 0, \ldots, k_{2s-1} = 0\},$$
$$V = \{k \in K \mid k_{2N-2s} = 0, \ldots, k_{2N-1} = 0\}.$$

Both subspaces have dimension $d = 80 - 2s$, and respective bases can be easily computed by forward and backward computation or by Gaussian elimination. It turns out that $U \cap V = \{0\}$ if $N \ge 40$ and $s < N$. Hence, K can be partitioned into affine sets of the form

$$k \oplus U \oplus V = \{k \oplus u \oplus v \mid u \in U, v \in V\}.$$

Each subset has size 2^{2d}. We call k the base key of the set. The partition can be described by set of 2^{80-2d} base keys that are different modulo $U \cup V$. Such a set can be easily enumerated.

3.2 Attack Procedure

Let us see how to test the 2^{2d} keys in $k \oplus U \oplus V$ faster than 2^{2d} cipher evaluations. The main observation is that all keys in $k \oplus u \oplus V$, for $u \in U$ fixed, have identical round keys at the last s rounds. In the same way, the keys in $k \oplus U \oplus v$, for $v \in V$ fixed, have the same round keys at the first s rounds. Given a plaintext / ciphertext pair (p, c) an attacker proceeds as follows:

1. Compute a list of 2^d values q_v obtained by encrypting p through the rounds 0 to $s - 1$ under key $k \oplus v$ for $v \in V$.
2. Compute a list of 2^d values q_u obtained by decrypting c through the rounds $N - 1$ to $N - s$ under the key $k \oplus u$ for $u \in U$.
3. For each pair (q_u, q_v) check if q_u decrypts to q_v through the rounds $N - s + 1$ to s under key $k \oplus u \oplus v$. If yes, save $k \oplus u \oplus v$ as a candidate key.
4. Check candidate keys with one or two additional text pairs.

3.3 Complexity Analysis

The procedure works for any variant with $N \ge 40$ rounds. It has to be repeated 2^{80-2d} times to test the entire key space. This results in a total cost of

$$2^{80-d}(C_1 + C_2) + 2^{80}C_3 + 2^{80-n} + 2^{80-2n},$$

where $C_1, C_2 = s/N$ are the costs for the partial encryptions resp. decryptions at the steps 1 and 2, and $C_3 = (N - 2s)/N$ is for the checking at step 3. The last two summands are for checking the candidates. For block sizes $n = 48, 64$ only one additional text pair must be used and the last summand disappears. The cost essentially depends on the choice of s. Figure 1 shows that a cost of 79.5 is obtained with $s = 37 \pm 1$.

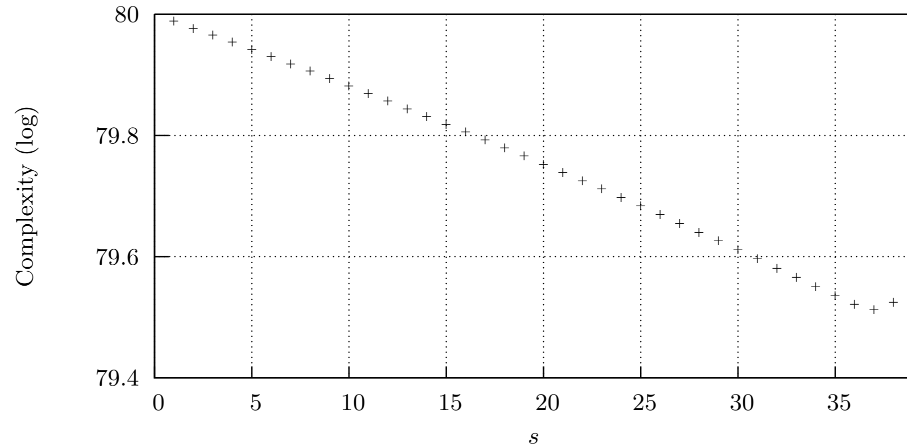


Fig. 1. Complexity of accelerated search depending on the parameter s. Here, $N = 254$, but the curve is qualitatively the same for reduced variants.

4 Optimizations Specific to the Round Transformation

The procedure as described so far applies to all members of the KATAN family and can be translated to other ciphers that use only a small part of a linearly expanded key at each round. The method is completely independent of the round transformation. Exploiting specific properties of the latter allows some improvements.

We are combining partial matching and biclique techniques with accelerated search, still aiming for a method that applies to all number of rounds. The details are described for block size $n = 32$, but a similar analysis applies to the other block sizes.

4.1 Partial Matching

For the following analysis $s = 37$ is fixed (based on Fig. 1). The state of KATAN32 consists of two non-linear shift registers of length 13 and 19, respectively. At each round only one bit per register is updated, and the corresponding round key is added linearly. As a consequence, the bits $s_{14}, \ldots, s_{18}$, and s_{31} after round 57 depend only linearly on $k_{74}, \ldots, k_{79}$. This leads to the following modification of the procedure:

– At step 1, we encrypt until round 57, but we also store the intermediate states after round 36. Step 2 is not modified.

– At step 3, we decrypt until round 58, apply a linear correction (depending on u) to the bits $s_{14}, \dots, s_{18}, s_{31}$ and compare them with the corresponding bits computed at step 1. For each match, we check the remaining state after round 57 by encrypting the intermediate state from step 1 under the key $k \oplus u \oplus v$. If the whole state matches, we save $k \oplus u \oplus v$ as a candidate right key.

The average cost of the modified procedure is given by

$$2^{80-d}(C_1 + C_2) + 2^{80}C_3 + 2^{80-r}C_4 + 2^{80-n} + 2^{80-2n},$$

where $C_1 = 58/N$, $C_2 = 37/N$, $C_3 = (N - 58 - 37)/N$, and $C_4 = (58 - 37)/N$ is the cost for checking candidate keys (those who match on the $r = 6$ bits after round 57).

4.2 Splice and Cut

The idea of the splice and cut technique is to start the computations at an intermediate round. This gives some freedom for the choice of the subspaces U and V. One of them is not required to have zero differences at at consecutive rounds, but on two consecutive parts at the beginning and at the end. Let s' be the total number of rounds covered in both parts. Splice and cut provides an advantage if $\Delta s = s' - s > 0$. Then, the total cost can be reduced by about $\Delta s/N \cdot 2^{80}$. We computed Δs for all $d = \{4, 6, 8\}$, $N \in \{128, \dots, 254\}$, and starting points $t \in \{0, \dots, N\}$ (see Appendix A). In most cases, Δs is 0, and it is never larger than 4.

We conclude that there is a very small benefit from the splice and cut technique for variants with a particular number of rounds. However, the technique significantly increases the data complexity, because each time when computing through round 0, the cipher must be queried. If the starting point is not very close to 0, the search essentially requires the whole codebook.

4.3 Bicliques

Bicliques are a formalization of initial structures. The idea is to precompute two sets of 2^d states $\{x_u \mid u \in U\}$ and $\{y_v \mid v \in V\}$ for each base key k such that the following holds: for all $(u, v) \in U \times V$, x_u encrypts to y_v through the rounds i to $i + \ell - 1$ under key $k \oplus u \oplus v$. Then, during the key search phase, encryption with $k \oplus v$ starts from y_v, and decryption with $k \oplus u$ starts from x_u. We call ℓ the length of the biclique and d its dimension. The use of bicliques can decrease the total cost of the attack by about $\ell/N \cdot 2^{80}$ if the following conditions are satisfied:

– For each base key of the partition a suitable biclique exists.
– The total cost for finding the bicliques is negligible compared to the total cost of the attack.

The second condition is not a concern if the bicliques are shorter than 40 rounds. Then, $80 - 2\ell$ key bits can be chosen independently from the biclique and the cost for finding bicliques can be amortized. The first condition is more restrictive.

In combination with the splice and cut technique, the position of the biclique can be freely chosen. It turns out that the position is not very important, and we focus on bicliques from round 0 to round $\ell - 1$. For different ℓ we analyzed the propagation of single bit differences through these rounds. For $0 \leq i < 80$, let δ_i denote the key difference with a 1 at position i and 0's otherwise. We say that δ_i and δ_j propagate independently if $\Delta p = p \oplus p'' = 0$, where p and p'' satisfy

$$p \xrightarrow[k]{\text{enc}} c \xrightarrow[k \oplus \delta_i]{\text{dec}} p' \xrightarrow[k \oplus \delta_i \oplus \delta_j]{\text{enc}} c' \xrightarrow[k \oplus \delta_j]{\text{dec}} p''.$$

Here, encryption and decryption is only from round 0 to $\ell - 1$. It turns out that all single bit differences propagate independently from each other for $\ell \leq 10$. This implies that bicliques up to length 10 always exist and can be easily found. For $\ell = 11$, the differences δ_1 and δ_{20} influence each other. Careful analysis shows that $\Delta p_{25} = 1, \Delta p_{29} = p_{22}$, and $\Delta p_{30} = 1$. A biclique only exists for U and V with zero differences either at position 1 or at position 20. This can be satisfied at the cost of a lower dimension d or smaller s. In both cases, the benefit of having a longer biclique is lost.

For larger ℓ, the number of conditions grows very quickly. Considering other parameters N or bicliques starting at some intermediate round, bicliques of length 11, 12, or even 13 can be found in some very specific cases (where U and V satisfy the conditions automatically). Longer bicliques have not been found with $d \geq 4$.

The complexity illustrated in Fig. 2 considers a biclique of length 10 combined with partial matching. It is compared to accelerated search and accelerated search with partial matching. For the latter, the data complexity is very low, but when using bicliques, essentially the whole codebook is required.

5 Conclusion

We presented an accelerated key search technique for the KATAN family of lightweight block ciphers. The entire key space can be searched at the cost of $2^{79.5}$ cipher evaluations instead of the expected 2^{80}. Exploiting specific properties of the round function, the complexity can be lowered to $2^{79.3}$ for KATAN32 using the partial matching technique. Similar improvements are possible for KATAN48 and KATAN64. The technique requires no more than two or three known plaintext / ciphertext pairs. It applies the other block ciphers with linear key expansion as well.

The applicability of splice and cut and biclique techniques have been studied. For the full cipher variants they do not provide a significant speed-up, but require the whole codebook to be known.

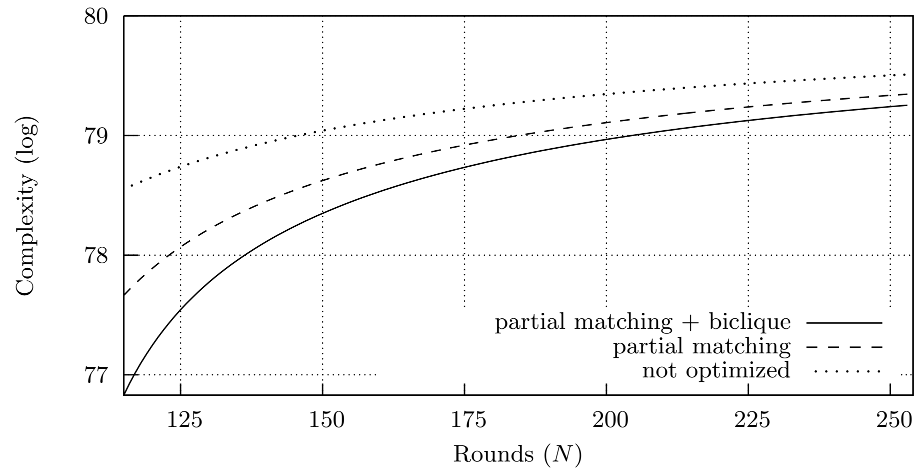


Fig. 2. Performance of accelerated search and its optimizations.

For the KATAN family, the speed-up factor is too small for considering our results as an attack on the full ciphers. To the contrary, we confirm that the designer's choice for 254 rounds was very appropriate, lying on the edge between speed and security. For example, with only 192 rounds, the accelerated key search would provide a speed-up factor 2 over the naive exhaustive search.

Acknowledgements. We thank the reviewers of the ECRYPT workshop on lightweight cryptography for their very helpful comments. This work was supported by the European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II.

References

1. On the Preimage Resistance of SHA-1 (2011), In submission.
2. Ågren, M.: Some Instant- and Practical-Time Related-Key Attacks on KTANTAN32/48/64. In: Miri, A., Vaudenay, S. (eds.) Selected Areas in Cryptography (2011)
3. Aoki, K., Guo, J., Matusiewicz, K., Sasaki, Y., Wang, L.: Preimages for Step-Reduced SHA-2. In: Matsui, M. (ed.) ASIACRYPT. Lecture Notes in Computer Science, vol. 5912, pp. 578–597. Springer (2009)
4. Aoki, K., Sasaki, Y.: Preimage Attacks on One-Block MD4, 63-Step MD5 and More. In: Avanzi, R.M., Keliher, L., Sica, F. (eds.) Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 5381, pp. 103–119. Springer (2008)

5. Bogdanov, A., Khovratovich, D., Rechberger, C.: Biclique cryptanalysis of the full aes. Cryptology ePrint Archive, Report 2011/449 (2011), http://eprint.iacr.org/, Accepted for publication at ASIACRYPT.
6. Bogdanov, A., Rechberger, C.: A 3-Subset Meet-in-the-Middle Attack: Cryptanalysis of the Lightweight Block Cipher KTANTAN. In: Biryukov, A., Gong, G., Stinson, D.R. (eds.) Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 6544, pp. 229–240. Springer (2010)
7. Cannière, C.D., Dunkelman, O., Knežević, M.: KATAN and KTANTAN - A Family of Small and Efficient Hardware-Oriented Block Ciphers. In: Clavier, C., Gaj, K. (eds.) CHES. Lecture Notes in Computer Science, vol. 5747, pp. 272–288. Springer (2009)
8. Diffie, W., Hellman, M.: Special Feature Exhaustive Cryptanalysis of the NBS Data Encryption Standard. Computer 10, 74–84 (1977)
9. Khovratovich, D., Rechberger, C., Savelieva, A.: Bicliques for Preimages: Attacks on Skein-512 and the SHA-2 family. Cryptology ePrint Archive, Report 2011/286 (2011), http://eprint.iacr.org/
10. Knellwolf, S., Meier, W., Naya-Plasencia, M.: Conditional Differential Cryptanalysis of NLFSR-Based Cryptosystems. In: Abe, M. (ed.) ASIACRYPT. Lecture Notes in Computer Science, vol. 6477, pp. 130–145. Springer (2010)
11. Knellwolf, S., Meier, W., Naya-Plasencia, M.: Conditional Differential Cryptanalysis of Trivium and KATAN. In: Miri, A., Vaudenay, S. (eds.) Selected Areas in Cryptography (2011)
12. Sasaki, Y., Aoki, K.: Finding Preimages in Full MD5 Faster Than Exhaustive Search. In: Joux, A. (ed.) EUROCRYPT. Lecture Notes in Computer Science, vol. 5479, pp. 134–152. Springer (2009)
13. Wei, L., Rechberger, C., Guo, J., Wu, H., Wang, H., Ling, S.: Improved Meet-in-the-Middle Cryptanalysis of KTANTAN (Poster). In: Parampalli, U., Hawkes, P. (eds.) ACISP. Lecture Notes in Computer Science, vol. 6812, pp. 433–438. Springer (2011)

A Splice and Cut Analysis

For $N \in \{128, \ldots, 254\}$ and $d \in \{4, 6, 8\}$, Table 2 shows the maximal benefit $\Delta s = s' - s$ over all starting points $t \in \{0, \ldots, N\}$ using the splice and cut technique (see Section 4).

Table 2. Maximal values for Δs.

N	d			N	d			N	d			N	d		
	4	6	8		4	6	8		4	6	8		4	6	8
128	1	1	0	129	2	1	0	130	2	1	0	131	2	0	0
132	1	1	1	133	2	2	0	134	3	0	0	135	1	0	0
136	0	0	0	137	0	0	0	138	0	0	0	139	1	1	0
140	2	0	0	141	0	0	0	142	0	0	0	143	0	0	0
144	1	0	0	145	1	0	0	146	1	1	0	147	2	1	1
148	2	2	2	149	3	3	0	150	4	0	0	151	1	0	0
152	0	0	0	153	0	0	0	154	1	1	0	155	2	0	0
156	0	0	0	157	0	0	0	158	0	0	0	159	1	0	0
160	1	0	0	161	0	0	0	162	1	1	1	163	2	2	2
164	3	3	0	165	4	0	0	166	1	0	0	167	1	0	0
168	0	0	0	169	0	0	0	170	0	0	0	171	1	1	0
172	2	0	0	173	1	1	0	174	2	0	0	175	1	1	0
176	2	0	0	177	0	0	0	178	0	0	0	179	1	0	0
180	0	0	0	181	0	0	0	182	0	0	0	183	0	0	0
184	0	0	0	185	0	0	0	186	1	0	0	187	0	0	0
188	0	0	0	189	1	0	0	190	1	0	0	191	0	0	0
192	0	0	0	193	1	0	0	194	1	0	0	195	1	0	0
196	1	0	0	197	0	0	0	198	1	0	0	199	1	0	0
200	1	0	0	201	1	0	0	202	0	0	0	203	0	0	0
204	0	0	0	205	0	0	0	206	1	0	0	207	0	0	0
208	1	1	0	209	2	0	0	210	0	0	0	211	1	0	0
212	1	1	0	213	2	0	0	214	1	0	0	215	0	0	0
216	1	0	0	217	1	0	0	218	0	0	0	219	0	0	0
220	0	0	0	221	0	0	0	222	1	0	0	223	0	0	0
224	1	0	0	225	1	1	0	226	2	0	0	227	0	0	0
228	1	0	0	229	0	0	0	230	0	0	0	231	0	0	0
232	1	0	0	233	1	1	0	234	2	0	0	235	1	0	0
236	1	0	0	237	1	0	0	238	1	0	0	239	1	0	0
240	1	0	0	241	0	0	0	242	1	0	0	243	0	0	0
244	1	0	0	245	1	1	1	246	2	2	0	247	3	0	0
248	1	0	0	249	0	0	0	250	1	1	0	251	2	0	0
252	1	0	0	253	1	0	0	254	1	1	0				

Some preliminary studies on the differential behavior of the lightweight block cipher LBlock

Marine Minier[1] and María Naya-Plasencia[2]

[1] Université de Lyon, INRIA, CITI, F-69621, France
[2] University of Versailles, France

Abstract. LBlock is a new lightweight block cipher proposed by Wu and Zhang at ACNS 2011 [16]. It is based on a modified 32-round Feistel structure. It uses keys of length 80 bits and message blocks of length 64 bits.
In this paper, we examine the security arguments given in the original articles and show that the bounds given for differential attacks on 12 and 13 rounds can be improved. In the same way, we can improve the impossible differential attack given in the original article on 20 rounds by constructing a 22-round related key impossible differential attack that relies on intrinsic weaknesses of the key schedule.

1 Introduction

During the last five years, many lightweight block ciphers for constrained environments have been proposed. We could cite: PRESENT [1], HIGH [7], mCrypton [11], DESL [10], CGEN [13], MIBS [8], KATAN & KTANTAN [3], TWIS [12], SEA [15], LED [6], KLEIN [5], Piccolo [14] and LBlock [16].

Even if some cryptanalytic results (see [4, 2, 9] for example) have already appeared concerning the security of those particular block ciphers, it still remains necessary to intensively study their security and their efficiency. Moreover, when designing lightweight block ciphers, a particular care must be taken to the design of the key schedule. The reason is that it is not always possible to store the round-keys generated by the key schedule on small platforms due to their limited memory. In that case, the round-keys must be generated "on the fly". This problem has been carefully addressed in the case of CGEN [13].

In this paper, we focus on the security evaluation of the new lightweight block cipher LBlock [16]. We essentially show that the bounds given in the original article for differential cryptanalysis applied on 12 and 13 rounds can be improved and we show how the original impossible differential attack proposed in the LBlock article can be extended by two rounds using a related key impossible differential attack.

This paper is organized as follows: Section 2 gives a brief description of the LBlock lightweight block cipher, Section 3 gives some results concerning the differential distinguishers and attacks whereas in Section 4 we describe the related key impossible differential attack on 21 and 22 rounds of LBlock. Finally, Section 5 concludes this paper.

2 Description of LBlock

LBlock is a new lightweight block cipher presented by Wu and Zhang at ACNS 2011 [16]. It uses 80-bit keys and 64-bit blocks and is based on a modified 32-round Feistel structure (see Fig. 1).

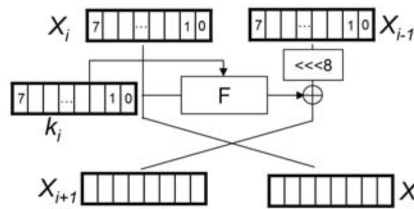


Fig. 1. Overview of one modified Feistel round of LBlock (the numbering corresponds with the nibble ordering notation).

The round function F first computes $X_i \oplus k_i$ and then applies a transformation S (composed of 8 parallel applications of 8 different 4-bit bijective S-boxes) and a permutation P (that exchanges the places of the nibbles as shown on Fig. 2).

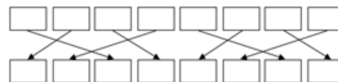


Fig. 2. The permutation P seen as nibble level.

The key schedule takes as input a master key K seen as a key register denoted at bit level as $K = K_{79}K_{78}\cdots K_0$ and outputs 32 round subkeys k_i. It repeats the following steps for $i = 1$ to 31 knowing that k_1 is initialized with the 32 leftmost bits of the key register K:

1. $K <<< 29$
2. $[K_{79}K_{78}K_{77}K_{76}] = S_9[K_{79}K_{78}K_{77}K_{76}]$ where S_9 is the ninth S-box.
3. $[K_{75}K_{74}K_{73}K_{72}] = S_8[K_{75}K_{74}K_{73}K_{72}]$ where S_8 is the eighth S-box.
4. $[K_{50}K_{49}K_{48}K_{47}] = [K_{50}K_{49}K_{48}K_{47}] \oplus [i]_2$
5. k_{i+1} is selected as the leftmost 32 bits of the key register K.

3 Improved Differential Analysis

In [16], a reduced differential analysis of the cipher is performed, where only the maximal number of active Sboxes per number of rounds is given. The authors conclude that no useful characteristic for 15 rounds exists as the probability of such a path would be smaller than 2^{-64}, being 2^{64} the size of the code book.

We have performed a more detailed analysis, and considered truncated adaptive differentials. We have found the characteristic represented in Fig. 3 that we use in our analysis. The initial states are 0 and 1. After i rounds, state i is the right half of the feistel state and state $i+1$ the left one. In the figure, X represents a nibble with a difference, while the remaining nibbles have no difference (we denote this by 0). The colored nibbles represented by a, b and c in Fig. 3 are the words that we will use for sampling and generating more pairs that verify the path with a lower cost. The colored active nibbles (in yellow and gray) are the ones that will impose the conditions that define the probability of the path. In the right side of the figure we can see the instant where the nibble conditions are imposed and the number of them (2 each time). Each 4-bit condition has a probability of $1/15 \approx 2^{-3.9}$ of being satisfied. In total the probability of verifying the path, that has 2×6 4-bit conditions, will be $2^{-3.9\times 12} = 2^{-46.88}$. We will first consider a distinguisher on the first 12 rounds. We will then extend the attack on 13 rounds to recover few bits of information on the key.

3.1 Distinguisher for 12 Rounds

After 12 rounds, we can see in Fig. 3 that the right-most 32-bit half of the state will have a $XX000000$ difference, and the left-most 32-bit half of the state will have a $XXX0X0XX$ difference. That is 8 nibbles won't have any difference. For 64-bit values, this happens in the random case with a probability of 2^{-32} while it will happen due to our path with a probability of $2^{-46.88}$.

This means that if we try $2^{46.88}$ input pairs, we will find $2^{14.88}$ pairs that have the wanted difference in the output, but only one that satisfies the differential path. For finding out which is the pair that satisfies the differential path, we will use the colored nibbles.

If we have a pair of inputs verifying the differential path, and we make all the $4 \times 4 = 16$ input nibbles marked with an a, b or c to take different values (while keeping the same ones for the remaining nibbles), the probability for each of them to satisfy again the differential path will be of $2^{-3.9\times(2\times 2)} = 2^{-15.6}$, as the first 4×2 nibble conditions are still assured. As we have 2^{16} pairs to try, if we find that one of them provides the wanted output difference, this will mean that the original input pair that we tried verifies the differential path. Otherwise it won't, as in the random case this will happen with a probability of 2^{-32}.

This can be used to distinguish 12 rounds of LBlock with an a priori complexity of $2^{46.88} + 2^{14.88+15.6} \simeq 2^{46.88}$.

This complexity can be reduced by considering the fact that we do not need to compute all the $2^{46.88}$ pairs, but we can store one ordered list of size 2^{24}. With 2^{24} elements we can build $\binom{2^{24}}{2} = 2^{46.99}$ pairs, which is slightly more than

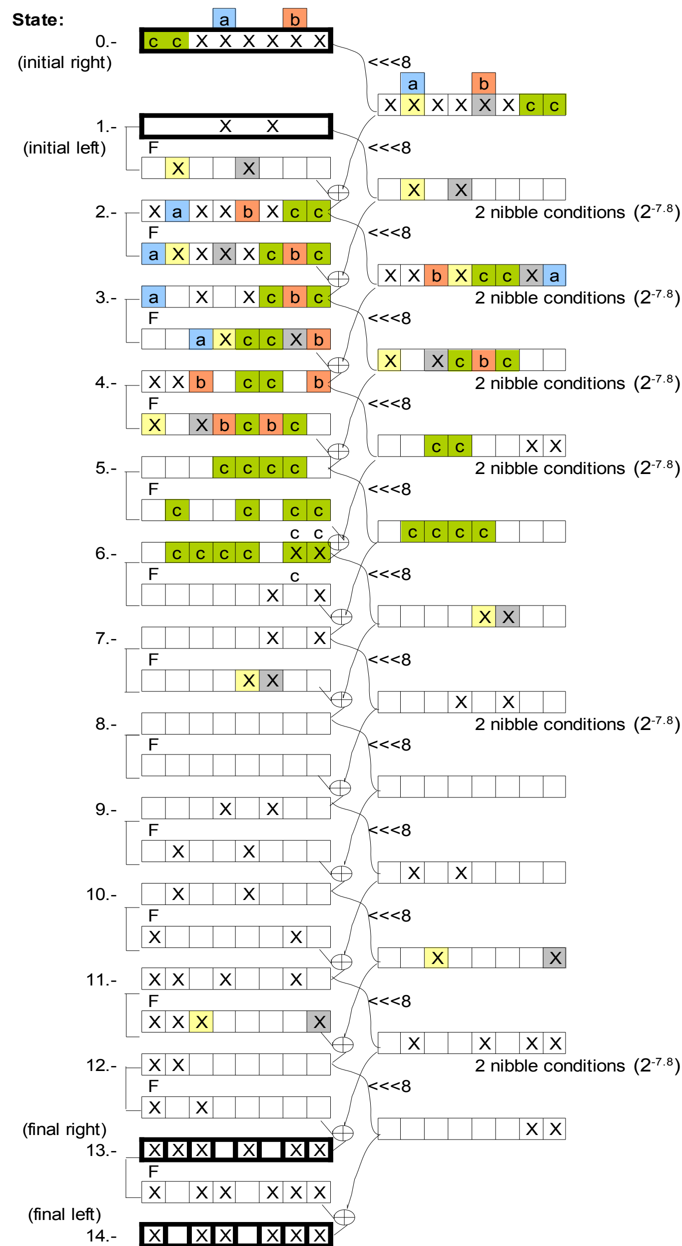


Fig. 3. Differential path used in our analysis. The X nibbles represent differences.

what we need to perform our analysis. In this list we will store the inputs and outputs associated to the 2^{24} input values in state 0 : $AABBBBBB$ and in

state 1 : $AAABABAA$, where the nibbles at positions A represent the words fixed to some random values that are the same for all the 2^{24} elements of the list. The words represented by B will take different values for each element in the list. The list will be ordered by the values of the 8 nibbles in the output where we will look for a 0 difference when performing the distinguisher. Then, for each element in the list we will check if there exists another element that has the same value in the 8 corresponding nibbles. As the verification can be done with a cost of about 1 in an ordered list (or better, with the help of a hash table), the cost of this step will then be the maximum between the size of the list and the number of matches that we expect to find, so $2^{24} + 2^{14.88}$. To compute the overall complexity of this attack we also need to add the one of the final step that remains unchanged and that becomes now the bottleneck. We have then a distinguisher on 12 rounds with a total complexity of $2^{14.88+15.6} = 2^{30.48}$.

Let us remark here that the best differential attack presented in [16] for 12 rounds had a complexity of $2^{24\times 2} = 2^{48}$ compared to the $2^{30.48}$ that we have just proposed.

3.2 Recovering some bits of information on the key for 13 Rounds

Exploiting the distinguisher on 12 rounds, we can recover some information on the key when considering 13 rounds. As we can see in Fig. 3, after 13 rounds the most left half of the state has two non-active nibbles, and the right half of the state has also two non-active nibbles.

This means that the initial crible is less effective as, out of all the $2^{46.48}$ pairs that we will consider, $2^{30.48}$ will pass this first test. In addition, we consider that on average, to one input difference of the LBlock S-boxes, we can associate half of the possible output differences, and that each of these transitions can be done, on average, by two values.

This means that for each of the $2^{30.48}$ pairs kept we have to check if they also verify that the four most-left words with differences in the left-side state of the output have difference that could have been generated from the corresponding nibbles from the right-side state. As we just said, on average, this will happen with a probability of 2^{-4}. Then, out of the $2^{30.48}$ pairs, we will just keep $2^{26.48}$ pairs. The cost so far has been determined by the number of pairs to keep during the first crible, that is $2^{30.48}$.

For each of the $2^{26.48}$ pairs kept so far, we have to check if for the different values of the nibbles marked by a, b or c in the initial state, as done in the 12-round distinguisher, we will also find a pair that will satisfy the path. We then try $2^{15.6}$ different values for these nibbles, leaving the remaining values unchanged. With a probability of $2^{-15.6}$ we will find pairs that have words without difference in the same 4 positions of the output as before, keeping $2^{26.48}$ double pairs. Then, for each of the 2^4 possible values of the 4 nibbles of the subkey determined by the previous crible, we check if the state 12 after the inversion will have the differences as wanted in our path for the new pair obtained. This will happen with a probability of 2^{-16}, as there are 4 additional words with a 0 difference.

In total we will keep $2^{26.48+4-16} = 2^{14.48}$ pairs, and for each of these pairs, the values of 16 bits of the key are already fixed and determined.

This means that we only have $2^{14.48}$ possibilities for 16 bits of the key, meaning that we have recovered $2^{1.52}$ bits of information, or that we have reduced the exhaustive search by that factor.

The bottleneck of this analysis is $2^{26.48+15.6} = 2^{42.08}$. Let us recall that in [16] the best proposed differential analysis on 13 rounds has a complexity of 2^{56}.

4 Related Key Impossible differential Attacks on 21 and 22 rounds of LBlock

The attack described in this section is a related key impossible differential attack. This analysis takes advantage of a similar 14-round impossible differential path than the one presented in the original paper [16] and of some weaknesses of the key schedule. In this section, we first introduce the used related key differential sets and the impossible differential path. We then provide the complete description of the attack.

4.1 Related Key sets

The details of the related key sets are given in Appendix A. The main properties of those related keys come from some intrinsic properties of the key schedule. First, when a low weight difference is introduced in a pair of keys, those differences do not cross the S-boxes every round but in average only every 9 rounds (among 32). Moreover, an injected difference will appear in average only every three subkeys, creating low weight differential paths. Thus, we are able to construct related keys differential paths with a very low general weight (the ones presented in Appendix A have only between 12 and 15 active nibbles on all the 32 subkeys). In summary, the diffusion is not sufficient to correctly spread the differences in the LBlock key-schedule.

However, the four related key differential paths given in Appendix A do not work for all possible values of the bits $K_{75}, K_{74}, K_{73}, K_{72}$ of the key. But, from those four related key paths, we are able to have a complete partition of all possible values. This is due to the small size of the S-boxes that work on nibbles. Moreover, those differentials cross almost always the same S-box s_8 leading to produce always the same differences.

Thus, it will be always possible according to the value of 5 bits (see details on Appendix A) of the master key K to build a second key $K' = K \oplus \Delta K$ with ΔK equal to 0 everywhere except on the nibble $K_{75}, K_{74}, K_{73}, K_{72}$ which takes the value 2 or 4.

4.2 Impossible differential path

In the original paper describing LBlock, the authors give the following 14-round impossible differential:

$$(00000000, 00\alpha00000) \text{ after 14 rounds could not give } (0\beta000000, 00000000)$$

As the differences injected through the subkey additions in our related key sets have really low weight, we are able to continue to construct 14-round impossible differentials even taking into account the differences coming from the subkeys. For example, the following 14-round impossible differential (starting at the beginning of round 5 and ending after round 18) can not happen:

$(00000000, 0000000\alpha)$ after 14 rounds can not give $(00000000, 00000000)$.

This impossible differential works for all the related key paths presented in Appendix A. As we just said, this impossible differential is taken from the fifth round until the 19th round and combined with the first four rounds at the beginning as shown on Fig. 4 and with the last three rounds (four for analysing 22 rounds) in the end as shown on Fig. 5.

4.3 The attack description for 21 rounds

If we consider Fig. 4 and Fig. 5 we see which differences will have the extended impossible differential path in the first and in the last round. In Fig. 4 we show a case that works for 2 out of the 4 possible differential paths, being similar for the other two.

The procedure of our attack is as follows:

- For each one of the 4 possible differential paths in the key schedule, we find m good pairs of input messages that satisfy the extended differential path. This can be done by the limited-birthday approach with a complexity of about $m2^8 2^{32-15} = m2^{25}$. As the partial keybits will be determined only in a second step, we need to build the m set and repeat the following procedure for all the 4 possibilities of the differential path in the key schedule.
- For each of the m good pairs (and for the 4 possible differential paths) we check if the conditions of getting from the input pair to the beginning of the impossible differential, and from the output to the end of the impossible differential, can be verified by some values of the keybits that intervene in these conditions. In total, we have 45 keybits involved.
- The keybits that make both transitions possible for at least one of the m good pairs will be filtered out of the possible key guesses as otherwise they would imply that the impossible differential had occured. We will compute next which size must m have so that we filter all the wrong key guesses.
- From Fig. 4 we can see that there are 7 nibble conditions for errasing the active nibbles and obtaining the differential configuration at the input of the impossible differential. The keybits involved are K_{77} to K_{68}, K_{63} to K_{48}, K_{46} to K_{41}, K_{34} to K_{31}, K_{26} to K_{19} (44 in total).
- From Fig. 5 we can see that there exist 2 nibble conditions for obtaining from the output, the differential configuration of the end of the impossible differential. They involve keybits K_{77} to K_{74}, K_{55} to K_{52}, K_{47} to K_{44} (12 in total and just one not included in the previous set).

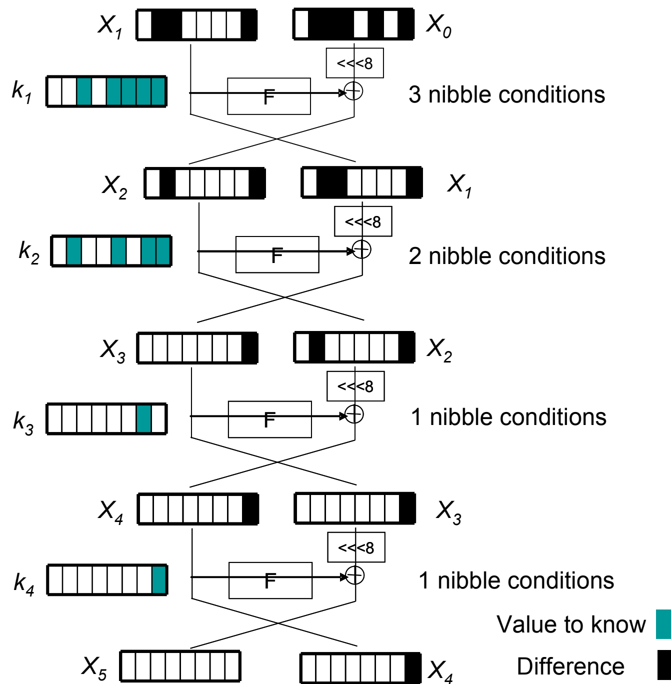


Fig. 4. The initial rounds.

– As the probability that for a good pair, the $7 + 2 = 9$ nibble conditions are verified is 2^{-36}, for each key guess the probability that none of the m good pairs verifies all the conditions is

$$P = (1 - 2^{-36})^m.$$

– We have 2^{45} possibilities for the involved keybits, which means that if we choose $m = 2^{42}$, and so $P \approx 2^{-92.33}$, we will filter out all the wrong key guesses but the correct one.

The complexity of the attack, where we recover 45 keybits (and then the remaining ones with much lower complexity) is then

$$4 * 2^{42+25} + 4 * 2^{42} 2^{45-36} \approx 2^{69},$$

where the first term represents the complexity of obtaining the 2^{42} pairs with the wanted input-outut differences for the 4 differential paths of the key schedule, and the second term comes from the fact that, for each of the 2^{42} pairs of messages, and for the 4 possible key schedule paths, we filter out all the partial keys that verify the conditions. For each one of the m pairs, we have on average

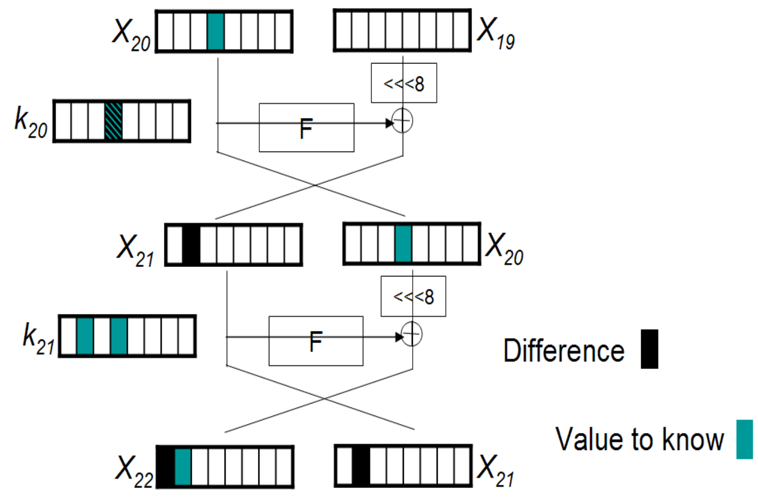


Fig. 5. The final rounds.

$2^{45-36} = 2^9$ such values that verify the 9 nibble conditions for the 45 keybits: we can determine directly, from a given good pair, the values of the involved keybits that will verify the conditions, but this will have the same result as if we performed an exhaustive search on the 45 involved bits, to check for which ones the conditions are verified. As the probability of verifying the conditions is 2^{-36}, for each one of the m pairs we will find about $2^{45-36} = 2^9$ different values verifying the conditions for the 45 keybits.

4.4 Extending the attack to 22 rounds

We can extend the attack by considering one more round in the end. In this case, the procedure is similar as the one described in the previous section. Now, we have 4 active nibbles in the output after 22 rounds.

Now, the cost of finding the m pairs of messages will be $m2^{12}2^{32-23} = m2^{21}$, as now the size of differences in the output is of 12 bits.

Adding the last round we add one extra condition in one nibble plus we involve 12 additional keybits (57 in total): K_{30} to K_{27}, K_{18} to K_{15} and K_6 to K_3. The probability is given now by $P = (1-2^{-40})^m$. Choosing $m = 2^{47}$ makes that $P \approx 2^{-184.66}$, so we can expect that $2^{-184.66+57} = 2^{-127.66}$ wrong guesses remain, so we find the correct key guess with a very high probability.

The complexity in this case for recovering key is

$$4 * 2^{47+21} + 4 * 2^{47} 2^{57-40} \approx 2^{70}.$$

5 Conclusion

We have provided in this article a more detailed analysis of differential and related key impossible differential behaviors of the new lightweight block cipher LBlock. We also show that the bounds originally given in the LBlock article could be improved.

In the differential case, we have shown that assuming that no attack can be build on 15 rounds because the number of active S-boxes is at least 32 is a bit premature, as attacks with smaller complexities than the ones bounded by the authors have been shown here (though they are more evolved). In the same way, we take advantage in the proposed related key impossible differential attack of some particular weaknesses of the key-schedule that could produce differential paths with really low weight for an initial difference carefully chosen.

Finally, we have been able to give the best attack known on LBlock, that works up to 22-rounds, while the analysis for the biggest number of rounds in the original article worked on 20 rounds. We believe that our analysis can still be improved.

References

1. Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In *Cryptographic Hardware and Embedded Systems - CHES 2007*, Lecture Notes in Computer Science 4727, pages 450–466. Springer Verlag, 2007.
2. Andrey Bogdanov and Christian Rechberger. A 3-subset meet-in-the-middle attack: Cryptanalysis of the lightweight block cipher ktantan. In *Selected Areas in Cryptography - SAC 2010*, volume 6544 of *Lecture Notes in Computer Science*, pages 229–240. Springer, 2010.
3. Christophe De Cannière, Orr Dunkelman, and Miroslav Knezevic. KATAN and KTANTAN - A Family of Small and Efficient Hardware-Oriented Block Ciphers. In *Cryptographic Hardware and Embedded Systems - CHES 2009*, Lecture Notes in Computer Science 5747, pages 272–288. Springer Verlag, 2009.
4. Baudoin Collard and François-Xavier Standaert. A Statistical Saturation Attack against the Block Cipher PRESENT. In *Topics in Cryptology - CT-RSA 2009*, Lecture Notes in Computer Science 5473, pages 195–210. Springer Verlag, 2009.
5. Zhen Gong, Svetla Nikova, and Yee-Wei Law. KLEIN: a new family of lightweight block ciphers. In *RFIDSec*, 2011.
6. Jian Guo, Thomas Peyrin, Axel Poschmann, and Matt Robshaw. The led block cipher. In *Workshop on Cryptographic Hardware and Embedded Systems 2011 - CHES 2011*, volume to appear of *Lecture Notes in Computer Science*. Springer, 2011.

7. Deukjo Hong, Jaechul Sung, Seokhie Hong, Jongin Lim, Sangjin Lee, Bonseok Koo, Changhoon Lee, Donghoon Chang, Jaesang Lee, Kitae Jeong, Hyun Kim, Jongsung Kim, and Seongtaek Chee. HIGHT: A New Block Cipher Suitable for Low-Resource Device. In *Cryptographic Hardware and Embedded Systems - CHES 2006*, Lecture Notes in Computer Science 4249, pages 46–59. Springer Verlag, 2006.
8. Maryam Izadi, Babak Sadeghiyan, Seyed Saeed Sadeghian, and Hossein Arabnezhad Khanooki. MIBS: A New Lightweight Block Cipher. In *Cryptology and Network Security - CANS 2009*, Lecture Notes in Computer Science 5888, pages 334–348, 2009.
9. Gregor Leander, Mohamed Ahmed Abdelraheem, Hoda AlKhzaimi, and Erik Zenner. A cryptanalysis of printcipher: The invariant subspace attack. In *Advances in Cryptology - CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 206–221. Springer, 2011.
10. Gregor Leander, Christof Paar, Axel Poschmann, and Kai Schramm. New lightweight des variants. In *Fast Software Encryption - FSE 2007*, Lecture Notes in Computer Science 4593, pages 196–210. Springer Verlag, 2007.
11. Chae Hoon Lim and Tymur Korkishko. mCrypton - A Lightweight Block Cipher for Security of Low-Cost RFID Tags and Sensors. In *Workshop on Information Security Applications - WISA 2005*, Lecture Notes in Computer Science 3786, pages 243–258. Springer Verlqg, 2005.
12. Shrikant Ojha, Naveen Kumar, Kritika Jain, and Sangeeta Lal. TWIS - a lightweight block cipher. In *ICISS*, pages 280–291, 2009.
13. Matthew J. B. Robshaw. Searching for compact algorithms: cgen. In *VIETCRYPT*, pages 37–49, 2006.
14. Kyoji Shibutani, Takanori Isobe, Harunaga Hiwatari, Atsushi Mitsuda, Toru Akishita, and Taizo Shirai. Piccolo: An ultra-lightweight blockcipher. In *CHES*, volume 6917 of *Lecture Notes in Computer Science*, pages 342–357. Springer, 2011.
15. François-Xavier Standaert, Gilles Piret, Neil Gershenfeld, and Jean-Jacques Quisquater. Sea: A scalable encryption algorithm for small embedded applications. In *CARDIS*, pages 222–236, 2006.
16. Wenling Wu and Lei Zhang. Lblock: A lightweight block cipher. In *Applied Cryptography and Network Security - ACNS 2011*, volume 6715 of *Lecture Notes in Computer Science*, pages 327–344. Springer, 2011.

A Related key differences used in the related key impossible differential attack

There are four cases of related key differentials that depend on the value of the five bits K_{76} and $(K_{75}, K_{74}, K_{73}, K_{72})$. According to the values of those bits, the difference that must be injected in the key is 2 or 4 on K_{18}. The Tables 1 and 2 give those differentials on the keys and on the subkeys. We then obtain 4 related key differentials that could be used whatever the 5 bits values are. The four differences are chosen according to:

– If the key bits $(K_{75}, K_{74}, K_{73}, K_{72})$ take the values 0, 1, 4 and 5 and if $K_{76} = 0$, then the good related key differential is the right one given in Tab. 2, else if $K_{76} = 1$, the good related key differential is the right one given in Tab. 1.

- If the key bits $(K_{75}, K_{74}, K_{73}, K_{72})$ take the values 2, 3, 6 and 7 and if $K_{76} = 0$, then the related key differential is the right one given in Tab. 1, else if $K_{76} = 1$, the good related key differential is the right one given in Tab. 2.
- If the key bits $(K_{75}, K_{74}, K_{73}, K_{72})$ take the values 8, 9, 10 and 11 and if $K_{76} = 0$, then the related key differential is the left one given in Tab. 2, else if $K_{76} = 1$, the good related key differential is the left one given in Tab. 1.
- If the key bits $(K_{75}, K_{74}, K_{73}, K_{72})$ take the values 12, 13, 14 and 15 and if $K_{76} = 0$, then the related key differential is the left one given in Tab. 1, else if $K_{76} = 1$, the good related key differential is the left one given in Tab. 2.

As Shown on Tab. 1, the key schedule algorithm does not provide a sufficient diffusion of differences.

Diff Key:	0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Diff Key:	0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Diff SubKey1:	0 2 0 0 0 0 0 0	Diff SubKey1:	0 4 0 0 0 0 0 0
Diff SubKey2:	0 0 0 0 0 0 0 0	Diff SubKey2:	0 0 0 0 0 0 0 0
Diff SubKey3:	0 0 0 0 0 0 0 8	Diff SubKey3:	0 0 0 0 0 0 1 0
Diff SubKey4:	0 0 0 0 0 0 0 0	Diff SubKey4:	0 0 0 0 0 0 0 0
Diff SubKey5:	0 0 0 0 0 0 0 0	Diff SubKey5:	0 0 0 0 0 0 0 0
Diff SubKey6:	0 0 0 0 0 4 0 0	Diff SubKey6:	0 0 0 0 0 8 0 0
Diff SubKey7:	0 0 0 0 0 0 0 0	Diff SubKey7:	0 0 0 0 0 0 0 0
Diff SubKey8:	0 0 0 0 0 0 0 0	Diff SubKey8:	0 0 0 0 0 0 0 0
Diff SubKey9:	0 0 0 2 0 0 0 0	Diff SubKey9:	0 0 0 4 0 0 0 0
Diff SubKey10:	0 0 0 0 0 0 0 0	Diff SubKey10:	0 0 0 0 0 0 0 0
Diff SubKey11:	0 0 0 0 0 0 0 0	Diff SubKey11:	0 0 0 0 0 0 0 0
Diff SubKey12:	0 6 0 0 0 0 0 0	Diff SubKey12:	0 2 0 0 0 0 0 0
Diff SubKey13:	0 0 0 0 0 0 0 0	Diff SubKey13:	0 0 0 0 0 0 0 0
Diff SubKey14:	0 0 0 0 0 0 1 8	Diff SubKey14:	0 0 0 0 0 0 0 8
Diff SubKey15:	0 0 0 0 0 0 0 0	Diff SubKey15:	0 0 0 0 0 0 0 0
Diff SubKey16:	0 0 0 0 0 0 0 0	Diff SubKey16:	0 0 0 0 0 0 0 0
Diff SubKey17:	0 0 0 0 0 c 0 0	Diff SubKey17:	0 0 0 0 0 4 0 0
Diff SubKey18:	0 0 0 0 0 0 0 0	Diff SubKey18:	0 0 0 0 0 0 0 0
Diff SubKey19:	0 0 0 0 0 0 0 0	Diff SubKey19:	0 0 0 0 0 0 0 0
Diff SubKey20:	0 0 0 6 0 0 0 0	Diff SubKey20:	0 0 0 2 0 0 0 0
Diff SubKey21:	0 0 0 0 0 0 0 0	Diff SubKey21:	0 0 0 0 0 0 0 0
Diff SubKey22:	0 0 0 0 0 0 0 0	Diff SubKey22:	0 0 0 0 0 0 0 0
Diff SubKey23:	0 5 0 0 0 0 0 0	Diff SubKey23:	0 3 0 0 0 0 0 0
Diff SubKey24:	0 0 0 0 0 0 0 0	Diff SubKey24:	0 0 0 0 0 0 0 0
Diff SubKey25:	0 0 0 0 0 0 1 4	Diff SubKey25:	0 0 0 0 0 0 0 c
Diff SubKey26:	0 0 0 0 0 0 0 0	Diff SubKey26:	c 0 0 0 0 0 0 0
Diff SubKey27:	0 0 0 0 0 0 0 0	Diff SubKey27:	0 0 0 0 0 0 0 0
Diff SubKey28:	0 0 0 0 0 b 4 0	Diff SubKey28:	0 0 0 0 0 7 0 0
Diff SubKey29:	0 0 0 0 0 0 0 0	Diff SubKey29:	0 0 0 0 0 0 0 0
Diff SubKey30:	0 0 0 0 0 0 0 0	Diff SubKey30:	0 0 0 0 0 0 0 0
Diff SubKey31:	0 0 0 5 a 0 0 0	Diff SubKey31:	0 0 0 3 8 0 0 0
Diff SubKey32:	0 0 0 0 0 0 0 0	Diff SubKey32:	0 0 0 0 0 0 0 0

Table 1. The two first related key differentials used in the attack presented in Section 4. Note that the differential trails from Subkey23 and Subkey26 respectively could have different values from the ones given here.

Diff Key:	0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Diff Key:	0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Diff SubKey1:	0 2 0 0 0 0 0 0	Diff SubKey1:	0 4 0 0 0 0 0 0
Diff SubKey2:	0 0 0 0 0 0 0 0	Diff SubKey2:	0 0 0 0 0 0 0 0
Diff SubKey3:	0 0 0 0 0 0 0 8	Diff SubKey3:	0 0 0 0 0 0 1 0
Diff SubKey4:	0 0 0 0 0 0 0 0	Diff SubKey4:	0 0 0 0 0 0 0 0
Diff SubKey5:	0 0 0 0 0 0 0 0	Diff SubKey5:	0 0 0 0 0 0 0 0
Diff SubKey6:	0 0 0 0 0 4 0 0	Diff SubKey6:	0 0 0 0 0 8 0 0
Diff SubKey7:	0 0 0 0 0 0 0 0	Diff SubKey7:	0 0 0 0 0 0 0 0
Diff SubKey8:	0 0 0 0 0 0 0 0	Diff SubKey8:	0 0 0 0 0 0 0 0
Diff SubKey9:	0 0 0 2 0 0 0 0	Diff SubKey9:	0 0 0 4 0 0 0 0
Diff SubKey10:	0 0 0 0 0 0 0 0	Diff SubKey10:	0 0 0 0 0 0 0 0
Diff SubKey11:	0 0 0 0 0 0 0 0	Diff SubKey11:	0 0 0 0 0 0 0 0
Diff SubKey12:	0 2 0 0 0 0 0 0	Diff SubKey12:	0 6 0 0 0 0 0 0
Diff SubKey13:	0 0 0 0 0 0 0 0	Diff SubKey13:	0 0 0 0 0 0 0 0
Diff SubKey14:	0 0 0 0 0 0 0 8	Diff SubKey14:	0 0 0 0 0 0 1 8
Diff SubKey15:	0 0 0 0 0 0 0 0	Diff SubKey15:	0 0 0 0 0 0 0 0
Diff SubKey16:	0 0 0 0 0 0 0 0	Diff SubKey16:	0 0 0 0 0 0 0 0
Diff SubKey17:	0 0 0 0 0 4 0 0	Diff SubKey17:	0 0 0 0 0 c 0 0
Diff SubKey18:	0 0 0 0 0 0 0 0	Diff SubKey18:	0 0 0 0 0 0 0 0
Diff SubKey19:	0 0 0 0 0 0 0 0	Diff SubKey19:	0 0 0 0 0 0 0 0
Diff SubKey20:	0 0 0 2 0 0 0 0	Diff SubKey20:	0 0 0 6 0 0 0 0
Diff SubKey21:	0 0 0 0 0 0 0 0	Diff SubKey21:	0 0 0 0 0 0 0 0
Diff SubKey22:	0 0 0 0 0 0 0 0	Diff SubKey22:	0 0 0 0 0 0 0 0
Diff SubKey23:	0 2 0 0 0 0 0 0	Diff SubKey23:	0 1 0 0 0 0 0 0
Diff SubKey24:	0 0 0 0 0 0 0 0	Diff SubKey24:	0 0 0 0 0 0 0 0
Diff SubKey25:	0 0 0 0 0 0 0 8	Diff SubKey25:	0 0 0 0 0 0 0 4
Diff SubKey26:	0 0 0 0 0 0 0 0	Diff SubKey26:	c 0 0 0 0 0 0 0
Diff SubKey27:	0 0 0 0 0 0 0 0	Diff SubKey27:	0 0 0 0 0 0 0 0
Diff SubKey28:	0 0 0 0 0 4 0 0	Diff SubKey28:	0 0 0 0 0 2 c 0
Diff SubKey29:	0 0 0 0 0 0 0 0	Diff SubKey29:	0 0 0 0 0 0 0 0
Diff SubKey30:	0 0 0 0 0 0 0 0	Diff SubKey30:	0 0 0 0 0 0 0 0
Diff SubKey31:	0 0 0 2 0 0 0 0	Diff SubKey31:	0 0 0 1 6 0 0 0
Diff SubKey32:	0 0 0 0 0 0 0 0	Diff SubKey32:	0 0 0 0 0 0 0 0

Table 2. The two other related key differentials used in the attack presented in Section 4 that provide a complete partition on all possible key values.

Compact Hardware Implementations of the Ultra-Lightweight Block Cipher *Piccolo*

Harunaga Hiwatari, Kyoji Shibutani, Takanori Isobe, Atsushi Mitsuda, Toru Akishita, and Taizo Shirai

Sony Corporation
1-7-1 Konan, Minato-ku, Tokyo 108-0075, Japan
{Harunaga.Hiwatari,Kyoji.Shibutani,Takanori.Isobe,Atsushi.Mitsuda, Toru.Akishita,Taizo.Shirai}@jp.sony.com

Abstract. In CHES 2011, a 64-bit lightweight block cipher *Piccolo* supporting key lengths of 80 and 128 bits has been proposed. *Piccolo* has a variant of a generalized Feistel structure with an SPS-type F-function, a half-word based round permutation and permutation-based key scheduling. The authors claimed that *Piccolo* achieves both high security and compact implementations. In this paper, we show the detailed descriptions of hardware implementations of *Piccolo*. We make efficient use of equivalent transformations to reduce area requirements for key scheduling including key whitening. Moreover, we show the novel technique for serialized architectures using the inverse of S-box. Our technique can avoid the use of additional registers for intermediate values of F-functions. In addition, we compare the implementation results of *Piccolo* with other lightweight block ciphers in both flexible and fixed-key settings, and show that *Piccolo* is very competitive in both settings.

Key words: block cipher, *Piccolo*, compact hardware implementation, ASIC

1 Introduction

Block ciphers are essential primitives for cryptographic applications such as data integrity, confidentiality, and protection of privacy. At the same time, with the large deployment of low resource devices such as RFID tags and sensor nodes and increasing need to provide security among devices, lightweight cryptography has become a hot topic. Hence, recently, research on designing and analyzing lightweight block ciphers has received a lot of attention. In fact, many ultra-lightweight block ciphers have been designed for use in small embedded devices such as mCrypton [18], HIGHT [13], DESL/DESXL [17], PRESENT [5], KATAN/KTANTAN [8], PRINTcipher [15] and LED [11].

Recently, a 64-bit block cipher *Piccolo* [27] supporting key lengths of 80 and 128 bits has been proposed in CHES 2011. *Piccolo* has a variant of a generalized Feistel structure with a SPS-type F-function, a half-word based round permutation and permutation-based key scheduling. *Piccolo* is designed to achieve

enough immunity against known attacks including recent related-key differential attacks [3, 2] and meet-in-the-middle attacks [6]. From the view of hardware implementations, *Piccolo* achieves both very small requirements on an area footprint and high efficiency on energy consumption. Moreover, *Piccolo* requires relatively small additional area requirements to support decryption due to its Feistel-type structure and permutation-based key scheduling.

In this paper, we show the detailed descriptions of hardware implementations of *Piccolo*. We designed two types of implementations to achieve high efficiency and small area requirements: a round-based implementation and a serialized implementation. While one round function is processed within one cycle in a round-based implementation, only a fraction of one round is treated in a cycle in a serialized implementation. As for round-based implementations, XOR operations with round keys are moved by an equivalent transformation into the data lines where key whitening operations are processed. Thus, XOR gates for round keys and whitening keys area merged, and then 176 GE in total can be saved.

As for serialized implementations, we utilize a novel technique to implement a generalized Feistel structure with SPS-type F-functions. In SAC 2011, compact hardware architectures of the block cipher CLEFIA [28] have been proposed [1]. The authors introduced the serialized implementations of Maximum Distance Separable (MDS) matrices without additional registers in a generalized Feistel structure with SP-type F-functions. However, their technique cannot be applied to *Piccolo* that employs SPS-type F-functions. In our technique, we utilize the inverse of S-box to avoid additional registers for intermediate values of F-functions, which leads to savings of 60 GE. Our technique is also applicable to a Feistel structure and SPSP-type F-functions. Moreover, we show detailed data flow of serialized implementations including how to implement the round permutation efficiently.

We compare the implementation result of *Piccolo* with other lightweight block ciphers in consideration of the difference between the two key input settings: flexible and fixed-key setting. The difference strongly affects the area requirements. We discuss the influence to implementation results from the difference. We also mention the good feature of permutation-based key scheduling part in the flexible-key setting. As a result, it is shown that *Piccolo* is very competitive to other lightweight block ciphers in both key settings.

The rest of the paper is organized as follows. Sect. 2 gives the description of *Piccolo*. Sect. 3 and Sect. 4 describe our implementation techniques for *Piccolo* about round-based and serialized implementation, respectively. In Sect. 5, we discuss the difference between the two key input settings and provide evaluation results for our implementation, compared with the previous results of lightweight block ciphers. Finally, we conclude in Sect. 6.

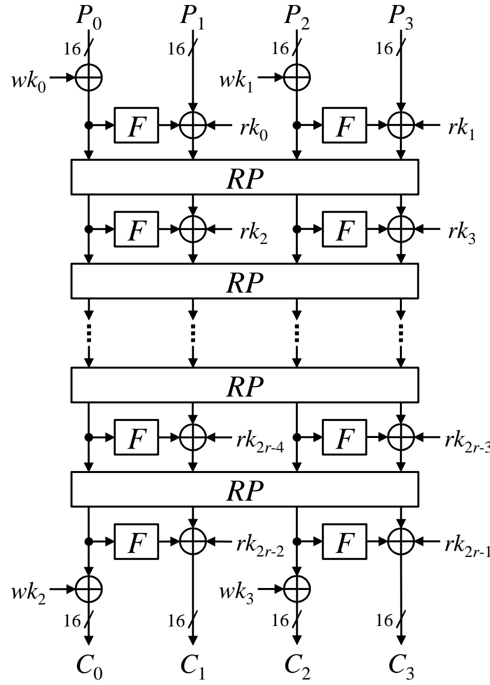


Fig. 1. Encryption function ENC_r

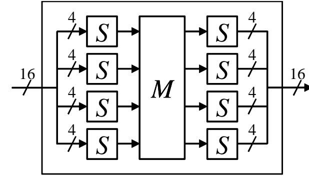


Fig. 2. F-function F

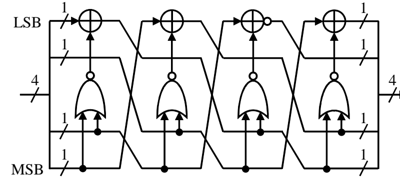


Fig. 3. S-box S

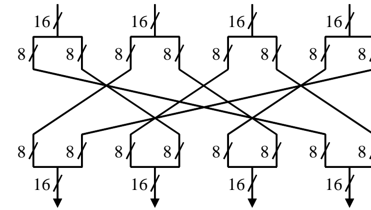


Fig. 4. Round permutation RP

2 Algorithm Description of *Piccolo*

Piccolo [27] is a 64-bit block cipher supporting 80-bit and 128-bit keys, denoted as *Piccolo*-80 and *Piccolo*-128, respectively. *Piccolo* is divided into two parts: the data processing part and the key scheduling part.

The data processing part employs a 4-branch Type-2 generalized Feistel network with a 16-bit F-function F and a 64-bit round permutation RP. The number of rounds r is 25 an 31 for *Piccolo*-80 and -128, respectively. The encryption function ENC_r takes a 64-bit plaintext $P = P_0|P_1|P_2|P_3$, 16-bit whitening keys wk_i $(0 \leq i < 4)$, and 16-bit round keys rk_j $(0 \leq j < 2r)$ as inputs, and outputs a 64-bit ciphertext $C = C_0|C_1|C_2|C_3$ as shown in Fig. 1.

The F-function F consists of two S-box layers between which a diffusion layer is sandwiched (See Fig. 2). The S-box layer consists of four 4-bit bijective S-boxes S, which can be constructed by 4 NOR gates, 3 XOR gates and 1 XNOR gate as shown in Fig. 3. The diffusion matrix M is defined as

$$M = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix}.$$

The multiplication between the matrix and a vector is performed in GF(2^4) defined by a primitive polynomial z^4+z+1. The round permutation RP permutes eight 8-bit data as shown in Fig. 4.

The key scheduling part of *Piccolo*-80 takes a 80-bit key $K = k_0|k_1|k_2|k_3|k_4$ as input and outputs 16-bit whitening keys wk_i $(0 \leq i < 4)$ and 16-bit round keys rk_j $(0 \leq j < 50)$. wk_i and rk_j are defined as

$$wk_0 \leftarrow k_0^L|k_1^R, wk_1 \leftarrow k_1^L|k_0^R, wk_2 \leftarrow k_4^L|k_3^R, wk_3 \leftarrow k_3^L|k_4^R$$
$$(rk_{2j}, rk_{2j+1}) \leftarrow (con_{2j}^{80}, con_{2j+1}^{80}) \oplus (sk_{2j}, sk_{2j+1})$$
$$(sk_{2j}, sk_{2j+1}) \leftarrow \begin{cases} (k_2, k_3) & (j \bmod 5 = 0, 2) \\ (k_0, k_1) & (j \bmod 5 = 1, 3) \\ (k_4, k_4) & (j \bmod 5 = 4), \end{cases}$$

where k_i^L and k_i^R indicate left and right half 8 bits of k_i, respectively, and con_j^{80} $(0 \leq j < 50)$ are constant values for *Piccolo*-80.

For *Piccolo*-128, 16-bit whitening key wk_i $(0 \leq i < 4)$ and 16-bit round keys rk_j $(0 \leq j < 62)$ are generated from a 128-bit key $K = k_0|k_1|k_2|k_3|k_4|k_5|k_6|k_7$ and constant values con_j^{128} $(0 \leq j < 62)$ as follows:

$$wk_0 \leftarrow k_0^L|k_1^R, wk_1 \leftarrow k_1^L|k_0^R, wk_2 \leftarrow k_4^L|k_7^R, wk_3 \leftarrow k_7^L|k_4^R$$
$$(rk_{2j}, rk_{2j+1}) \leftarrow (con_{2j}^{128}, con_{2j+1}^{128}) \oplus (sk_{2j}, sk_{2j+1})$$
$$sk_{2j} \leftarrow \begin{cases} k_0 & (j \bmod 16 = 5, 10, 12, 15) \\ k_2 & (j \bmod 16 = 0, 3, 9, 14) \\ k_4 & (j \bmod 16 = 1, 6, 8, 11) \\ k_6 & (j \bmod 16 = 2, 4, 7, 13) \end{cases}$$
$$sk_{2j+1} \leftarrow \begin{cases} k_1 & (j \bmod 4 = 3) \\ k_3 & (j \bmod 12 = 0, 5, 10) \\ k_5 & (j \bmod 12 = 1, 6, 8) \\ k_7 & (j \bmod 12 = 2, 4, 9). \end{cases}$$

3 Round-Based Implementation

Since it is only necessary to implement a whole round function, a round-based implementation is usually done straightforwardly. However, in this section, we present a novel technique to reduce the area requirement of round-based architecture.

The key scheduling part of *Piccolo* can be implemented by using multiplexers in a way similar to the implementation of GOST and KTANTAN [23, 8]. Since our technique focuses on the data processing part, we omit the detail description about the key scheduling part. Moreover, for ease of explanation, we omit constant values included in key scheduling part for *Piccolo*; we only consider (sk_{2j}, sk_{2j+1}) for round keys (rk_{2j}, rk_{2j+1}).

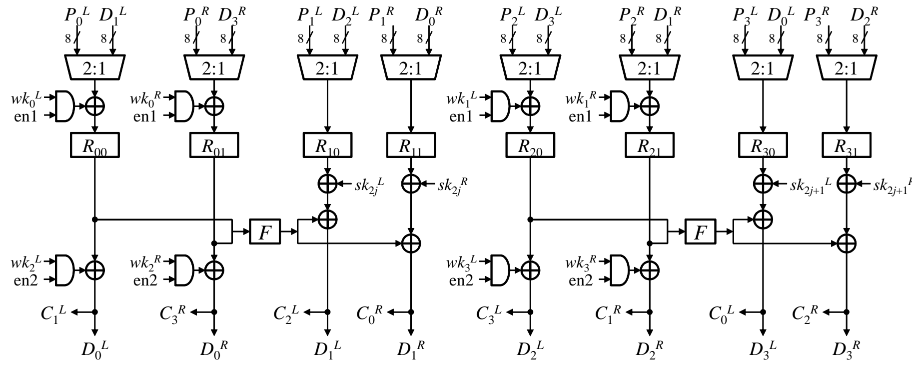


Fig. 5. Straightforward data path of round-based architecture

3.1 Sharing XOR between Round Key and Whitening Key

Piccolo has key whitening operations which are XOR operations with whitening keys at initial and final rounds. The key whitening operations are important from a viewpoint of security. However, the area overhead is not negligible.

We describe a straightforward data path of *Piccolo* in Fig. 5. Note that X^L and X^R indicate left and half 8 bits of a 16-bit X. When inputing a 64-bit plaintext $P = P_0|P_1|P_2|P_3$, the key whitening operations are done by setting an enable signal to high. Then, the intermediate results D_i^L, D_i^R $(0 \leq i \leq 3)$ are calculated and stored in the registers R_{ij} $(0 \leq i \leq 3,\ 0 \leq j \leq 1)$ according to the round permutation RP. In the process of the final round, the key whitening operations are executed by setting an enable signal en2 to high before a ciphertext is output. Fig. 5 shows that we require additional eight 8-bit XORs and eight 8-bit ANDs for the key whitening operations. Those additional gates cost totally 192 GE in the library we use. Note that we actually use negative logic to reduce the area requirement in the logic synthesis, although we describe the architecture by using positive logic.

In order to minimize additional gates for the key whitening operations, we propose a new implementation technique. We move the XOR operations with round keys into next round beyond the round permutation by an equivalent transformation as shown in Fig. 6. The inverse permutation of round permutation RP is denoted as RP^{-1}. The encryption process in Fig. 6 (b) executes XOR operations with the round keys and the whitening keys at only two lines: the 1st line and the 3rd line from left. According to the encryption process, we can construct a data path shown in Fig. 7. In the data path, a plaintext is XORed with whitening key before stored in the register R_{ij} $(0 \leq i \leq 3,\ 0 \leq j \leq 1)$. After that, the intermediate results D_i^L, D_i^R $(0 \leq i \leq 3)$ are calculated and stored in the registers R_{ij} according to the round permutation RP. In the process of the final round, the data path executes not the key whitening operations for the output but the round permutation. Finally, the contents of the registers R_{ij} is permuted through the inversion of round permutation RP^{-1}. Then, the key

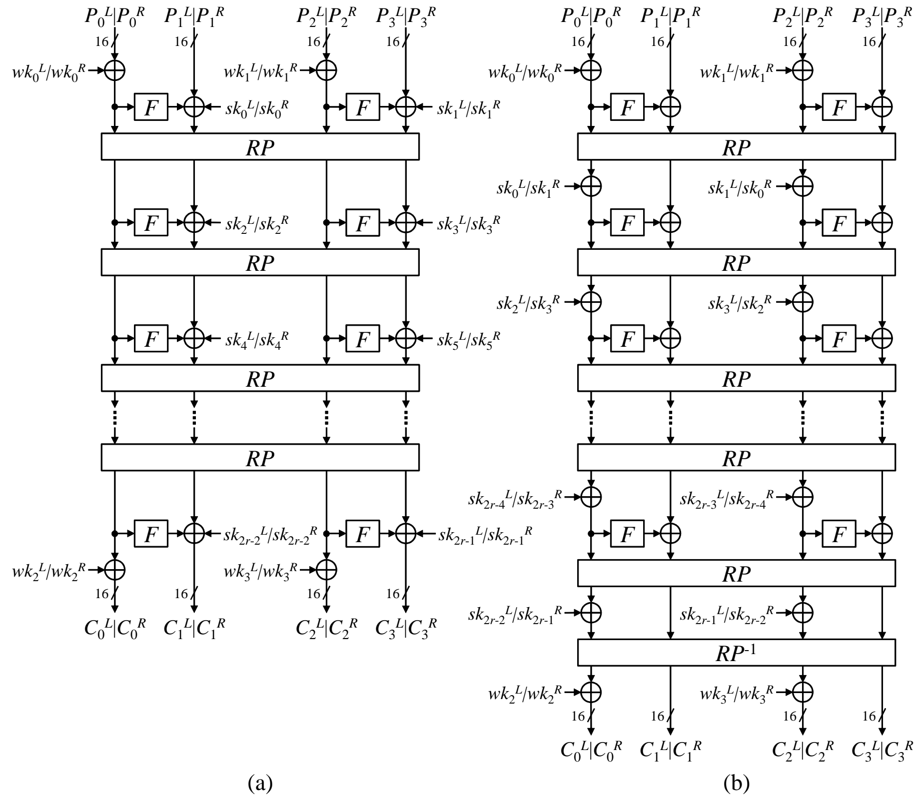


Fig. 6. (a) Encryption process, (b) Optimized encryption process. XOR operations with the part of round keys are moved by an equivalent transformation.

whitening operations at the final round are executed by XORing the whitening keys with L_1^L, L_1^R, L_3^L and L_3^R

In contrast to the straightforward implementation Fig. 5, we can reduce the area requirements of eight 8-bit XORs and eight 8-bit ANDs, although four 8-bit 3-to-1 MUXes is added instead of four 8-bit 2-to-1 MUXes. If a library includes a 3-to-1 MUX which has low area requirement, the additional cost for key whitening operation becomes smaller. In fact, we can reduce 176 GE from the straightforward implementation in our library.

4 Serialized Implementation

In this section, we propose compact hardware implementation techniques of the serialized hardware architectures. Especially, we focus on how to implement an MDS matrix in the structure of a generalized Feistel network.

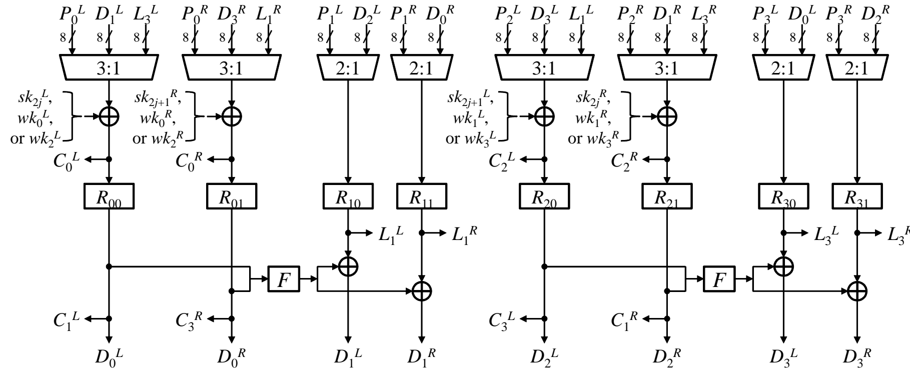
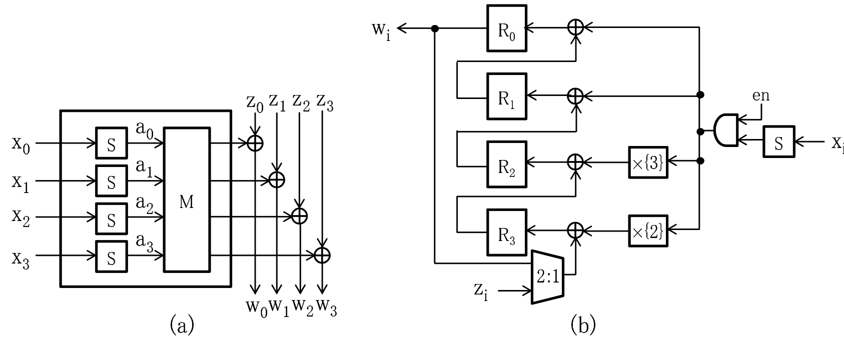


Fig. 7. Our data path of round-based architecture

4.1 Serialized Architecture of SP-type F-functions

Several proposals about implementation techniques for MDS matrices in an SPN structure have been reported in [11, 10, 20, 12, 9]. The techniques used in [12, 9] implement only a fraction of an MDS matrix with additional three 8-bit registers. On the other hand, in [20], a whole MDS matrix is implemented in a serialized architecture to avoid the additional registers and complexity of control logic. Recently, another approach was proposed in [11, 10]. The authors construct new MDS matrices suitable for serialized implementation. This MDS matrices have a feature that enables low-area implementation of the MDS matrices without additional registers in a serialized architecture.

MDS matrices are also adopted in a generalized Feistel network. In [1], a compact matrix multiplier was proposed by using a feature of the generalized Feistel network which has an SP-type F-function as shown in Fig. 8 (a). We assume that the matrix in the SP-type F-function is the same as the matrix described in Sect. 2. The technique proposed in [1] allows us to construct the matrix multiplier without additional registers for the SP-type F-function in a generalized Feistel network as shown in Fig. 8 (b). Fig. 8 (c) presents the contents of the registers R_i $(0 \leq i \leq 3)$ at the l-th cycle $(1 \leq l \leq 8)$. For the duration of first four cycles, the enable signal en is set to low and the data z_i $(0 \leq i \leq 3)$ are stored in the four registers R_j. At 5-th cycle, the output a_0 of S-box is calculated, fed to the multiplier and multiplied by $\{2\}$, $\{3\}$, $\{1\}$, and $\{1\}$. The data stored in the four registers R_j is updated by XORing with the products. As the same multiplier coefficients are used for each column in the matrix, we can perform the matrix operation at the following cycle by adding x_i $(1 \leq i \leq 3)$ and cyclically shifting the intermediate results included in the four registers R_j.



l	1	2	3	4	5	6	7	8
R_0				z_0	$z_1 \oplus a_0$	$z_2 \oplus a_0 \oplus a_1$	$z_3 \oplus 3 \cdot a_0 \oplus a_1 \oplus a_2$	$z_0 \oplus 2 \cdot a_0 \oplus 3 \cdot a_1 \oplus a_2 \oplus a_3$
R_1			z_0	z_1	$z_2 \oplus a_0$	$z_3 \oplus 3 \cdot a_0 \oplus a_1$	$z_0 \oplus 2 \cdot a_0 \oplus 3 \cdot a_1 \oplus a_2$	$z_1 \oplus a_0 \oplus 2 \cdot a_1 \oplus 3 \cdot a_2 \oplus a_3$
R_2		z_0	z_1	z_2	$z_3 \oplus 3 \cdot a_0$	$z_0 \oplus 2 \cdot a_0 \oplus 3 \cdot a_1$	$z_1 \oplus a_0 \oplus 2 \cdot a_1 \oplus 3 \cdot a_2$	$z_2 \oplus a_0 \oplus a_1 \oplus 2 \cdot a_2 \oplus 3 \cdot a_3$
R_3	z_0	z_1	z_2	z_3	$z_0 \oplus 2 \cdot a_0$	$z_1 \oplus a_0 \oplus 2 \cdot a_1$	$z_2 \oplus a_0 \oplus a_1 \oplus 2 \cdot a_2$	$z_3 \oplus 3 \cdot a_0 \oplus a_1 \oplus a_2 \oplus 2 \cdot a_3$

(c)

Fig. 8. Matrix multiplier proposed in [1]: (a) SP-type F-function, (b) Data path, (c) Contents of registers R_j $(0 \leq j \leq 3)$ at the l-th cycle

The above technique uses the equivalent transformation described below,

$$\begin{aligned} w_0 &= 2 \cdot \mathrm{S}(x_0) + 3 \cdot \mathrm{S}(x_1) + \mathrm{S}(x_2) + \mathrm{S}(x_3) + z_0 \\ &= \underbrace{[2 \cdot \mathrm{S}(x_0) + z_0]}_{\text{5th cycle}} + \underbrace{[3 \cdot \mathrm{S}(x_1)]}_{\text{6th cycle}} + \underbrace{[\mathrm{S}(x_2)]}_{\text{7th cycle}} + \underbrace{[\mathrm{S}(x_3)]}_{\text{8th cycle}} . \end{aligned}$$

Since calculations relevant to z_i $(0 \leq i \leq 3)$ are processed prior to calculations relevant to x_i $(1 \leq i \leq 3)$, w_i can be calculated using the register where z_i for the newly processing F-function is stored. Thus, we can design a compact matrix multiplier without additional registers for the generalized Feistel network which has an SP-type F-function.

4.2 Introducing Inverse of S-box for SPS-type F-functions

Next, we discuss the implementation of an MDS matrix for the generalized Feistel network which has an SPS-type F-function as shown in Fig. 9. Since the results of the matrix operation are input to S-boxes, calculations relevant to z_i cannot be processed prior to calculations relevant to x_i $(1 \leq i \leq 3)$ as well as in the case of an SP-type F-function. Thus, we cannot apply the technique in [1] to design a matrix multiplier without addtional registers. In addition to registers to store x_i and z_i, the matrix multiplier requires four additional registers for intermediate values of the matrix calculation. We present a trivial construction of the matrix multiplier with additional four registers in Fig. 10 (a). Fig. 10 (b) presents the contents of the registers R_i $(0 \leq i \leq 3)$ at the l-th cycle $(1 \leq l \leq 8)$. During the first four cycles, the outputs a_i $(0 \leq i \leq 3)$ of S-boxes are stored in the register

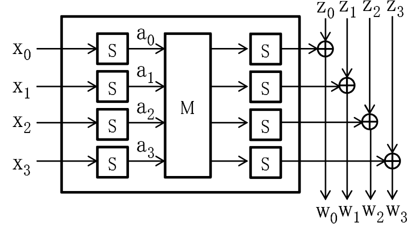
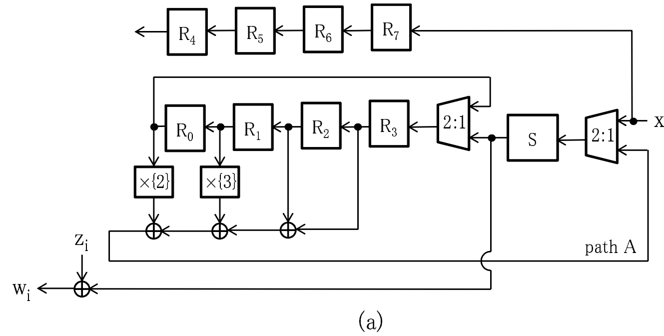


Fig. 9. SPS-type F-function



(a)

l	1	2	3	4	5	6	7	8
R_0				a_0	a_1	a_2	a_3	a_0
R_1			a_0	a_1	a_2	a_3	a_0	a_1
R_2		a_0	a_1	a_2	a_3	a_0	a_1	a_2
R_3	a_0	a_1	a_2	a_3	a_0	a_1	a_2	a_3

(b)

Fig. 10. Trivial construction for SPS-type F-function: (a) Data path, (b) Contents of registers R_j $(0 \leq j \leq 3)$ at the l-th cycle

R_j $(0 \leq j \leq 3)$. At the same time, the inputs x_i $(0 \leq i \leq 3)$ of S-boxes are also stored in other registers R_j $(4 \leq j \leq 7)$ since this value is necessary in the next round. At 5-th cycle, a_0, a_1, a_2 and a_3 is multiplied by $\{2\}, \{3\}, \{1\}$ and $\{1\}$, respectively, and the sum of the products is calculated. Then, the sum are input to the S-box through path A described in Fig. 10 and w_0 is obtained by XORing the output of S-box with z_0. As we assume that the matrix is circulant, w_i $(1 \leq i \leq 3)$ is processed by rotating the data stored in the four registers R_j $(0 \leq j \leq 3)$.

In the above trivial construction, we use the four additional registers for storing intermediate values of the F-function. However, flip-flops require relatively high area requirements. We propose new implementation technique of an MDS matrix without additional registers. Our technique introduces the inverse function of S-box which is usually unnecessary to process both an encryption and decryption in a generalized Feistel network. We describe a new data path for

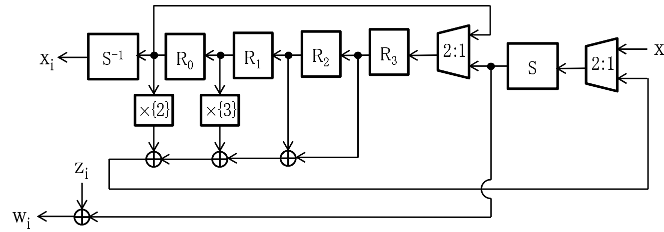


Fig. 11. Data path for SPS-type F-function without additional registers

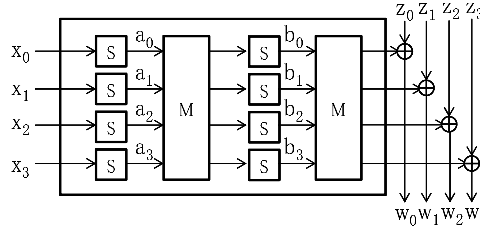


Fig. 12. SPSP-type F-function

SPS-type F-function in Fig. 11. Although we obtain w_i $(0 \leq i \leq 3)$ in the same way as the process of trivial construction described in Fig. 10, we do not store x_i $(0 \leq i \leq 3)$ in other registers during the calculation of w_i. After the calculation of w_i, we recover x_i through S^{-1} which is the inversion of S-box. Note that the required number of cycles for calculation of F-function is the same as the cycle in the trivial construction, since we can set a next input of F-function in order during recovering x_i.

In contrast to the trivial construction, our implementation requires the circuit of S^{-1} instead of additional registers. If the area requirements of additional registers are larger than that of S^{-1}, our technique contributes to the compact implementation. In fact, the area requirement of S^{-1} in *Piccolo* is very small and it costs 12 GE in our library. On the other hand, in the trivial construction, it costs 72 GE in our library to use 16-bit additional registers. Therefore, we can save 60 GE in the implementation of *Piccolo*.

Combined with the technique in [1], our technique can be applied to implementation of an SPSP-type F-function described in Fig. 12 without additional registers. In [7], it is reported that the SPSP-type F-function has a good property from the aspect of a proportion of active S-boxes. However, additional registers is necessary to construct a serialized architecture if we use only one of technique in [1] and our proposal above technique. We combine the two techniques to implement the SPSP-type F-function without additional registers. We provide the construction in Fig. 13. Fig. 14 presents the contents of the registers R_j $(0 \leq j \leq 3)$ at the l-th cycle $(1 \leq l \leq 8)$. During the first four cycles, a_i and z_i $(0 \leq i \leq 3)$ are stored in the register R_i $(0 \leq i \leq 7)$ described in Fig. 14. At

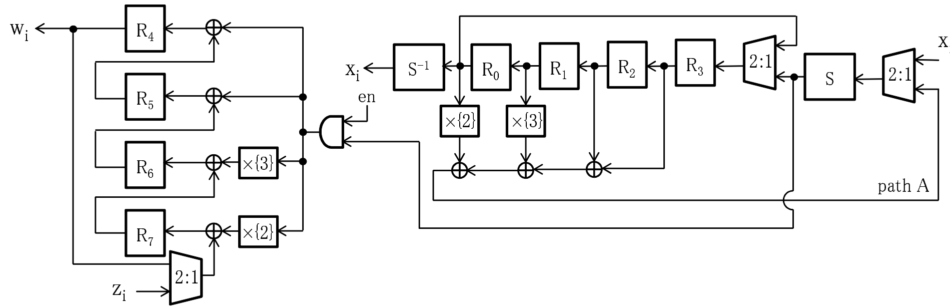


Fig. 13. Data path for SPSP-type F-function without additional registers

l	1	2	3	4	5	6	7	8
R_0				a_0	a_1	a_2	a_3	a_0
R_1			a_0	a_1	a_2	a_3	a_0	a_1
R_2		a_0	a_1	a_2	a_3	a_0	a_1	a_2
R_3	a_0	a_1	a_2	a_3	a_0	a_1	a_2	a_3
R_4				z_0	$z_1 \oplus b_0$	$z_2 \oplus b_0 \oplus b_1$	$z_3 \oplus 3 \cdot b_0 \oplus b_1 \oplus b_2$	$z_0 \oplus 2 \cdot b_0 \oplus 3 \cdot b_1 \oplus b_2 \oplus b_3$
R_5			z_0	z_1	$z_2 \oplus b_0$	$z_3 \oplus 3 \cdot b_0 \oplus b_1$	$z_0 \oplus 2 \cdot b_0 \oplus 3 \cdot b_1 \oplus b_2$	$z_1 \oplus b_0 \oplus 2 \cdot b_1 \oplus 3 \cdot b_2 \oplus b_3$
R_6		z_0	z_1	z_2	$z_3 \oplus 3 \cdot b_0$	$z_0 \oplus 2 \cdot b_0 \oplus 3 \cdot b_1$	$z_1 \oplus b_0 \oplus 2 \cdot b_1 \oplus 3 \cdot b_2$	$z_2 \oplus b_0 \oplus b_1 \oplus 2 \cdot b_2 \oplus 3 \cdot b_3$
R_7	z_0	z_1	z_2	z_3	$z_0 \oplus 2 \cdot b_0$	$z_1 \oplus b_0 \oplus 2 \cdot b_1$	$z_2 \oplus b_0 \oplus b_1 \oplus 2 \cdot b_2$	$z_3 \oplus 3 \cdot b_0 \oplus b_1 \oplus b_2 \oplus 2 \cdot b_3$

Fig. 14. Contents of registers R_j $(0 \leq j < 4)$ at the l-th cycle

5-th cycle, the output b_0 of second S-box is calculated through path A described in Fig. 13 and the data stored in the four registers R_i $(4 \leq i \leq 7)$ is updated by XORing with b_0. At the following cycle, since b_i $(1 \leq i \leq 3)$ is obtained by rotating the data stored in the four registers R_j $(0 \leq j \leq 3)$, w_i is evaluated as the contents of the four register R_i $(4 \leq i \leq 7)$ by the same procedure as Fig. 8.

4.3 Serialized Architecture of *Piccolo*

In this section, we propose two serialized architectures for *Piccolo*.

First, we present the data path of a serialized architecture of *Piccolo* supporting both encryption and decryption in Fig. 15, where the width of data path is 4 bit except those written in the figure. In Appendix, we show the detailed data flow of the data registers R_i $(0 \leq i \leq 15)$ described in Fig. 15. The data flow of both encryption and decryption is similar. We roughly explain the data flow as follows. We set most significant four nibble of a plaintext into the four registers R_i $(0 \leq i \leq 3)$ through an S-box during the input of a plaintext. At the first four cycles for calculating round function, the output of F-function are computed by using technique described in Sect. 4.2 and are XORed with a content of the register R_4 and chunks of constant value CON and secret key. Then, the calculated values are stored in the four register R_i $(12 \leq i \leq 15)$. In the next four cycle, the input of F-function are recovered through S^{-1} and are stored in the four register R_i $(12 \leq i \leq 15)$. At the same time, the input for a calculation of the other F-function in the round function is set the four registers R_i $(0 \leq i \leq 3)$

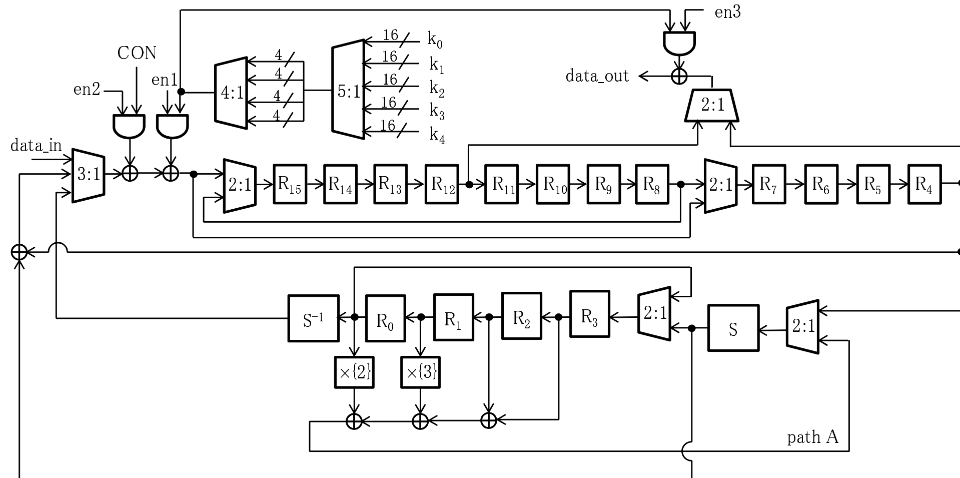


Fig. 15. Data path of serialized architecture supporting both encryption and decryption

through an S-box. Although there are some differences about the data flow, the other output of the round function are calculated in the same manner as the above flow in the next eight cycles. After the calculation of the final round, a ciphertext is output nibble by nibble from data_out described in Fig. 15 in 16 cycles. In the ciphertext output phase, we need 4-bit 2-to-1 MUX and XOR due to the alignment of a content of the register R_i $(0 \leq i \leq 15)$ and key whitening operation, respectively.

Next, we propose a implementation technique to construct the data path supporting only encryption. We move the XOR operations with the related a chunk of secret key into the lines before round permutation by an equivalent transformation as shown in Fig. 16. In Fig. 16 (b), each XOR operation of constant values and a chunk of secret key are set in the different lines except the first round key. By doing so, we can construct a serialized architecture which one of the XOR operations of constant values and a chunk of secret key can be processed every cycle. Since it is unnecessary to calculate the two XOR operations in one cycle, the two XOR operations can be shared. Furthermore, while key whitening operations in the final round is done at the ciphertext output phase in Fig. 15, key whitening operations can be calculated before the ciphertext output phase. This indicate a possibility of reduction of the XOR operation in front of data_out in Fig. 15. Moreover, we can change the data flow in the final round computation so that it is unnecessary to align data after the final round. We also reduce 2-to-1 MUX from Fig. 15. We describe a data path in Fig. 17 constructed from the equivalent transformation. Since the data flow of Fig. 17 is similar to that of Fig. 15, we roughly explain the data flow as follows. In addition to key whitening operations, we calculate XOR operation of the part of first round key

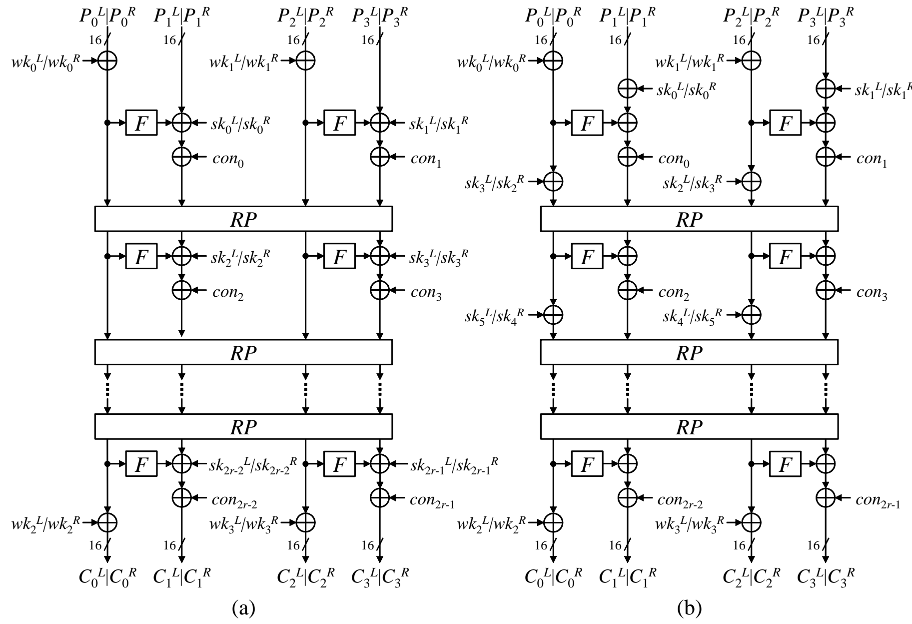


Fig. 16. (a) Encryption process, (b) Optimized encryption process. XOR operations with the part of round keys are moved by an equivalent transformation.

related to secret key, during the input of a plaintext. Then, the round function is calculated in the same manner as the data flow of Fig. 15, except calculating XOR operations alternately with a chunk of constant value CON and secret key every four cycles. In the final round, instead of round permutation RP, we execute the round function by 16-bit cyclic shift so that a ciphertext is output nibble by nibble from data_out described in Fig. 17 in 16 cycles.

5 Implementation Results

5.1 Evaluation Environment

We design and evaluate the hardware implementation presented in Sect. 3 and 4. The environment of our hardware design and evaluation is as follows:

Language	Verilog-HDL
Design library	0.13 μm CMOS ASIC library
Simulator	VCS version 2006.06
Logic synthesis	Design Compiler version 2007.03-SP3

In order to compare the area requirements independently of the technology used, it is common to state the area as Gate Equivalents (GE). One GE is equivalent to the area of a 2-way NAND with the lowest drive strength. For synthesis, we

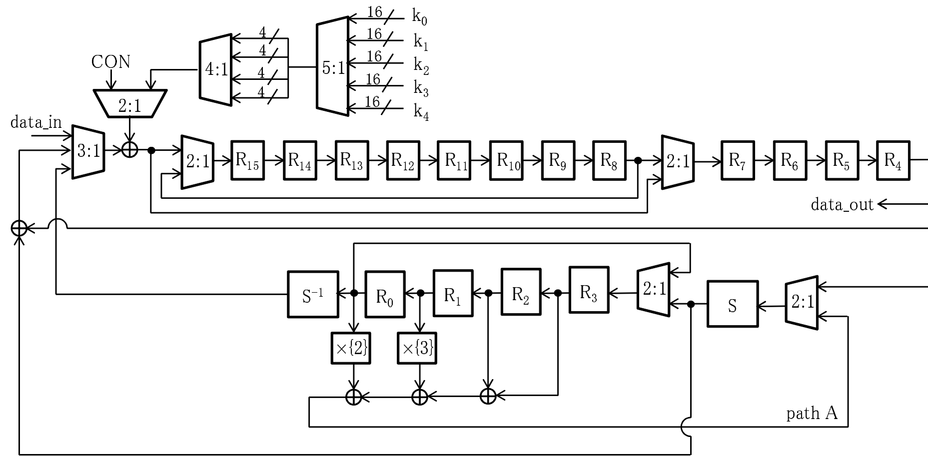


Fig. 17. Data path of serialized architecture supporting only encryption

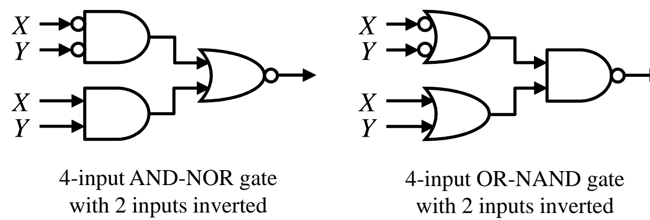


Fig. 18. 4-input AND-NOR and 4-input OR-NAND gate with 2 inputs inverted, which correspond to XOR and XNOR gate

use a clock frequency of 100 KHz, which is widely used operating frequency for RFID applications to lower power consumption.

In a recent trend, the implementation of lightweight block ciphers uses a scan flip-flop instead of a combination of a D flip-flop and 2-to-1 MUX [25, 8, 23, 20] to reduce the gate requirement. In our evaluation, a D flip-flop and a 2-to-1 MUX cost 4.5 and 2.0 GE, respectively, while a scan flip-flop costs 6.25 GE. Thus, we can save 0.25 GE per bit of storage. Moreover, the library we used has the 4-input AND-NOR and 4-input OR-NAND gates with 2 inputs inverted described in Fig. 18. The outputs of these cells are corresponding to those of XOR or XNOR gates when the inputs X, Y are set as shown in Fig. 18. Since these cells cost 2 GE instead of 2.25 GE required for XOR or XNOR cell, we can save 0.25 GE per XOR or XNOR gate. We also use clock gating logics. Clock gating is a power-saving technique used in synchronous circuits. For hardware implementations of block ciphers, it was firstly introduced in [25] as a technique to reduce gate counts and power, and have been applied to KATAN family [8],

AES [20], and CLEFIA [1]. Clock gating works by taking the enable conditions attached to registers. It can remove feedback MUXes to hold their present state and replace them with clock gating logic. In the case that several bits of registers take the same enable conditions, their gate counts will be saved by applying clock gating.

5.2 Key Input Setting

Depending on whether an application needs to update a key, implementation of key scheduling part may be changed. If the application is allowed to update a key, implementation of the key scheduling part requires the key-length flip-flops. On the other hand, if the application does not need to update a key, the key scheduling part may be designed by using a hard-wired key. The hard-wired key can reduce the area requirements in some structures of key scheduling part. Especially, permutation-based key scheduling part such as that of GOST allows to be designed without flip-flops by using the hard-wired key and multiplexers. In [23], GOST with a key-length of 256 bits requires less than 100 GE for the key scheduling part without 256-bit flip-flops. Since the difference of application changes the area requirements of key scheduling part, we compare the area requirements under the same key input conditions.

As noted above, there are two key input setting: flexible-key and fixed-key setting. Next, we discuss a good feature of permutation-based key scheduling part in the flexible-key setting. In practice, before an execution of data encryption, a key usually needs to be loaded into registers from a memory through a bus. If the values of the registers change during the execution, the key needs to be restored from the memory or reconstructed from the updated registers before next execution. Since a bus width is generally smaller than key length, several cycles may be necessary for reloading the key. When reconstructing the key from the registers, additional circuit and cycles for it are required. Depending on a situation, throughput is important in addition to area requirements. In case of the permutation-based key scheduling part, it is not necessary to change values of the registers during the execution, since multiplexers select certain chunk from a key. Therefore, an encryption algorithm is continuously executed without restoring or reconstructing the key. From the view point of the above discussion, we consider that the permutation-based key scheduling has desirable property in the flexible-key setting.

5.3 Comparative Results

Table 1 and Table 2 show the comparative results regarding the hardware efficiency for lightweight block ciphers in flexible and fixed-key setting, respectively. These tables show *Piccolo* achieves very small area requirements in both the flexible and fixed-key settings. The energy efficiency of *Piccolo* in both settings is very high in terms of *energy per bit* that is a metric for energy consumption proposed by [27]. Moreover, *Piccolo* requires only about 60 additional GE to support decryption function in the serialized architecture. As a result, we consider that

Table 1. Comparative results in hardware implementations in flexible-key setting

Algorithm	block size [bit]	key size [bit]	type	serialized arch. area [GE]	serialized arch. cycles/ block	round-based arch. area [GE]	round-based arch. cycles/ block	round-based arch. energy/*2 bit
DESXL [17]	64	184	Feistel	2,168	144	-	-	-
†HIGHT [13]*1	64	128	GFN	-	-	3,048	34	1,620
mCrypton-96 [18]	64	96	SPN	-	-	2,681	13	545
mCrypton-128 [18]	64	128	SPN	-	-	2,949	13	600
PRESENT-80 [5, 25]	64	80	SPN	1,000	547	1,570	32	785
KATAN64 [8]	64	80	stream	1,054	254	-	-	-
LED-64 [11]	64	64	SPN	966	1,248	2,695	32	1,347
LED-80 [11]	64	80	SPN	1,040	1,872	2,780	48	2,085
LED-128 [11]	64	128	SPN	1,265	1,872	3,036	48	2,277
Piccolo-80	64	80	GFN	**1,048**	432	**1,499**	27	**633**
Piccolo-128	64	128	GFN	**1,338**	528	**1,776**	33	**916**
Piccolo-80*1	64	80	GFN	**1,109**	432	**1,638**	27	**692**
Piccolo-128*1	64	128	GFN	**1,397**	528	**1,942**	33	**1,002**
AES-128 [20],[26]*1	128	128	SPN	2,400	226	12,454*3	11	1,071
CLEFIA-128 [1],[28]*1	128	128	GFN	2,678	176	5,979	18	841

*1: Including decryption function. The others support encryption-mode only.
*2: energy / bit = (area [GE] × required cycles for one block process [cycle]) / block size [bit].
*3: This implementation is not intended to be high efficiency but high throughput.

Piccolo is the competitive general purpose ultra-lightweight block cipher suitable for extremely constrained environments under any situation.

6 Conclusion

In this paper, we have presented the detailed descriptions of hardware implementations of *Piccolo*. We proposed a new technique to implement a generalized Feistel structure with SPS-type F-functions in serialized architectures. Moreover, We designed round-based architecture to reduce additional area requirements for key whitening operation. As a result, it was shown that *Piccolo* is very competitive to other lightweight block ciphers in both flexible and fixed-key setting

Future work will include the application of side-channel countermeasures such as threshold implementations [21, 22] and software implementations of *Piccolo*.

References

1. Toru Akishita and Harunaga Hiwatari. Very compact hardware implementations of block cipher CLEFIA. In *Selected Areas in Cryptography*, 2011. Available at `http://sac2011.ryerson.ca/SAC2011/AH.pdf`.
2. Eli Biham, Orr Dunkelman, and Nathan Keller. Related-key boomerang and rectangle attacks. In Ronald Cramer, editor, *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 507–525. Springer, 2005.
3. Eli Biham, Orr Dunkelman, and Nathan Keller. A unified approach to related-key attacks. In Kaisa Nyberg, editor, *FSE*, volume 5086 of *Lecture Notes in Computer Science*, pages 73–96. Springer, 2008.
4. Alex Biryukov, editor. *Fast Software Encryption, 14th International Workshop, FSE 2007, Luxembourg, Luxembourg, March 26-28, 2007, Revised Selected Papers*, volume 4593 of *Lecture Notes in Computer Science*. Springer, 2007.

Table 2. Comparative results in hardware implementations in fixed-key setting

Algorithm	block size [bit]	key size [bit]	type	serialized arch. area [GE]	serialized arch. cycles/ block	round-based arch. area [GE]	round-based arch. cycles/ block	round-based arch. energy/ bit
‡KTANTAN64 [8]	64	80	stream	688	254	-	-	-
‡GOST-PS [23]	64	256	Feistel	651	264	1,017	32	509
‡GOST-FB [23]	64	256	Feistel	800	264	1,000	32	500
LED-64 [11]	64	64	SPN	688	1,248	2,354	32	1,177
LED-80 [11]	64	80	SPN	690	1,872	2,354	48	1,765
LED-128 [11]	64	128	SPN	700	1,872	2,354	48	1,765
Piccolo-80	64	80	GFN	**616***2	432	**1,051***3	27	**444**
Piccolo-128	64	128	GFN	**654***2	528	**1,083***3	33	**559**
Piccolo-80*1	64	80	GFN	**675***2	432	**1,199***3	27	**506**
Piccolo-128*1	64	128	GFN	**721***2	528	**1,249***3	33	**645**
PRINTcipher-48 [15]	48	80	SPN	402	768	503	48	503
PRINTcipher-96 [15]	96	160	SPN	726	3,072	967	96	967

†: Theoretically broken under related-key setting [16].
‡: Theoretically broken under single-key setting [6, 14].
∗1: Including decryption function. The others support encryption-mode only.
∗2: These values are based on average area requirement of ten key randomly chosen.
∗3: We estimate these values stochastically.

5. Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *CHES*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007.
6. Andrey Bogdanov and Christian Rechberger. A 3-subset meet-in-the-middle attack: Cryptanalysis of the lightweight block cipher ktantan. In Alex Biryukov, Guang Gong, and Douglas R. Stinson, editors, *Selected Areas in Cryptography*, volume 6544 of *Lecture Notes in Computer Science*, pages 229–240. Springer, 2010.
7. Andrey Bogdanov and Kyoji Shibutani. Double SP-functions: Enhanced generalized feistel networks - extended abstract. In Udaya Parampalli and Philip Hawkes, editors, *ACISP*, volume 6812 of *Lecture Notes in Computer Science*, pages 106–119. Springer, 2011.
8. Christophe De Cannière, Orr Dunkelman, and Miroslav Knezevic. KATAN and KTANTAN - a family of small and efficient hardware-oriented block ciphers. In Christophe Clavier and Kris Gaj, editors, *CHES*, volume 5747 of *Lecture Notes in Computer Science*, pages 272–288. Springer, 2009.
9. Martin Feldhofer, Johannes Wolkerstorfer, and Vincent Rijmen. AES implementation on a grain of sand. *IEE proceedings / information security*, 152:13 – 20, 2005.
10. Jian Guo, Thomas Peyrin, and Axel Poschmann. The PHOTON family of lightweight hash functions. In Phillip Rogaway, editor, *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pages 222–239. Springer, 2011.
11. Jian Guo, Thomas Peyrin, Axel Poschmann, and Matthew J. B. Robshaw. The LED block cipher. In Preneel and Takagi [24], pages 326–341.
12. Panu Hämäläinen, Timo Alho, Marko Hännikäinen, and Timo D. Hämäläinen. Design and implementation of low-area and low-power AES encryption hardware core. In *DSD*, pages 577–583. IEEE Computer Society, 2006.
13. Deukjo Hong, Jaechul Sung, Seokhie Hong, Jongin Lim, Sangjin Lee, Bonseok Koo, Changhoon Lee, Donghoon Chang, Jaesang Lee, Kitae Jeong, Hyun Kim, Jongsung Kim, and Seongtaek Chee. HIGHT: A new block cipher suitable for

low-resource device. In Louis Goubin and Mitsuru Matsui, editors, *CHES*, volume 4249 of *Lecture Notes in Computer Science*, pages 46–59. Springer, 2006.
14. Takanori Isobe. A single-key attack on the full gost block cipher. In Antoine Joux, editor, *FSE*, volume 6733 of *Lecture Notes in Computer Science*, pages 290–305. Springer, 2011.
15. Lars R. Knudsen, Gregor Leander, Axel Poschmann, and Matthew J. B. Robshaw. PRINTcipher: A block cipher for IC-printing. In Mangard and Standaert [19], pages 16–32.
16. Bonwook Koo, Deukjo Hong, and Daesung Kwon. Related-key attack on the full hight. In Kyung Hyune Rhee and DaeHun Nyang, editors, *ICISC*, volume 6829 of *Lecture Notes in Computer Science*, pages 49–67. Springer, 2010.
17. Gregor Leander, Christof Paar, Axel Poschmann, and Kai Schramm. New lightweight DES variants. In Biryukov [4], pages 196–210.
18. Chae Hoon Lim and Tymur Korkishko. mCrypton - a lightweight block cipher for security of low-cost RFID tags and sensors. In JooSeok Song, Taekyoung Kwon, and Moti Yung, editors, *WISA*, volume 3786 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2005.
19. Stefan Mangard and François-Xavier Standaert, editors. *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*. Springer, 2010.
20. Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. Pushing the limits: A very compact and a threshold implementation of AES. In Kenneth G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 69–88. Springer, 2011.
21. Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *ICICS*, volume 4307 of *Lecture Notes in Computer Science*, pages 529–545. Springer, 2006.
22. Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure hardware implementation of non-linear functions in the presence of glitches. In Pil Joong Lee and Jung Hee Cheon, editors, *ICISC*, volume 5461 of *Lecture Notes in Computer Science*, pages 218–234. Springer, 2008.
23. Axel Poschmann, San Ling, and Huaxiong Wang. 256 bit standardized crypto for 650 ge - GOST revisited. In Mangard and Standaert [19], pages 219–233.
24. Bart Preneel and Tsuyoshi Takagi, editors. *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*. Springer, 2011.
25. Carsten Rolfes, Axel Poschmann, Gregor Leander, and Christof Paar. Ultra-lightweight implementations for smart devices - security for 1000 gate equivalents. In Gilles Grimaud and François-Xavier Standaert, editors, *CARDIS*, volume 5189 of *Lecture Notes in Computer Science*, pages 89–103. Springer, 2008.
26. Akashi Satoh and Sumio Morioka. Hardware-focused performance comparison for the standard block ciphers aes, camellia, and triple-des. In Colin Boyd and Wenbo Mao, editors, *ISC*, volume 2851 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2003.
27. Kyoji Shibutani, Takanori Isobe, Harunaga Hiwatari, Atsushi Mitsuda, Toru Akishita, and Taizo Shirai. Piccolo: An ultra-lightweight blockcipher. In Preneel and Takagi [24], pages 342–357.

28. Taizo Shirai, Kyoji Shibutani, Toru Akishita, Shiho Moriai, and Tetsu Iwata. The 128-bit blockcipher CLEFIA (extended abstract). In Biryukov [4], pages 181–195.

Appendix

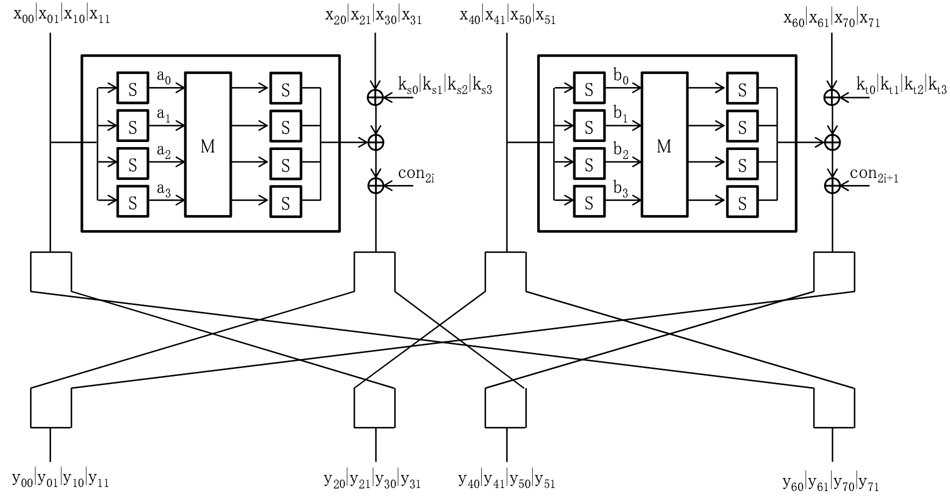


Fig. 19. A round of encryption process

Table 3. Contents of registers R_{ij} $(0 \leq i, j < 4)$ at the l-th cycle during encryption mode

l	0	1	2	3	4	5	6	7	8
R_0	$S(x_0)\ (= a_0)$	a_1	a_2	a_3	a_0	a_1	a_2	a_3	$S(x_{40})\ (= b_0)$
R_1	$S(x_1)\ (= a_1)$	a_2	a_3	a_0	a_1	a_2	a_3	$S(x_{40})$	$S(x_{41})\ (= b_1)$
R_2	$S(x_2)\ (= a_2)$	a_3	a_0	a_1	a_2	a_3	$S(x_{40})$	$S(x_{41})$	$S(x_{50})\ (= b_2)$
R_3	$S(x_3)\ (= a_3)$	a_0	a_1	a_2	a_3	$S(x_{40})$	$S(x_{41})$	$S(x_{50})$	$S(x_{51})\ (= b_3)$
R_4	x_{20}	x_{21}	x_{30}	x_{31}	x_{40}	x_{41}	x_{50}	x_{51}	x_{60}
R_5	x_{21}	x_{30}	x_{31}	x_{40}	x_{41}	x_{50}	x_{51}	x_{60}	x_{61}
R_6	x_{30}	x_{31}	x_{40}	x_{41}	x_{50}	x_{51}	x_{60}	x_{61}	x_{70}
R_7	x_{31}	x_{40}	x_{41}	x_{50}	x_{51}	x_{60}	x_{61}	x_{70}	x_{71}
R_8	x_{40}	x_{41}	x_{50}	x_{51}	x_{60}	x_{61}	x_{70}	x_{71}	y_{00}
R_9	x_{41}	x_{50}	x_{51}	x_{60}	x_{61}	x_{70}	x_{71}	y_{00}	y_{01}
R_{10}	x_{50}	x_{51}	x_{60}	x_{61}	x_{70}	x_{71}	y_{00}	y_{01}	y_{50}
R_{11}	x_{51}	x_{60}	x_{61}	x_{70}	x_{71}	y_{00}	y_{01}	y_{50}	y_{51}
R_{12}	x_{60}	x_{61}	x_{70}	x_{71}	y_{00}	y_{01}	y_{50}	y_{51}	$x_{00}\ (= y_{60})$
R_{13}	x_{61}	x_{70}	x_{71}	y_{00}	y_{01}	y_{50}	y_{51}	x_{00}	$x_{01}\ (= y_{61})$
R_{14}	x_{70}	x_{71}	y_{00}	y_{01}	y_{50}	y_{51}	x_{00}	x_{01}	$x_{10}\ (= y_{30})$
R_{15}	x_{71}	y_{00}	y_{01}	y_{50}	y_{51}	x_{00}	x_{01}	x_{10}	$x_{11}\ (= y_{31})$

l	8	9	10	11	12	13	14	15	16
R_0	b_0	b_1	b_2	b_3	b_0	b_1	b_2	b_3	$S(y_{00})\ (= a'_0)$
R_1	b_1	b_2	b_3	b_0	b_1	b_2	b_3	$S(y_{00})$	$S(y_{01})\ (= a'_1)$
R_2	b_2	b_3	b_0	b_1	b_2	b_3	$S(y_{00})$	$S(y_{01})$	$S(y_{10})\ (= a'_2)$
R_3	b_3	b_0	b_1	b_2	b_3	$S(y_{00})$	$S(y_{01})$	$S(y_{10})$	$S(y_{11})\ (= a'_3)$
R_4	x_{60}	x_{61}	x_{70}	x_{71}	y_{00}	y_{01}	y_{10}	y_{11}	$x_{40}\ (= y_{20})$
R_5	x_{61}	x_{70}	x_{71}	y_{00}	y_{01}	y_{10}	y_{11}	x_{40}	$x_{41}\ (= y_{21})$
R_6	x_{70}	x_{71}	y_{00}	y_{01}	y_{10}	y_{11}	x_{40}	x_{41}	y_{30}
R_7	x_{71}	y_{00}	y_{01}	y_{10}	y_{11}	x_{40}	x_{41}	y_{30}	y_{31}
R_8	y_{00}	y_{01}	y_{50}	y_{51}	y_{60}	y_{61}	y_{30}	y_{31}	y_{40}
R_9	y_{01}	y_{50}	y_{51}	y_{60}	y_{61}	y_{30}	y_{31}	y_{40}	y_{41}
R_{10}	y_{50}	y_{51}	y_{60}	y_{61}	y_{30}	y_{31}	y_{40}	y_{41}	y_{50}
R_{11}	y_{51}	y_{60}	y_{61}	y_{30}	y_{31}	y_{40}	y_{41}	y_{50}	y_{51}
R_{12}	y_{60}	y_{61}	y_{30}	y_{31}	y_{40}	y_{41}	y_{50}	y_{51}	y_{60}
R_{13}	y_{61}	y_{30}	y_{31}	y_{40}	y_{41}	y_{50}	y_{51}	y_{60}	y_{61}
R_{14}	y_{30}	y_{31}	y_{40}	y_{41}	y_{50}	y_{51}	y_{60}	y_{61}	$x_{50}\ (= y_{70})$
R_{15}	y_{31}	y_{40}	y_{41}	y_{50}	y_{51}	y_{60}	y_{61}	x_{50}	$x_{51}\ (= y_{71})$

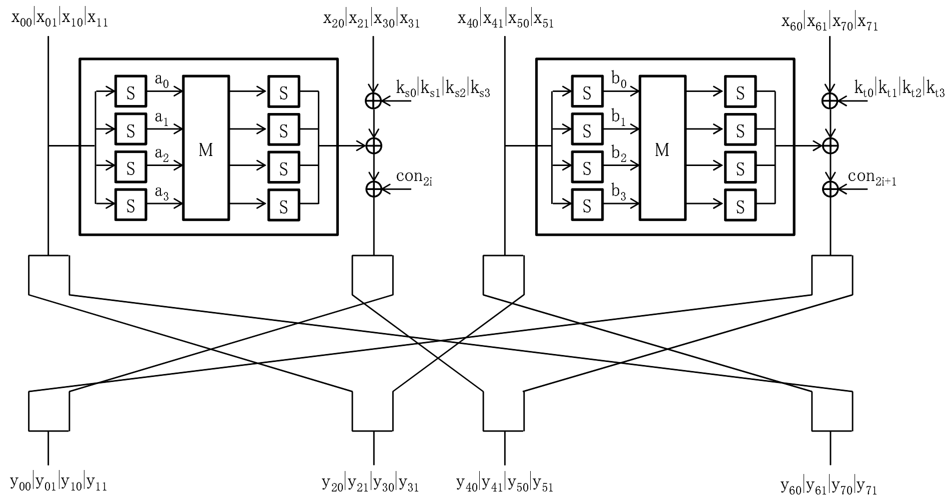


Fig. 20. A round of decryption process

Table 4. Contents of registers R_{ij} $(0 \leq i, j < 4)$ at the l-th cycle during decryption mode

l	0	1	2	3	4	5	6	7	8
R_0	$S(x_0)$ $(= a_0)$	a_1	a_2	a_3	a_0	a_1	a_2	a_3	$S(x_{40})$ $(= b_0)$
R_1	$S(x_1)$ $(= a_1)$	a_2	a_3	a_0	a_1	a_2	a_3	$S(x_{40})$	$S(x_{41})$ $(= b_1)$
R_2	$S(x_2)$ $(= a_2)$	a_3	a_0	a_1	a_2	a_3	$S(x_{40})$	$S(x_{41})$	$S(x_{50})$ $(= b_2)$
R_3	$S(x_3)$ $(= a_3)$	a_0	a_1	a_2	a_3	$S(x_{40})$	$S(x_{41})$	$S(x_{50})$	$S(x_{51})$ $(= b_3)$
R_4	x_{20}	x_{21}	x_{30}	x_{31}	x_{40}	x_{41}	x_{50}	x_{51}	x_{60}
R_5	x_{21}	x_{30}	x_{31}	x_{40}	x_{41}	x_{50}	x_{51}	x_{60}	x_{61}
R_6	x_{30}	x_{31}	x_{40}	x_{41}	x_{50}	x_{51}	x_{60}	x_{61}	x_{70}
R_7	x_{31}	x_{40}	x_{41}	x_{50}	x_{51}	x_{60}	x_{61}	x_{70}	x_{71}
R_8	x_{40}	x_{41}	x_{50}	x_{51}	x_{60}	x_{61}	x_{70}	x_{71}	y_{40}
R_9	x_{41}	x_{50}	x_{51}	x_{60}	x_{61}	x_{70}	x_{71}	y_{40}	y_{41}
R_{10}	x_{50}	x_{51}	x_{60}	x_{61}	x_{70}	x_{71}	y_{40}	y_{41}	y_{10}
R_{11}	x_{51}	x_{60}	x_{61}	x_{70}	x_{71}	y_{40}	y_{41}	y_{10}	y_{11}
R_{12}	x_{60}	x_{61}	x_{70}	x_{71}	y_{40}	y_{41}	y_{10}	y_{11}	x_{00} $(= y_{20})$
R_{13}	x_{61}	x_{70}	x_{71}	y_{40}	y_{41}	y_{10}	y_{11}	x_{00}	x_{01} $(= y_{21})$
R_{14}	x_{70}	x_{71}	y_{40}	y_{41}	y_{10}	y_{11}	x_{00}	x_{01}	x_{10} $(= y_{70})$
R_{15}	x_{71}	y_{40}	y_{41}	y_{10}	y_{11}	x_{00}	x_{01}	x_{10}	x_{11} $(= y_{71})$

l	8	9	10	11	12	13	14	15	16
R_0	b_0	b_1	b_2	b_3	b_0	b_1	b_2	b_3	$S(y_{00})$ $(= a'_0)$
R_1	b_1	b_2	b_3	b_0	b_1	b_2	b_3	$S(y_{00})$	$S(y_{01})$ $(= a'_1)$
R_2	b_2	b_3	b_0	b_1	b_2	b_3	$S(y_{00})$	$S(y_{01})$	$S(y_{10})$ $(= a'_2)$
R_3	b_3	b_0	b_1	b_2	b_3	$S(y_{00})$	$S(y_{01})$	$S(y_{10})$	$S(y_{11})$ $(= a'_3)$
R_4	x_{60}	x_{61}	x_{70}	x_{71}	y_{00}	y_{01}	y_{10}	y_{11}	y_{20}
R_5	x_{61}	x_{70}	x_{71}	y_{00}	y_{01}	y_{10}	y_{11}	y_{20}	y_{21}
R_6	x_{70}	x_{71}	y_{00}	y_{01}	y_{10}	y_{11}	y_{20}	y_{21}	x_{50} $(= y_{30})$
R_7	x_{71}	y_{00}	y_{01}	y_{10}	y_{11}	y_{20}	y_{21}	x_{50}	x_{51} $(= y_{31})$
R_8	y_{40}	y_{41}	y_{10}	y_{11}	y_{20}	y_{21}	y_{70}	y_{71}	y_{40}
R_9	y_{41}	y_{10}	y_{11}	y_{20}	y_{21}	y_{70}	y_{71}	y_{40}	y_{41}
R_{10}	y_{10}	y_{11}	y_{20}	y_{21}	y_{70}	y_{71}	y_{40}	y_{41}	y_{50}
R_{11}	y_{11}	y_{20}	y_{21}	y_{70}	y_{71}	y_{40}	y_{41}	y_{50}	y_{51}
R_{12}	y_{20}	y_{21}	y_{70}	y_{71}	y_{40}	y_{41}	y_{50}	y_{51}	x_{40} $(= y_{60})$
R_{13}	y_{21}	y_{70}	y_{71}	y_{40}	y_{41}	y_{50}	y_{51}	x_{40}	x_{41} $(= y_{61})$
R_{14}	y_{70}	y_{71}	y_{40}	y_{41}	y_{50}	y_{51}	x_{40}	x_{41}	y_{70}
R_{15}	y_{71}	y_{40}	y_{41}	y_{50}	y_{51}	x_{40}	x_{41}	y_{70}	y_{71}

Compact Implementation and Performance Evaluation of Block Ciphers in ATtiny Devices

Thomas Eisenbarth[1], Zheng Gong[2], Tim Güneysu[3], Stefan Heyse[3], Sebastiaan Indesteege[4,5], Stéphanie Kerckhof[6], François Koeune[6], Tomislav Nad[7], Thomas Plos[7], Francesco Regazzoni[6,8], François-Xavier Standaert[6], Loic van Oldeneel tot Oldenzeel[6].

[1] Department of Mathematical Sciences, Florida Atlantic University, FL, USA.
[2] School of Computer Science, South China Normal University.
[3] Horst Görtz Institute for IT Security, Ruhr-Universität, Bochum, Germany.
[4] Department of Electrical Engineering ESAT/COSIC, KULeuven, Belgium.
[5] Interdisciplinary Institute for BroadBand Technology (IBBT), Ghent, Belgium.
[6] UCL Crypto Group, Université catholique de Louvain, Belgium.
[7] Institute for Applied Information Processing and Communications (IAIK), Graz University of Technology, Austria.
[8] ALaRI Institute, University of Lugano, Switzerland.

Abstract. The design of lightweight block ciphers has been a very active research topic over the last years. However, the lack of comparative source codes generally makes it hard to evaluate the extent to which different ciphers actually reach their low-cost goals, on different platforms. This paper reports on an initiative aimed to partially relax this issue. First, we implemented 12 block ciphers on an ATMEL ATtiny45 device, and made the corresponding source code available on a webpage, with an open-source license. Common design goals and interface have been sent to all designers in order to enhance the comparability of the implementation results. Second, we evaluated the performances of these implementations according to different metrics, including energy-consumption measurements. Although inherently limited by slightly different design choices, we hope this initiative can trigger more work in this direction, e.g. by extending the list of implemented ciphers, or adding countermeasures against physical attacks in the future.

1 Introduction

Small embedded devices (including smart cards, RFIDs, sensor nodes) are now deployed in many applications. They are usually characterized by strong cost constraints. Yet, as they may manipulate sensitive data, they also require cryptographic protection. As a result, many lightweight ciphers have been proposed in order to allow strong security guarantees at a lower cost than standard solutions. Quite naturally, the very idea of "low-cost" is highly dependent on the target technology. Some operations that are extremely low cost in hardware (e.g. wire crossings) may turn out to be annoyingly expensive in software. Even within

a class of similar devices (e.g. software), the presence or absence of some options (such as hardware multipliers) may cause strong variations in the performance analysis of different algorithms. As a result, it is sometimes difficult to have a good understanding of which algorithms are actually lightweight on which device. Also, the lack of comparative studies prevents a good understanding of the cost vs. performance tradeoff for these algorithms.

In this paper, we consider this issue of performance evaluation for low-cost block ciphers, and investigate their implementation in ATMEL ATtiny devices [4], i.e. small microcontrollers, with limited memory and instruction set. Despite the relatively frequent use of such devices in different applications, little work has been done in benchmarking cryptographic algorithms in this context. Notable exceptions include B. Poettering's open-source codes for the AES Rijndael [2], the XBX frameworks from CHES 2010 [20] and an interesting survey of lightweight cryptography implementations [10]. Unfortunately, these references are still limited by the number of ciphers under investigation and the fact that their source code is not always available for evaluation.

Following, the goal of our work is to extend the benchmarking of 12 lightweight and standard ciphers, and to make their implementation available under an open-source license. The ciphers were chosen according to three criteria: all selected candidates should (a) give no indication of flawed security, (b) be freely usable without patent restrictions and (c) likely result in lightweight implementations with a footprint of less than 256 bytes of RAM and 4 KB of code size for a combined encryption and decryption function.

In order to make comparisons as meaningful as possible, we tried to adapt the guidelines proposed in [11] for the evaluation of hardware implementations to our software context. Yet, as the project was involving 12 different designers, we also acknowledge that some biases can appear in our conclusions, due to slightly different implementation choices. Hence, as usual for performance evaluations, looking at the source codes is essential in order to properly understand the reasons of different performance figures. Overall, we hope that this initiative can be used as a first step in better analyzing the performances of block ciphers in a specific but meaningful class of devices. We also hope that it can be used as a germ to further develop cryptographic libraries for embedded platforms and, in the long term, add security against physical attacks (e.g. based on faults or side-channel leakage) as another evaluation criteria.

The rest of the paper is structured as follows. Section 2 contains a brief specification of the implemented ciphers. Section 3 establishes our evaluation methodology and metrics, followed by Section 4 that gives details about the ATtiny45 microcontroller. Section 5 provides succinct descriptions and motivation of the implementation choices made by our 12 designers. Finally, our performance evaluations are in Section 6 and conclusions are drawn in Section 7. The webpage containing all our open-source codes is given here [1].

2 List of Investigated Ciphers

AES Rijndael [8] is the new encryption standard selected in 2002 as a replacement of the DES. It supports key sizes of 128, 192 or 256 bits, and block size of 128 bits. The encryption iterates a round function a number of times, depending on the key size. The round is composed of four transformations: SubBytes (that applies a non-linear S-box to the bytes of the states), ShiftRows (a wire crossing), MixColumns (a linear diffusion layer), and finally AddRoundKey (a bitwise XOR of the round key). The round keys are generated from the secret key by means of an expansion routine that re-uses the S-box used in SubBytes. For low-cost application, the typical choice is to support only the key size of 128 bits.

DESL, DESX, and DESXL [15] are lightweight variants of the DES cipher. For the L-variant, all eight DES S-boxes are replaced by a single S-Box with well chosen characteristics to resist known attacks against DES. Additionally the initial permutation (IP) and its inverse (IP^{-1}) are omitted, because they do not provide additional cryptographic strength. The X-variant includes an additional key whitening of the form: $\mathsf{DESX}_{k,k1,k2}(x) = k2 \oplus \mathsf{DES}_k(k1 \oplus x)$. DESXL is the combination of both variants. The main goal of the developer was a low gate count in hardware implementations as for the original DES.

HIGHT [13] is a hardware-oriented block cipher designed for low-cost and low-power applications. It uses 64-bit blocks and 128-bit keys. HIGHT is a variant of the generalized Feistel network and is only composed of simple operations: XOR, mod 2^8 additions and bitwise rotations. Its key schedule consists of two algorithms: one generating whitening key bytes for initial and final transformations; the other one for generating subkeys for the 32 rounds. Each subkey byte is the result of a mod 2^8 addition between a master key byte and a constant generated using a linear feedback shift register.

IDEA [14] is a patented cipher whose patent expired in May 2011 (in all countries with a 20 year term of patent filing). Its underlying Lai-Massey construction does not involve an S-box or a permutation network such as in other Feistel or common SPN ciphers. Instead, it interleaves mathematical operations from three different groups to establish security, such as addition modulo 2^{16}, multiplication modulo $2^{16} + 1$ and addition in GF(2^{16}) (XOR). IDEA has a 128-bit key and 64-bit input and output. A major drawback of its construction is the inverse key schedule that requires the complex extended Euclidean algorithm during decryption. For efficient implementation, this complex key schedule needs to be precomputed and stored in memory.

KASUMI [3] is a block cipher derived from MISTY1 [18]. It is used as a keystream generator in the UMTS, GSM, and GPRS mobile communications systems. It has a 128-bit key and 64-bit input and output. The core of KASUMI is an eight-round Feistel network. The round functions in the main Feistel network are irreversible Feistel-like network transformations. The key scheduling is done by bitwise rotating the 16-bit subkeys or XORing them with a constant. There are two S-boxes, one 7 bit and the other 9 bit.

KATAN and KTANTAN [6] are two families of hardware-oriented block ciphers. They have 80-bit keys and a block size of either 32, 48 or 64 bits. The cipher structure resembles that of a stream cipher, consisting of shift registers and non-linear feedback functions. A LFSR counter is used to protect against slide attacks. The difference between KATAN and KTANTAN lies in the key schedule. KTANTAN is intended to be used with a single key per device, which can then be burnt into the device. This allows KTANTAN to achieve a smaller footprint in a hardware implementation. In the following, we considered the implementation of KATAN with 64-bit block size.

KLEIN [12] is a family of lightweight software oriented block ciphers with 64-bit plaintexts and variable key length (64, 80 or 96 bits - our performance evaluations focus on the 80-bit version). It is primarily designed for software implementations in resource-constrained devices such as wireless sensors and RFID tags, but its hardware implementation can be compact as well. The structure of KLEIN is a typical Substitution-Permutation Network (SPN) with 12/16/20 rounds for KLEIN-64/80/96 respectively. One round transformation consists of four operations AddRoundKey, SubNibbles (4-bit involutive S-box), RotateNibbles and MixNibbles (borrowed from AES MixColumns). The key schedule of KLEIN has a Feistel-like structure. It is agile even if keys are frequently changed and is designed to avoid potential related-key attacks.

mCrypton [16] is another block cipher designed for resource-constrained devices such as RFID tags and sensors. It uses a block length of 64 bits and a variable key length of 64, 96 and 128 bits. In this paper, we implemented the variant with a 96-bit key. mCrypton consists of an AES-like round transformation (12 rounds) and a key schedule. The round transformation operates on a 4×4 nibble array and consists of a nibble-wise non-linear substitution, a column-wise bit permutation, a transposition and a key-addition step. The substitution step uses four 4-bit S-boxes. Encryption and decryption have almost the same form. The key scheduling algorithm generates round keys using non-linear S-box transformations, word-wise rotations, bit-wise rotations and a round constant. The same S-boxes are used for the round transformation and key scheduling.

NOEKEON [7] is a block cipher with a key length and a block size of 128 bits. The block cipher consists of a simple round function based only on bit-wise Boolean operations and cyclic shifts. The round function is iterated 16 times for both encryption and decryption. Within each round, a working key is XORed with the data. The working key is fixed during all rounds and is either the cipher key itself (direct mode) or the cipher key encrypted with a null string. The self-inverse structure of NOEKEON allows to efficiently combine the implementation of encryption and decryption operation with only little overhead.

PRESENT [5] is a hardware-oriented lightweight block cipher designed to meet tight area and power restrictions. It features a 64-bit block size and 80-bit or 128-bit key size (we focus on the 80-bit variant). PRESENT implements a substitution-permutation network and iterates 31 rounds. The permutation layer

consists only of bit permutations (i.e. wire crossings). Together with the tiny 4-bit S-box, the design enables minimalistic hardware implementations. The key scheduling consists of a single S-box lookup, a counter addition and a rotation.

SEA [19] is a scalable family of encryption algorithms, defined for low-cost embedded devices, with variable bus sizes and block/key lengths. In this paper, we implemented $SEA_{96,8}$, i.e. a version of the cipher with 96-bit blocks and keys. SEA is a Feistel cipher that exploits rounds with 3-bit S-boxes, a diffusion layer made of bit and word rotations and a mod 2^n key addition. Its key scheduling is based on rounds similar to the encryption ones and is designed such that keys can be derived "on-the-fly" both in encryption and decryption.

TEA [21] is a 64-bit block cipher using 128-bit keys (although equivalent keys effectively reduce the key space to 2^{126}) . TEA stands for Tiny Encryption Algorithm and, as the name says, this algorithm was built with simplicity and ease of implementation in mind. A C implementation of the algorithm corresponds to about 20 lines of code, and involves no S-box. TEA has a 64-round Feistel structure, each round being based on XOR, 32-bit addition and rotation. The key schedule is also very simple, alternating the two halves of the key at each round. TEA is sensitive to related-key attacks using 2^{23} chosen plaintexts and one related-key query, with a time complexity of 2^{32}.

3 Methodology and Metrics

In order to be able to compare the performances of the different ciphers in terms of speed, memory space and energy, the developers were asked to respect a list of common constraints, detailed hereunder.

1. The code has to be written in assembly, in a single file. It has to be commented and easily readable, for example, giving the functions the name they have in their original specifications.
2. The cipher has to be implemented in a low-cost way, minimizing the code size and the data-memory use.
3. Both encryption and decryption routines have to be implemented.
4. Whenever possible, and in order to minimize the data-memory use, the key schedule has to be computed "on-the-fly". The computation of the key schedule is always included in the algorithm evaluations.
5. The encryption process should start with plaintext and key in data memory. The ciphertext should overwrite the plaintext at the end of this process (and vice versa for decryption).
6. The target device is an 8-bit microcontroller from the ATMEL AVR device family, more precisely the ATtiny45. It has a reduced set of instructions and, e.g. has no hardware multiplier.
7. The encryption and decryption routines are called by a common interface.

The SEA reference code was sent as an example to all designers, together with the common interface (also provided on [1]).

The basic metrics considered for evaluation are code size, number of RAM words, cycle count in encryption and decryption and energy consumption. From these basic metrics, a combined metric was extracted (see Section 6). For the energy-consumption evaluations, each cipher has been flashed in an ATtiny45 mounted on a power-measurement board. A 22 Ohms shunt resistor was inserted between the Vdd pin and the 5V power supply, in order to measure the current consumed by the controller while encrypting. The common interface generates a trigger at the beginning of each encryption, and a second one at the end of each of them. The power traces were measured between those two triggers by our oscilloscope through a differential probe. The plaintexts and keys were generated randomly for each encryption. One hundred encryption traces were averaged for each energy evaluation. The average energy consumed by an encryption has been deduced afterwards, by integrating the measured current.

Note finally that, as mentioned in introduction, the 12 ciphers were implemented by 12 different designers, with slightly different interpretations of the low-cost optimizations. As a result, some of the guidelines were not always followed, because of the cipher specifications making them less relevant. In particular, the following exceptions deserve to be mentioned. (1) The key scheduling of IDEA is not computed "on-the-fly" but precomputed (as explained in Section 2). (2) The key in KATAN has to be restored externally for subsequent invocations. (3) The 4-bit S-boxes of KLEIN, mCrypton, PRESENT were implemented as 8-bit tables (because of a better memory vs. speed tradeoff).

4 Description of the ATtiny45 Microcontroller

The ATtiny45 is an 8-bit RISC microcontroller from ATMEL's AVR series. The microcontroller uses a Harvard architecture with separate instruction and data memory. Instructions are stored in a 4 kB Flash memory (2048×16 bits). Data memory involves the 256-byte static RAM, a register file with 32 8-bit general-purpose registers, and special I/O memory for peripherals like timer, analog-to-digital converter or serial interface. Different direct and indirect addressing methods are available to access data in RAM. Especially indirect addressing allows accessing data in RAM with very compact code size. Moreover, the ATtiny45 has integrated a 256-bytes EEPROM for non-volatile data storage.

The instruction-set of the microcontroller contains 120 instructions which are typically 16-bits wide. Instructions can be divided into arithmetic logic unit (ALU) operations (arithmetic, logical, and bit operations) and conditional and unconditional jump and call operations. The instructions are processed within a two-stage pipeline with a pre-fetch and an execute phase. Most instructions are executed within a single clock cycle, leading to a good instructions-per-cycle ratio. Compared to other microcontrollers from ATMEL's AVR series such as the ATmega devices, the ATtiny45 has a reduced instruction set (e.g. no multiply instruction), smaller memories (Flash, RAM, EEPROM), no in-system debug capability, and less peripherals. However, the ATtiny45 has lower power consumption and is cheaper in price.

5 Implementation Details

AES Rijndael. The code was written following the standard specification and operates on a state matrix of 16 bytes. In order to improve performance, the state is stored into 16 registers, while the key is stored in RAM. Also, 5 temporary registers are used to implement the MixColumn steps. The S-box and the round constants were implemented as simple look-up tables. The multiplication operation needed in the MixColums is computed with shift and XOR instructions.

DESXL. In order to keep code size small, we wrote a function which can compute all permutations and expansions depending on the calling parameters. This function is also capable of writing six bit outputs for direct usage as S-box input. Because of the bit-oriented structure of the permutations which are slow in software, this function is the performance bottleneck of the implementation. The rest of the code is straightforward and is written according to the specification. Beside the storage for plain/ciphertext and the keys k, k_1, k_2, additional 16 bytes of RAM for the round key and the state are required. The S-box and all permutation and expansion tables are stored in Flash memory and processed directly from there.

HIGHT. The implementation choices were oriented in order to limit the code size. First, the intermediate states are stored in RAM at each round, and only two bytes of text and one byte of key are loaded at a time. This way, it is possible to re-use the same code fragment four times per round. Next, the byte rotation at the output of the round function is integrated in the memory accesses of the surrounding functions, in order to save temporary storage and gain cycles. Eight bytes of the subkeys are generated once every two rounds, and are stored in RAM. Finally, excepted for the mod 2^8 additions that are replaced by mod 2^8 subtractions and some other minor changes, decryption uses the same functions as encryption.

IDEA. This cipher was implemented including a precomputed key schedule performed by separate functions for encryption and decryption, respectively, prior the actual cipher operation. During cipher execution the precomputed key (104 bytes) is then read byte by byte from the RAM. The plaintext/ciphertext and the internal state are kept completely in registers (using 16 registers) and 9 additional registers are used for temporary computations and counters. IDEA requires a 16-bit modular multiplication as basic operation. However, in the AVR device used in this work, no dedicated hardware multiplier unit is available. Multiplication was therefore emulated in software resulting in a data-dependent execution time of the cipher operation and an increased cycle count (about a factor of 4) compared to an implementation for a device with a hardware multiplier. Note that IDEA's multiplication is special and maps zero as any input to 2^{16} (which is equivalent to $-1 \bmod 2^{16} + 1$). Therefore, whenever a zero is detected as input to the multiplication, our implementations returns the additive inverse of the other input, reduced modulo $2^{16} + 1$.

KASUMI. The code was written following the functions described in the cipher specifications. During the execution, the 16-byte key remains stored in the RAM, as well as the 8-byte running state. This allows using only 12 registers and 24 bytes of RAM. Some rearrangements were done to skip unnecessary moves between registers. The 9-bit S-box was implemented in an 8-bit table, with the MSBs concatenated in a secondary 8-bit table. The 7-bit S-box was implemented in an 8-bit table, wasting the MSBs in the memory. The round keys are derived "on-the-fly". Decryption is very similar to encryption, as usual for a Feistel structure.

KATAN-64[1]. The main optimization goal was to limit the code size. The entire state of the cipher is kept in registers during operation. To avoid excessive register pressure, the in- and outputs are stored in RAM, and this RAM space is used to backup the register contents during operation. Only three additional registers need to be stored on the stack. The fact that three rounds of KATAN can be run in parallel was not used in this implementation. Doing so would require more complicated shifting and masking to extract bits from the state, and thus significantly increase the code size, for little or no performance gain. As the KATAN key schedule is computed "on-the-fly", the key in RAM is clobbered and needs to be restored externally for subsequent invocations. Keeping the master key in RAM would require 10 additional words (note that the KTANTAN key schedule does not modify the key, so it does not have this limitation). In order to implement the non-linear functions efficiently, addition instructions were used to compute several logical AND's and XOR's in parallel through carefully positioning the input bits and using masking to avoid undesired carry propagation.

KLEIN-80. Despite the goal of small memory footprint, the 4-bit involutive S-box is stored as an 8-bit table for saving clock cycles. As it can be used in both encryption and decryption, this corresponds to a natural tradeoff between code size and processing speed (a similar choice is made for mCrypton and PRESENT, see the next paragraphs). To save memory usage during processing, the MixNibbles step (borrowed from AES MixColumns) is implemented by a single function without using lookup tables. Overall, 29 registers are used during the computations. Among them, 8 registers correspond to the intermediate state, 10 to the key scheduling, 9 registers are used for temporary storage and two for the round counter.

mCrypton. The reference code directly follows the cipher specification. The implementation aims for a limited code size. Therefore, we tried to reuse as much code as possible for decryption and encryption. In addition, we used up to 20 registers during the computations to reduce the cycle count. 12 registers are used to compute the intermediate state and the key scheduling, 6 registers for temporary storage, one for the current key scheduling constant and one for the round counter. After each round the modified state and key scheduling state

[1] All six variants of the KATAN/KTANTAN family are supported via conditional assembly. Our performance evaluations only focus on the 64-bit version of KATAN.

are stored in RAM. The round key is derived from the key scheduling state and is temporarily stored in RAM. The four 4-bit S-boxes are stored in four 8-bit tables, wasting the 4 most significant bits of each entry, but saving cycle counts. The constants used in the key scheduling algorithm are stored in an 8-bit table.

NOEKEON. The implementation aims to minimize the code size and the number of utilized registers. During execution of the block cipher, input data and cipher key are stored in the RAM (32 bytes are required). In that way, only 4 registers are used for the running state, one register for the round counter, and three registers for temporary computations. The X-register is used for indirect addressing of the data in the RAM. Similar to the implementation of SEA (detailed below), using more registers for the running state will decrease the cycle count, but will also increase the code size because of a less generic programming. For decrypting data, the execution sequence of the computation functions is changed, which leads only to a very small increase in code size.

PRESENT. The implementation is optimized in order to limit the code size with throughput as secondary criteria. State and round key are stored in the registers to minimize accesses to RAM. The S-boxes are stored as two 256-byte tables, one for encryption and one for decryption. This allows for two S-box lookups in parallel. However, code size can easily be reduced if only encryption or decryption is performed. A single 16-byte table for the S-boxes could halve the overall code size, but would significantly impact encryption times. The code for permutation, which is the true performance bottleneck, can be used for both encryption and decryption.

SEA. The reference code was written following directly the cipher specifications. During its execution, plaintexts and keys are stored in RAM (accounting for a total of 24 bytes), limiting the register consumption to 6 registers for the running state, one register for the round counter and three registers of temporary storage. Note that higher register consumption would allow decreasing the cycle count at the cost of a less generic programming. The S-box was implemented using its bitslice representation. Decryption uses exactly the same code as encryption, with "on-the-fly" key derivation in both cases.

TEA. Implementing TEA is almost straightforward due to the simplicity of the algorithm. The implementation was optimized to limit the RAM usage and code size. As far as RAM is concerned, we only use the 24 bytes needed for plaintext and key storage, with the ciphertext overwriting the plaintext in RAM at the end of the process. The only notable issue regarding implementing TEA concerns rotations. TEA was optimized for a 32-bit architecture and the fact that only 1-position shift and rotations are available on the ATtiny, plus the need to propagate carries, made these operations slightly more complex. In particular, 5-position shifts were optimized by replacing them by a 3-position shift in the opposite direction and recovering boundary carries. Nonetheless, TEA proved to be very easy to implement, resulting in a compact code of 648 bytes.

6 Performance Evaluation

We considered 6 different metrics: code size (in bytes), RAM use (in bytes), cycle count in encryption and decryption, energy consumption and a combined metric, namely the code size $\times$ cycle count product, normalized by the block size. The results for our different implementations are given in Figures 2, 3, 4, 5, 6, 7 (all given in appendix). We detail a few meaningful observations below.

First, as our primary goal was to consider compact implementations, we compared our code sizes with the ones listed in [10]. As illustrated in Figure 1, we reduced the memory footprint for most investigated ciphers, with specially strong improvements for DESXL, HIGHT and SEA.

Next, the code sizes of our new implementations are in Figure 2. The frontrunners are HIGHT, NOEKEON, SEA and KATAN (all take less than 500 bytes of ROM). One can notice the relatively poor performances of mCrypton, PRESENT and KLEIN. This can in part be explained by the hardware-oriented flavor of these ciphers (e.g. the use of bit permutations or manipulation of 4-bit nibbles is not optimal in 8-bit microcontrollers). As expected, standard ciphers such as the AES and KASUMI are more expensive, but only up to a limited extent (both are implemented in less than 2000 bytes of ROM).

The RAM use in Figure 3 first exhibits the large needs of IDEA regarding this metric (232 words) that are essentially due to the need to store a precomputed key schedule for this cipher. Besides, and following our design guidelines, this metric essentially reflects the size of the intermediate state that has to be stored during the execution of the algorithms. Note that for the AES, this is in contrast with the "Furious" implementation in [2], that uses 192 bytes of RAM (it also explains our slightly reduced performances for this cipher).

The cycle count in Figure 4 clearly illustrates the performance loss that is implied by the use of simple round functions in most lightweight ciphers. This loss is critical for DESXL and KATAN where the large number of round iterations lead to cycle counts beyond 50,000 cycles. It is also large for SEA, NOEKEON and HIGHT. By contrast, these metrics show the excellent efficiency of the AES Rijndael. Cycle count for decryption (Figure 5) shows similar results, with noticeable changes. Most visibly, IDEA decryption is much less efficient than its encryption. The AES also shows non-negligible overhead to decrypt. By contrast, a number of ciphers behave identically in encryption and decryption, e.g. SEA where the two routines are almost identical.

As expected, the energy consumption of all the implemented ciphers (Figure 6) is strongly correlated with the cycle count, confirming the experimental results in [9]. However, slight code dependencies can be noticed. It is an interesting scope for research to investigate whether different coding styles can further impact the energy consumption and to what extent.

Eventually, the combined metric in Figure 7 first shows the excellent size vs. performance tradeoff offered by the AES Rijndael. Among the low-cost ciphers, NOEKEON and TEA exhibit excellent figures as well, probably due to their very simple key scheduling. This comes at the cost of possible security concerns

regarding related-key attacks. HIGHT and KLEIN provide a good tradeoff between code size and cycle count. A similar comment applies to SEA, where parts of the overhead comes from a complex key scheduling algorithm (key rounds are as complex as the rounds for this cipher). Despite their hardware-oriented nature, PRESENT and mCrypton offer decent performance in 8-bit devices as well. KATAN falls a bit behind, mainly because of its very large cycle count. Only DESXL appears not suitable for such an implementation context.

7 Conclusion

This paper reported on an initiative to evaluate the performance of different standard and lightweight block ciphers on a low cost micro-controller. 12 different ciphers have been implemented with compactness as main optimization criteria. Their source code is available on a webpage, under an open-source license. Our results improve most prior work obtained for similar devices. They highlight the different tradeoffs between code size and cycle count that is offered by different algorithms. They also put forward the weaker performances of ciphers that were specifically designed with hardware performance in mind. Scopes for further research include the extension of this work towards more algorithms and the addition of countermeasures against physical attacks.

Acknowledgements. This work has been funded in part by the European Commission's ECRYPT-II NoE (ICT-2007-216676), by the Belgian State's IAP program P6/26 BCRYPT, by the ERC project 280141 (acronym CRASH), by the 7th framework European project TAMPRES, by the Walloon region's S@T Skywin, MIPSs and NANOTIC-COSMOS projects. Stéphanie Kerckhof is a PhD student funded by a FRIA grant, Belgium. F.-X. Standaert is a Research Associate of the Belgian Fund for Scientific Research (FNRS-F.R.S). Zheng Gong is supported by NSFC (No. 61100201). The authors would like to thank Svetla Nikova for her help regarding the implementation of the block cipher KLEIN.

References

1. http://perso.uclouvain.be/fstandae/lightweight_ciphers/.
2. http://point-at-infinity.org/avraes/.
3. 3rd Generation Partnership Project. Technical specification group services and system aspects, 3g security, specification of the 3gpp confidentiality and integrity algorithms, document 2: Kasumi specification (release 10), 2011.
4. ATMEL. Avr 8-bit microcontrollers, http://www.atmel.com/products/avr/.
5. A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. Present: An ultra-lightweight block cipher. In P. Paillier and I. Verbauwhede, editors, *CHES*, volume 4727 of *LNCS*, pages 450–466. Springer, 2007.
6. C. D. Cannière, O. Dunkelman, and M. Knezevic. Katan and ktantan - a family of small and efficient hardware-oriented block ciphers. In C. Clavier and K. Gaj, editors, *CHES*, volume 5747 of *LNCS*, pages 272–288. Springer, 2009.

7. J. Daemen, M. Peeters, G. V. Assche, and V. Rijmen. Nessie proposal: NOEKEON, 2000. Available online at `http://gro.noekeon.org/Noekeon-spec.pdf`.
8. J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.
9. G. de Meulenaer, F. Gosset, F.-X. Standaert, and O. Pereira. On the energy cost of communication and cryptography in wireless sensor networks. In *WiMob*, pages 580–585. IEEE, 2008.
10. T. Eisenbarth, S. S. Kumar, C. Paar, A. Poschmann, and L. Uhsadel. A survey of lightweight-cryptography implementations. *IEEE Design & Test of Computers*, 24(6):522–533, 2007.
11. K. Gaj, E. Homsirikamol, and M. Rogawski. Fair and comprehensive methodology for comparing hardware performance of fourteen round two sha-3 candidates using fpgas. In Mangard and Standaert [17], pages 264–278.
12. Z. Gong, S. Nikova, and Y.-W. Law. Klein: A new family of lightweight block ciphers. to appear in the proceedings of RFIDsec 2011.
13. D. Hong, J. Sung, S. Hong, J. Lim, S. Lee, B. Koo, C. Lee, D. Chang, J. Lee, K. Jeong, H. Kim, J. Kim, and S. Chee. Hight: A new block cipher suitable for low-resource device. In L. Goubin and M. Matsui, editors, *CHES*, volume 4249 of *LNCS*, pages 46–59. Springer, 2006.
14. X. Lai and J. L. Massey. A proposal for a new block encryption standard. In *EUROCRYPT*, pages 389–404, 1990.
15. G. Leander, C. Paar, A. Poschmann, and K. Schramm. New lightweight des variants. In A. Biryukov, editor, *FSE*, volume 4593 of *LNCS*, pages 196–210. Springer, 2007.
16. C. H. Lim and T. Korkishko. mcrypton - a lightweight block cipher for security of low-cost rfid tags and sensors. In J. Song, T. Kwon, and M. Yung, editors, *WISA*, volume 3786 of *LNCS*, pages 243–258. Springer, 2005.
17. S. Mangard and F.-X. Standaert, editors. *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *LNCS*. Springer, 2010.
18. M. Matsui. New block encryption algorithm misty. In E. Biham, editor, *FSE*, volume 1267 of *LNCS*, pages 54–68. Springer, 1997.
19. F.-X. Standaert, G. Piret, N. Gershenfeld, and J.-J. Quisquater. Sea: A scalable encryption algorithm for small embedded applications. In J. Domingo-Ferrer, J. Posegga, and D. Schreckling, editors, *CARDIS*, volume 3928 of *LNCS*, pages 222–236. Springer, 2006.
20. C. Wenzel-Benner and J. Gräf. Xbx: external benchmarking extension for the supercop crypto benchmarking framework. In Mangard and Standaert [17], pages 294–305.
21. D. J. Wheeler and R. M. Needham. Tea, a tiny encryption algorithm. In B. Preneel, editor, *FSE*, volume 1008 of *LNCS*, pages 363–366. Springer, 1994.

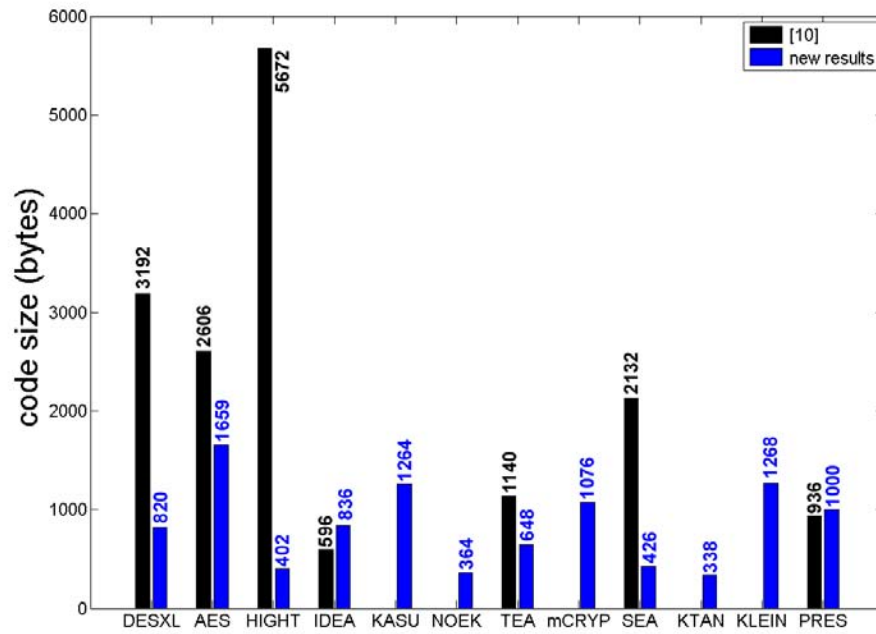


Fig. 1. Code size: comparison with previous work [10].

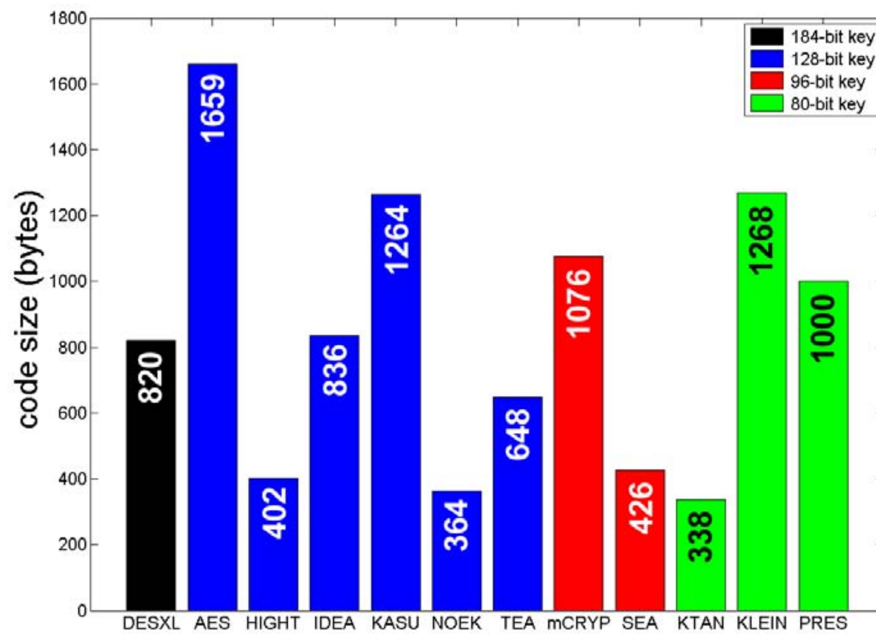


Fig. 2. Performance evaluation: code size.

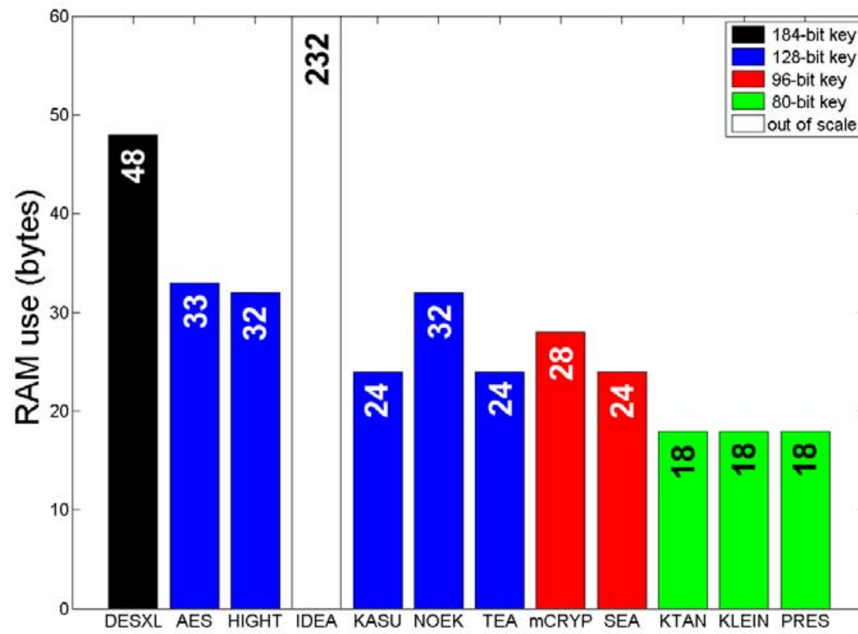


Fig. 3. Performance evaluation: RAM use.

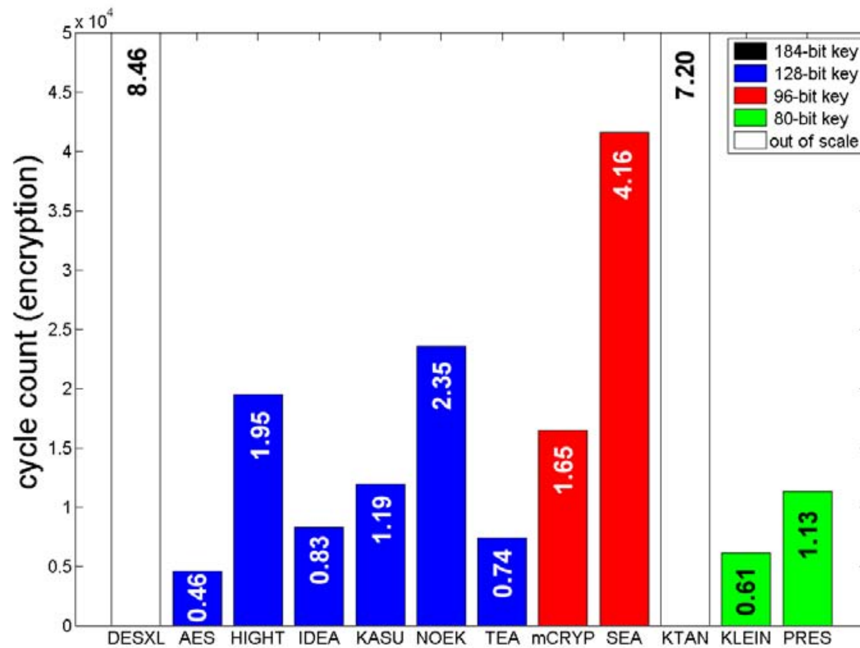


Fig. 4. Performance evaluation: cycle count (encryption).

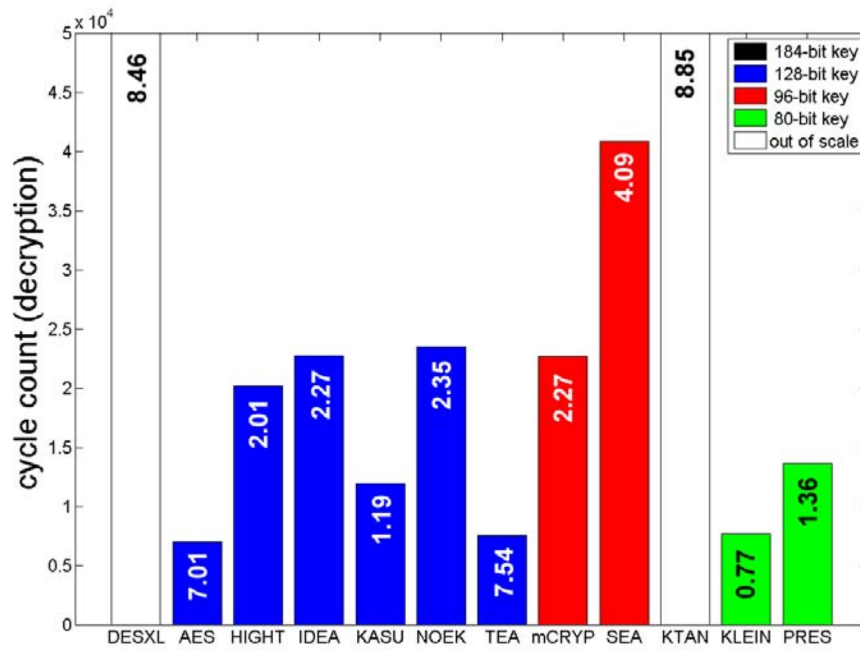


Fig. 5. Performance evaluation: cycle count (decryption).

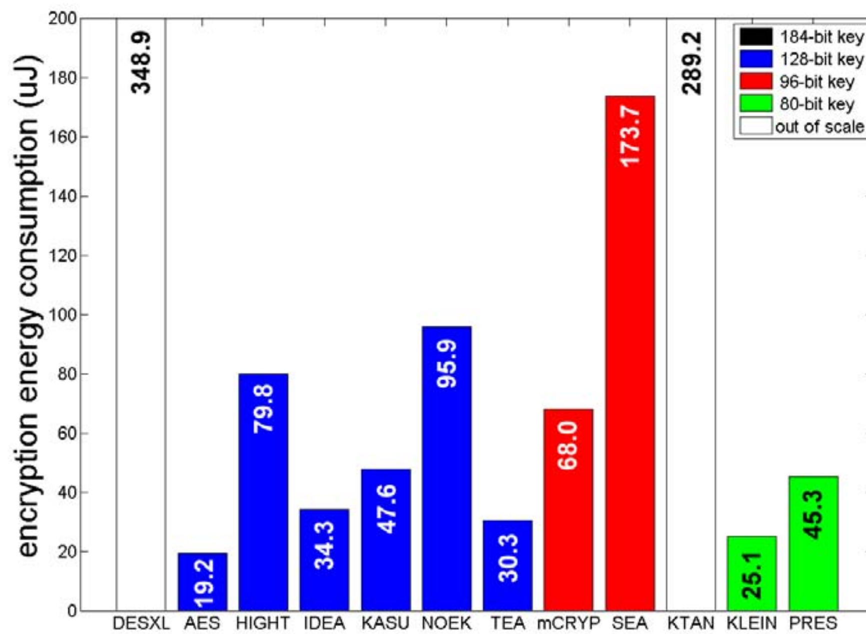


Fig. 6. Performance evaluation: energy consumption.

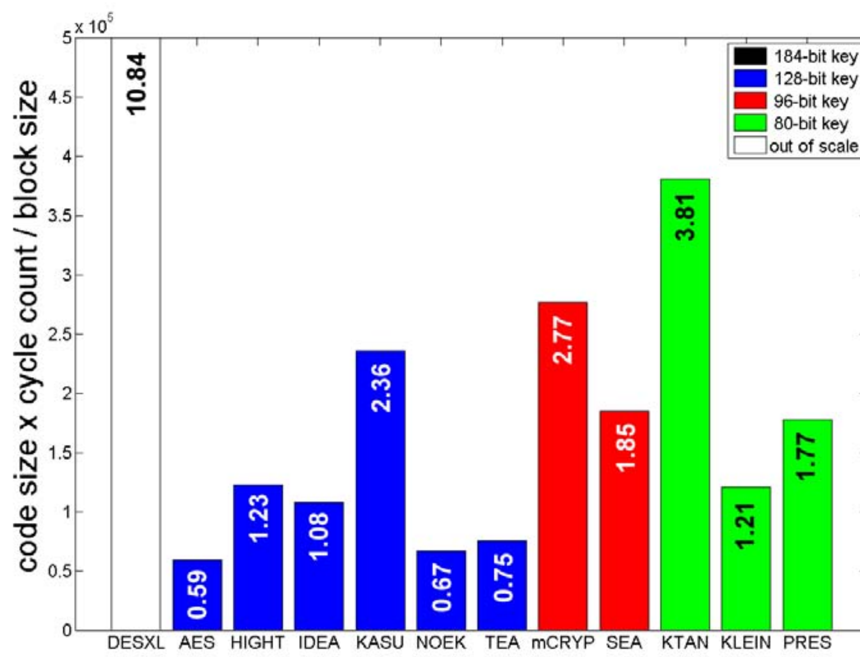


Fig. 7. Performance evaluation: combined metric.

High Speed Implementation of Authenticated Encryption for the MSP430X Microcontroller

Conrado P. L. Gouvêa*, Julio López

University of Campinas (Unicamp),
{conradoplg,jlopez}@ic.unicamp.br

Abstract. Authenticated encryption is a symmetric cryptography algorithm that provides both confidentiality and authentication in a single scheme. In this work we describe an optimized implementation of authenticated encryption for the MSP430X family of microcontrollers. The CCM, GCM, SGCM, OCB3 and Hummingbird-2 modes of authenticated encryption were implemented in the 128-bit level of security and their performance was compared. The AES accelerator present in some models of the MSP430X family is also studied and we explore its characteristics to improve the performance of the implemented modes, achieving up to 13 times of speedup.

1 Introduction

Constrained platforms such as sensor nodes, smart cards and radio-frequency identification (RFID) devices have a great number of applications, many of which with security requirements that require cryptographic schemes. The implementation of such schemes in these devices is very challenging since it must provide high speed while consuming a small amount of resources (energy, code size and RAM).

In this scenario, symmetric cryptography becomes an essential tool in the development of security solutions, since it can provide both confidentiality and authenticity after being bootstrapped by some protocol for key agreement or distribution (for example, with public key cryptography using elliptic curves [7], or identity-based cryptography using pairings [13]). Encryption and authentication can be done through generic composition of separate methods; however, the study of an alternative approach named authenticated encryption (AE) has gained popularity.

Authenticated encryption provides both confidentiality and authenticity within a single algorithm. It is often more efficient than using separate methods and usually consumes a smaller amount of resources. It also prevents common critical mistakes when combining encryption and authentication such as not using separate keys for each task, applying the methods in the wrong order (encrypt-then-authenticate is the only order proven to work with any underlying secure schemes [1]), or not authenticating the initialization vector used for encryption.

* Supported by FAPESP, grant 2010/15340-3.

There are many AE modes; see e.g. [9] for a non-exhaustive list. In this work, we follow the approach from [9] and compare the Counter with CBC-MAC (CCM) mode [20], the Galois/Counter Mode (GCM) [12] and the Offset Codebook (OCB3) mode [9]. We have also implemented the Sophie Germain Counter Mode [16] and the Hummingbird-2 cipher [4]. The CCM mode and GCM have been standardized by the National Institute of Standards and Technology (NIST); CCM is used for Wi-Fi WPA2 security (IEEE 802.11i) while GCM is used in TLS, IPSec and NSA Suite B, for example. The recently proposed OCB3 mode is the third iteration of the OCB mode and appears to be very efficient in multiple platforms. The SGCM is a variant of GCM and was proposed to be resistant against some existing attacks against GCM while being equally or more efficient; we have implemented it in order to check this claim and compare it to GCM. The Hummingbird-2 (which may be referred to as HB2 in this work) is specially suited for 16-bit platforms and was implemented in order to compare it to the other non-specially suited modes.

The main goal of this work is to provide an efficient implementation and comparison of the aforementioned AE modes (CCM, GCM, SGCM, OCB3 and Hummingbird-2) for the MSP430X microcontroller family from Texas Instruments. This family is an extension of the MSP430 which have been used in multiple scenarios such as wireless sensor networks [8,19]; furthermore, some microcontrollers of this family feature an AES accelerator module which can encrypt and decrypt using 128-bit keys. Our contributions are: to study (for the first time, to the best of our knowledge) the efficient usage and impact of this AES accelerator module in the implemented AE modes; to describe a high speed implementation of those AE modes for the MSP430X, achieving performance more than 13 times faster for CCM using the AES accelerator instead of AES in software; to show that CCM is the fastest of those modes whenever a non-parallel AES accelerator is available; and to provide a comparison of the five AE modes, with and without the AES accelerator. We remark that the results regarding the efficient usage of the AES accelerator can probably be applied to other devices featuring analogue accelerators, such as the AVR XMEGA.

This paper is organized as follows. In Section 2, the MSP430X microcontroller family is described. Section 3 offers an introduction to AE, along with a description and comparison of the implemented modes. Our implementation is described in Section 4, and the obtained results are detailed in Section 5. Section 6 provides concluding remarks.

2 The MSP430X Family

The MSP430X family is composed by many microcontrollers which share the same instruction set and 12 general purpose registers. Although it is essentially a 16-bit architecture, its registers have 20 bits, supporting up to 1 MB of addressing space. Each microcontroller model has distinct clock frequency, RAM and ROM (flash) size.

Some MSP430X microcontrollers (namely the CC430 series) have an integrated radio frequency transceiver, making them very suitable for wireless sensors. These models also feature an AES accelerator module that supports encryption and decryption with 128-bit keys only. The study of this accelerator is one key aspect of this study and for this reason we describe its basic usage as follows. In order to encrypt a block of 16 bytes, a flag must be set in a control register to specify encryption and the key must be written sequentially (in bytes or words) in a specific memory address. The input block must then be written, also sequentially, in another memory address. After 167 clock cycles, the result is ready and must be read sequentially from a third address. It is possible to poll a control register to check if the result is ready. Further blocks can be encrypted with the same key without writing the key again. The decryption follows the same procedure, but it requires 214 clock cycles of processing. It is worth noting that these memory read and writes are just like regular reads and writes to the RAM, and take the same time to be performed.

3 Authenticated Encryption Modes

An authenticated encryption mode is composed of two algorithms: authenticated encryption and decryption-verification (of integrity). The authenticated encryption algorithm is denoted by the function $\mathcal{E}_K(N, M, A)$ that returns (C, T), where $K \in \{0,1\}^k$ is the k-bit key, $N \in \{0,1\}^n$ is the n-bit nonce, $M \in \{0,1\}^*$ is the message, $A \in \{0,1\}^*$ is the associated data, $C \in \{0,1\}^*$ is the ciphertext and $T \in \{0,1\}^t$ is the authentication tag. The nonce is a non-secret value that must be unique for each message and prevents the same plaintext being always encrypted to the same ciphertext. (Some modes support variable-length nonces, but we have fixed its size to simplify the exposition. The same applies to the tag.) The associated data (AD) is authenticated by the algorithm, but not encrypted; this can be useful if some header must be sent in plaintext along with the encrypted message, for example, in an internet packet. The decryption-verification algorithm is denoted by the function $\mathcal{D}_K(N, C, A, T)$ that returns (M, V) where K, N, C, A, T, M are as above and V is a boolean value indicating if the given tag is valid (i.e. if the decrypted message and associated data are authentic).

Most AE modes are built using a block cipher such as AES. Let $E_K(B)$ denote the block cipher, where the key K is usually the same used in the AE mode and $B \in \{0,1\}^b$ is a b-bit message (a *block*). The inverse (decryption) function is denoted $D_K(B)$.

It is possible to identify several properties of AE modes; we offer a non-exhaustive list. The *number of block cipher calls* used in the mode is an important metric related to performance. A mode is considered *online* if it is able to encrypt a message with unknown length using constant memory (this is useful, for example, if the end of the data is indicated by a null terminator or a special packet). Some modes *only use the forward function* of the underlying block cipher (E_K), which reduces the size of software and hardware implementations. A mode *supports preprocessing of static AD* if the authentication of the AD de-

pends only on the key and can be cached between different messages being sent (this is useful for a header that does not change). Some modes are *covered by patents*, which usually discourages its use. A mode is *parallelizable* if it is possible to process multiple blocks (or partially process them) in a parallel manner. Some modes support *processing regular messages and AD in any order*, while some modes require the processing of AD before the message, for example. The properties of the AE modes implemented in this work are compared in Table 1.

The following notation is used in the description of the algorithms. Let $A \oplus B$ denote the logical xor of the bit strings A and B (if the strings have different sizes, align them to left and discard the excess from the larger string) and let $A \mid B$ denote the logical *or* in the same fashion. Let $A[i..j]$ denote the substring of the bit string A starting in the i-th bit and ending in the j-th bit, inclusive. The same slicing notation is used for array of words in Algorithm 4. Let $[i]_n$ denote the n-bit representation of the integer i (endianess will be made explicit in the description of the algorithm). Write $A \mid\mid B$ for the concatenation of the bit strings A and B. Write 0^{128} for the block of 128 bits filled with zeros. The AE algorithms will be presented in a simplified manner, omitting details in the handling of incomplete blocks (i.e. with size smaller than the block size) and the decryption-verification algorithms. We refer the reader to the original papers for their complete description.

Table 1. Comparison of implemented AE modes

Property	CCM	(S)GCM	OCB3	HB2
Block cipher calls*	$2m + a + 2^{\dagger}$	m	$m + a + 1^{\dagger}$	—
... in key setup	0	1	1	—
Online	No	Yes	Yes	Yes‡
Uses only E_K	Yes	Yes	No	—
Preprocessing of static AD	No	Yes	Yes	No
Patent-free	Yes	Yes	No	No
Parallelizable	No	Yes	Yes	No
Standardized	Yes	(No) Yes	No	No
Order of message and AD	AD first	AD first	Any	AD last

* m, a are the number of message and AD blocks, respectively
† May have an additional block cipher call
‡ AD size must be fixed

3.1 CCM

The CCM (Counter with CBC-MAC) mode [20] essentially combines the CTR mode of encryption with the CBC-MAC authentication scheme. It requires two cipher block calls for each block to be encrypted. Algorithm 1 presents CCM,

where the function `format` computes a header block B_0 (which encodes the tag length, message length and nonce), the blocks $A_1, \ldots, A_a$ (which encode the length of the associated data along with the data itself) and the blocks $M_1, \ldots, M_m$ which represent the original message. The function `init_ctr` returns the initial counter based on the nonce. The function `inc` increments the counter.

Properties. CCM is not online since the message length is encoded in the header block, which is the first to be processed. It is not very well parallelizable since the authentication requires the result of the previous block cipher call in order to authenticate the current block. It is not possible to preprocess static AD since the authentication depends on B_0, which is based on the nonce. CCM has also been criticized for disrupting word alignment (among other reasons [14]), since the header attached to the AD can have 2, 6 or 10 bytes (this also requires copying chunks of the AD to a buffer before xoring it and sending it to the block cipher).

Endianess issues. The encoding of the lengths and counters must be in big endian format; otherwise, CCM is not affected by endianess issues.

Algorithm 1 CCM encryption

Input: Message M, additional data A, nonce N, key K
Output: Ciphertext C, authentication tag T with t bits
1: $B_0, A_1, \ldots, A_a, M_1, \ldots, M_m \leftarrow \texttt{format}(N, A, M)$
2: $Y \leftarrow E_K(B_0)$
3: **for** $i \leftarrow 1$ **to** a **do**
4: $\quad Y \leftarrow E_K(A_i \oplus Y)$
5: **end for**
6: $J \leftarrow \texttt{init_ctr}(N)$
7: $S_0 \leftarrow E_K(J)$
8: $J \leftarrow \texttt{inc}(J)$
9: **for** $i \leftarrow 1$ **to** m **do**
10: $\quad U \leftarrow E_K(J)$
11: $\quad J \leftarrow \texttt{inc}(J)$ {delay slot}
12: $\quad S \leftarrow M_i \oplus Y$ {delay slot}
13: $\quad Y \leftarrow E_K(S)$
14: $\quad C_i \leftarrow M_i \oplus U$ {delay slot}
15: **end for**
16: $T \leftarrow Y[0..t-1] \oplus S_0[0..t-1]$

3.2 GCM

The GCM (Galois/Counter Mode) [12] employs the arithmetic of the finite field (Galois field) $\mathbb{F}_{2^{128}}$ for authentication and the CTR mode for encryption. It requires a single block cipher call for each block to be encrypted. Algorithm 2

describes GCM, where the function `init_ctr` initializes the counter and the function `inc_ctr` increments the counter. The operation $A \cdot B$ denotes the multiplication of A and B in $\mathbb{F}_{2^{128}}$. The mode benefits from precomputed lookup tables since the second operand is fixed for all multiplications (lines 6, 15 and 18 from Algorithm 1).

Properties. GCM is online and parallelizable. In fact, it has most of the "good" properties from Table 1. It is possible to employ different sizes for the precomputation lookup table as a speed/space tradeoff, varying from 256 bytes to 64 KB.

Endianess issues. In order to interpret a string of bytes as a $\mathbb{F}_{2^{128}}$ element, GCM chooses to view the bytes in a little endian fashion. More peculiarly, it treats the bits inside a byte in a reversed manner: the first bit (i.e. obtained with `c & 1` in the C language) is the *most* significant bit. Therefore, the element $a(z) = 1$ is represented as the byte string `80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`.

Algorithm 2 GCM encryption

Input: Message M, additional data A, nonce N, key K
Output: Ciphertext C, authentication tag T with t bits

1: $A_1, \ldots, A_a \leftarrow A$
2: $M_1, \ldots, M_m \leftarrow M$
3: $H \leftarrow E_K(0^{128})$
4: $Y \leftarrow 0^{128}$
5: **for** $i \leftarrow 1$ **to** a **do**
6: $\quad Y \leftarrow (A_i \oplus Y) \cdot H$
7: **end for**
8: $J \leftarrow$ `init_ctr`(N)
9: $S_0 \leftarrow E_K(J)$
10: $J \leftarrow$ `inc`(J)
11: **for** $i \leftarrow 1$ **to** m **do**
12: $\quad U \leftarrow E_K(J)$
13: $\quad J \leftarrow$ `inc`(J) {delay slot}
14: $\quad C_i \leftarrow M_i \oplus U$
15: $\quad Y \leftarrow (C_i \oplus Y) \cdot H$
16: **end for**
17: $L \leftarrow [len(A)]_{64} \,||\, [len(M)]_{64}$
18: $S \leftarrow (L \oplus Y) \cdot H$
19: $T \leftarrow (S \oplus S_0)[0..t-1]$

3.3 SGCM

The SGCM (Sophie Germain Counter Mode) [16] is a variant of GCM that is not susceptible to weak key attacks that exist against GCM. While these attacks

are of limited nature, the author claims that they should be avoided. It has the same structure as GCM, but instead of the $\mathbb{F}_{2^{128}}$ arithmetic, it uses the prime field $\mathbb{F}_p$ with $p = 2^{128} + 12451$.

Properties. The same as GCM.

Endianess issues. Elements of $\mathbb{F}_p$ are represented by little-endian byte arrays. However, unlike GCM, the bits inside a byte are viewed in the usual manner: the first bit (i.e. obtained with `c & 1` in the C language) is the least significant bit. Therefore, the element 1 is represented as the byte string `01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`.

3.4 OCB3

The OCB3 (Offset Codebook) mode [9] also employs the $\mathbb{F}_{2^{128}}$ arithmetic (using the same reduction polynomial from GCM), but in a simplified manner: it does not require full multiplication, but only multiplication by powers of z (the variable used in the polynomial representation of the field elements). It also requires a single block cipher call for each block being encrypted. It is described in Algorithm 3, where the function `init_delta` derives a value from the nonce and it may require a block cipher call, as explained later. The function $\texttt{ntz}(i)$ returns the number of trailing zeros in the binary representation of i (e.g. $\texttt{ntz}(1) = 0$, $\texttt{ntz}(2) = 1$). The function $\texttt{getL(}L_0\texttt{, } x\texttt{)}$ computes the field element $L_0 \cdot z^x$ and can benefit from a precomputed lookup table. Notice that the multiplication by z is simply a left shift of the operand by one bit, discarding the last bit and xoring the last byte of the result with 135 (which is the representation of $z^7+z^2+z^1+1$) if the discarded bit was 1. The function `hash` authenticates the additional data and is omitted for brevity.

Properties. OCB3 is also online and parallelizable, but uses both E_K and D_K and is covered by patents [15]. There is an interesting feature in the `init_delta` function: it requires a block cipher call whose input is the nonce, padded to the left with zeros to fill a block, and with the lower 6 bits set to zero. Therefore, when the nonce is a counter (which is often the case), the block cipher result can be cached between messages, saving a block cipher call 98% of the time.

Endianess issues. In order to interpret a string of bytes as a $\mathbb{F}_{2^{128}}$ element, OCB3 chooses to view the bytes in a big endian fashion. The bits inside a byte are viewed in the usual manner: the first bit (`c & 1`) is the least significant bit. Therefore, the field element $a(z) = 1$ is represented as the byte string `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01`.

3.5 Hummingbird-2 (HB2)

The Hummingbird-2 [4] is an authenticated encryption algorithm which is not built upon a block cipher. It processes 16-bit blocks and was specially designed

Algorithm 3 OCB3 mode encryption

Input: Message M, additional data A, nonce N, key K
Output: Ciphertext C, authentication tag T with t bits
1: $A_1, \ldots, A_a \leftarrow A$
2: $M_1, \ldots, M_m \leftarrow M$
3: $L_* \leftarrow E_K(0^{128})$
4: $L_\$ \leftarrow L_* \cdot z$
5: $L_0 \leftarrow L_\$ \cdot z$
6: $Y \leftarrow 0^{128}$
7: $\Delta \leftarrow \mathtt{init_delta}(N, K)$
8: **for** $i \leftarrow 1$ **to** m **do**
9: $\quad \Delta \leftarrow \Delta \oplus \mathtt{getL}(L_0, \mathtt{ntz}(i))$
10: $\quad U \leftarrow E_K(M_i \oplus \Delta)$
11: $\quad Y \leftarrow Y \oplus M_i$ {delay slot}
12: $\quad C_i \leftarrow U \oplus \Delta$
13: **end for**
14: $\Delta \leftarrow \Delta \oplus L_\$$
15: $F \leftarrow E_K(Y \oplus \Delta)$
16: $G \leftarrow \mathtt{hash}(K, A)$
17: $T \leftarrow (F \oplus G)[0..t-1]$

for resource-constrained platforms. The small block size is achieved by maintaining an 128-bit internal state that is updated with each block processed. Authenticated data is processed after the confidential data by simply processing the blocks and discarding the ciphertext generated. The algorithm is built upon the following functions for encryption:

$$\begin{aligned}
S(x) &= S_4(x[0..3]) \mid (S_3(x[4..7]) \ll 4) \\
&\quad \mid (S_2(x[8..11]) \ll 8) \mid (S_1(x[12..15]) \ll 12) \\
L(x) &= x \oplus (x \lll 6) \oplus (x \lll 10) \\
f(x) &= L(S(x)) \\
\mathtt{WD16}(x, a, b, c, d) &= f(f(f(f(x \oplus a) \oplus b) \oplus c) \oplus d)\,;
\end{aligned}$$

and their inverses for decryption:

$$\begin{aligned}
S^{-1}(x) &= S_4^{-1}(x[0..3]) \mid (S_3^{-1}(x[4..7]) \ll 4) \\
&\quad \mid (S_2^{-1}(x[8..11]) \ll 8) \mid (S_1^{-1}(x[12..15]) \ll 12) \\
L^{-1}(x) &= x \oplus (x \lll 2) \oplus (x \lll 4) \oplus (x \lll 12) \oplus (x \lll 14) \\
f^{-1}(x) &= S^{-1}(L^{-1}(x)) \\
\mathtt{WD16}^{-1}(x, a, b, c, d) &= f^{-1}(f^{-1}(f^{-1}(f^{-1}(x) \oplus d) \oplus c) \oplus b) \oplus a\,;
\end{aligned}$$

where S_1, S_2, S_3, S_4 are S-boxes and $\lll$ denotes the circular left shift of a 16-bit word. The function `WD16` is called four times for each block, therefore f is called 16 times for each block. We refer to [4] for further details.

Properties. Hummingbird-2 is patented. The AD must have fixed size, and must be processed after the confidential data.

Endianess issues. The algorithm uses the addition of 16-bit words, which must be viewed in little-endian fashion.

4 Efficient Implementation

We have written a fast software implementation of the AE modes in the C language, with critical functions written in assembly. The target chip was a CC430F6137 with 20 MHz clock, 32 KB flash for code and 4 KB RAM. The compiler used was the IAR Embedded Workbench version 5.20.

The interface to the AES accelerator was written in assembly, along with a function to xor two blocks and another to increment a block. In order to speed up the GCM mode, polynomial multiplication was implemented in unrolled assembly with the López-Dahab (LD) [11] algorithm using 4-bit window and two lookup tables, as described in Algorithm 4. The first precomputation lookup table holds the product of H and all 4-bit polynomials. Each of the 16 lines of the table has 132 bits, which take 9 words. This leads to a table with 288 bytes. The additional lookup table (which can be computed from the first one, shifting each line 4 bits to the left) allows the switch from three 4-bit shifts of 256-bit blocks to a single 8-bit shift of a 256-bit block, which can be computed efficiently with the `swpb` (swap bytes) instruction of the MSP430.

For SGCM, arithmetic in $\mathbb{F}_p$ can be carried with known algorithms such as Comba multiplication. We follow the approach in [6] which takes advantage of the multiply-and-accumulate operation present in the hardware multiplier of the MSP430 family. However, it must be noted that the CC430 series has a 32-bit multiplier; we have used it to improve even more the performance of the multiplication. The special form of the prime p allows fast modular reduction by taking advantage of the congruence $2^{128}x \equiv -12451 \pmod{p}$ — that is, to reduce a 256-bit value module p, multiply the higher 128 bits by -12451 and add it to the lower 128 bits, repeating until a 128-bit value is obtained.

In the OCB3 mode, a lookup table with 8 entries (128 bytes) was used to speed up the `getL` function. Two functions were implemented in assembly: doubling (using left shifts) and the `ntz` function (using right shifts).

For Hummingbird-2, we have unrolled the `WD16` function. The function f is critical since it is called 16 times per block and must be very efficient; our approach is to use two precomputed lookup tables f_L, f_H each one with 256 2-byte elements, such that $f(x) = f_L[x \,\&\, \texttt{0xFF}] \oplus f_H[(x \,\&\, \texttt{0xFF00}) \gg 8]$. These tables are generated by computing $f_L[x] \leftarrow L(S_4(x[0..3]) \mid (S_3(x[4..7]) \ll 4))$ for every byte x and $f_H[x] \leftarrow L((S_2(x[8..11]) \ll 8) \mid (S_1(x[12..15]) \ll 12))$ also for every byte x. This optimization does not apply for $f^{-1}(x)$ since the inverse S-boxes are applied after the shifts in $L^{-1}(x)$ (this is the reason why decryption is slower than encryption, as will be shown). In this case, we have used precomputed lookup tables L_L, L_H such that $L(x) = L_L[x \,\&\, \texttt{0xFF}] \oplus L_H[(x \,\&\, \texttt{0xFF00}) \gg 8]$.

These are computed as $f_L[x] \leftarrow L(x[0..7]), f_H[x] \leftarrow L(x[8..15] \ll 8)$ for every byte x. The four 4-bit inverse S-boxes have been merged in two 8-bit inverse S-boxes S_L^{-1}, S_H^{-1} such that $S^{-1}(x) = S_L^{-1}(x[0..7]) \mid (S_H^{-1}(x[8..15]) \ll 8)$.

In order to perform comparisons, we have used a software implementation of AES from [5] (the byte-oriented version, with the `VERSION_1` option disabled). It uses approximately 2 KB of precomputed lookup tables to improve speed.

Algorithm 4 López-Dahab multiplication in $\mathbb{F}_{2^{128}}$ for 16-bit words and 4-bit window, using 2 lookup tables.

Input: $a(z) = a[0..7], b(z) = b[0..7]$
Output: $c(z) = c[0..15]$
1: Compute $T_0(u) = u(z)b(z)$ for all polynomials $u(z)$ of degree lower than 4.
2: Compute $T_1(u) = u(z)b(z)z^4$ for all polynomials $u(z)$ of degree lower than 4.
3: $c[0..15] \leftarrow 0$
4: **for** $k \leftarrow 1$ **down to** 0 **do**
5:   **for** $i \leftarrow 0$ **to** 7 **do**
6:     $u_0 \leftarrow (a[i] \gg (8k)) \bmod 2^4$
7:     $u_1 \leftarrow (a[i] \gg (8k+4)) \bmod 2^4$
8:     **for** $j \leftarrow 0$ **to** 8 **do**
9:       $c[i+j] \leftarrow c[i+j] \oplus T_0(u_0)[j] \oplus T_1(u_1)[j]$
10:     **end for**
11:   **end for**
12:   **if** $k > 0$ **then**
13:     $c(z) \leftarrow c(z)z^8$
14:   **end if**
15: **end for**
16: **return** c

4.1 Using the AES accelerator

As previously mentioned, the AES encryption and decryption using the AES hardware accelerator requires waiting for 167 and 214 cycles, respectively, before reading the results. The key to a efficient implementation using the module is to use this "delay slot" to carry other operations that do not depend on the result of the encryption/decryption. In the listed algorithms, we have ordered the steps as they were implemented in order to take advantage of the delay slot. Take CCM as example (Algorithm 1). In line 10, J is written as input to the AES accelerator. Then, lines 11 and 12 can be executed in the delay slot. Finally, before line 13, $E_K(J)$ can be read from the AES accelerator (waiting until it is ready).

In CCM, the first delay slot is used to compute an increment and a xor (lines 11 and 12 in Algorithm 1) and the second delay slot is used for a xor (line 14 in Algorithm 1). In GCM, the increment in the line 13 of Algorithm 2 is computed in the delay slot. In OCB3, the delay slot is used to compute the xor in the line 3 of Algorithm 3.

5 Results

The performance of the implemented AE modes was measured for the authenticated encryption and decryption-verification of messages with 16 bytes and 4 KB, along with the Internet Performance Index (IPI) [12], which is a weighted timing for messages with 44 bytes (5%), 552 bytes (15%), 576 bytes (20%), and 1500 bytes (60%). For each message size, we have measured the time to compute all nonce-dependent values along with time for authenticated encryption / decryption-verification with 128-bit tags. The derivation of key-dependent values is not included. For OCB3, it was assumed that the block cipher call in `init_ctr` was cached.

The timings were obtained using a development board with a CC430F6137 chip and are reported on Table 2. The number of cycles taken by the algorithms was measured using the built-in cycle counter present in the CC430 models, which can be read in the IAR debugger. Stack usage was also measured using the debugger. Code size was determined from the reports produced by the compiler, adding the size for text (code) and constants.

Table 2. Timings of implemented AE modes for different message lengths, in cycles per byte

	Using AES accelerator			Using AES in software*		
Mode	16 bytes	IPI	4 KB	16 bytes	IPI	4 KB
Encryption						
CTR†	26	23	23	248	246	245
CCM	131	38	36	1 600	493	479
GCM	426	183	180	863	403	396
SGCM	242	89	87	674	306	301
OCB3	144	39	38	621	261	257
HB2‡				569	200	196
Decryption						
CTR†	26	23	23	248	246	245
CCM	144	47	46	1 603	493	479
GCM	429	183	180	862	404	397
SGCM	243	89	87	675	307	302
OCB3	217	48	46	749	385	382
HB2‡				669	297	292

* Based on the software AES implementation from [5]
† Non-authenticated encryption mode included for comparison
‡ It does not use AES

Using the AES accelerator. First, we analyze the results using the AES accelerator, for IPI and 4 KB messages. The GCM performance is more than 5 times slower than the other modes; this is due to the complexity of the full binary field multiplication. The SGCM is more than 50% faster than GCM, since the prime field arithmetic is much faster on this platform, specially using the 32-bit hardware multiplier. Still, it is slower than the other modes. Both CCM and OCB3 have almost the same speed, with CCM being around 5% faster. This is surprising, since that OCB3 essentially outperforms CCM in many platforms [9]. It is explained by the combination of two facts: the hardware support for AES, which reduces the overhead of an extra block cipher call in CCM; and the fact that the AES accelerator does not support parallelism, which prevents OCB3 from taking advantage of its support for it. We have measured that the delay slot optimization improves the encryption speed of GCM, SGCM and OCB3 by around 12% and CCM by around 24%.

Using the AES in software. We now consider the performance using the software AES implementation, for large messages. For reference, the block cipher takes 231 cycles per byte to encrypt and 356 cycles per byte to decrypt. The CCM mode becomes slower due to the larger overhead of the extra block cipher call. The GCM is still slower than OCB3 due to its expensive field multiplication. The SGCM is also faster than GCM, but the improvement is diluted to 25% with the software AES. The Hummingbird-2 cipher outperforms the other modes and is the most efficient if the AES accelerator is not available.

AES accelerator vs. AES in software. Using the AES accelerator, it is possible to encrypt in the CTR mode approximately 10.5 times faster than using AES in software; and it is possible to encrypt with CCM approximately 13 times faster for encryption and 10.6 times faster for decryption. The AES accelerator speedup for GCM and OCB3 is smaller (around 2.2 and 6.6, respectively), due to the larger software overhead of both.

Encryption vs. decryption. When considering the usage of the AES accelerator, GCM has roughly the same performance in encryption and decryption, since the algorithm for both is almost equal; the same applies for SGCM. For both CCM and OCB3, decryption is around 25% and 20% slower, respectively. This is explained by the differences in the data dependencies of the decryption, which prevents the useful use of the delay slot, and that D_K (used by OCB3) is slower than E_K in the AES accelerator. Considering now the usage of the AES in software, encryption and decryption have the same performance in CCM and GCM (since there is no delay slot now) but decryption is almost 50% slower for OCB3, since the underlying block cipher decryption is also slower than the encryption. This results in the OCB3 decryption being slower than SGCM. The decryption in Hummingbird-2 is almost 50% slower due to the $f^{-1}(x)$ function not being able to be fully precomputed, in contrast to $f(x)$. It is interesting to note that the decryption timings are often omitted in the literature, even though they may be substantially different from the encryption timings.

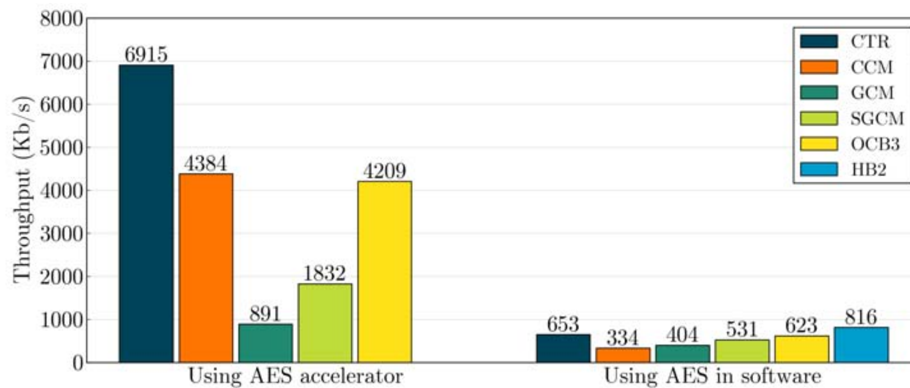


Fig. 1. Encryption throughput in Kbps of CTR and AE modes for 4 KB messages at 20 MHz

Performance for small messages. The timings for 16-byte messages are usually dominated by the computation of nonce-dependent values. The CCM has the worst performance using software AES since all of its initialization is nonce-dependent (almost nothing is exclusively key-dependent) and it includes two block cipher calls. When using the AES accelerator, this overhead mostly vanishes. The nonce setup of GCM is very cheap (just a padding of the nonce) while the nonce setup of OCB3 requires the left shift of an 192-bit block by 0–63 bits. Still, the GCM performance for 16-byte messages is worse than OCB3 since it is still dominated by the block processing. Comparing with the other modes using software AES, the Hummingbird-2 cipher is the fastest for small messages due to its small block size. It is also worth mentioning that, when using smaller tags, Hummingbird-2 is even faster since its tag is generated in 16-bit words at a time while the other algorithms generate a 128-bit tag which can then be truncated.

Further analysis. Figures 1 and 2 present the throughput of encryption and decryption in the CTR and AE modes, considering the 20 MHz clock of the CC430F6137. For comparison, consider the AES software implementation from [3] (also based on [5]) which achieved 286 Kbps at 8 MHz in the ECB mode. Scaling this to 20 MHz we get 716 Kbps, while our ECB implementation achieved 691 Kbps. This is 3.5% slower (probably since we have not spent much time fine-tuning it), but is good enough for our purposes (comparing the performance to the AES accelerator). In [3], it is also claimed that since the maximum throughput of the transceiver is 250 Kbps, it is not needed to encrypt faster than this. This may be true, but a faster (authenticated) encryption uses less energy and may free the controller for other data processing.

Table 3 lists the ROM and RAM usage for programs implementing AE modes for both encryption and decryption, using the AES accelerator. We recall that the MSP430X model we have used features 32 KB of flash for code and 4 KB RAM. The code for GCM is the largest due to the unrolled $\mathbb{F}_{2^{128}}$ multiplier, while

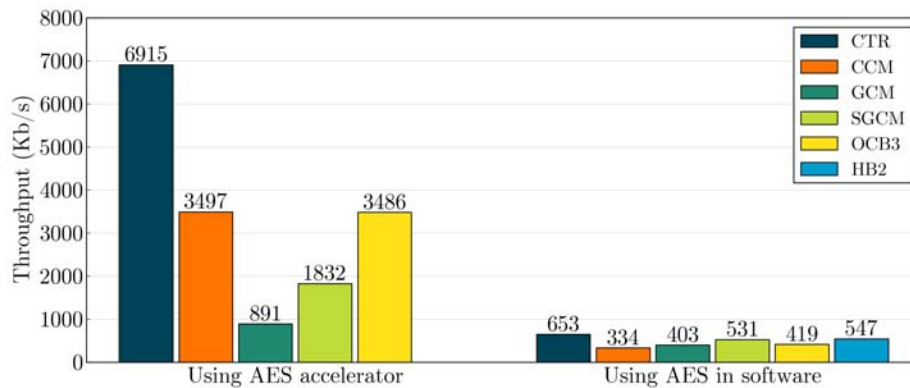


Fig. 2. Decryption throughput in Kbps of CTR and AE modes for 4 KB messages at 20 MHz

the code for CCM is the smallest since it mostly relies on the block cipher. The RAM usage follows the same pattern: GCM has the largest usage, since it has the largest precomputation table; the Hummingbird-2 cipher (followed by CCM) has the smallest RAM usage since it requires no runtime precomputation at all. When using the software AES implementation, 1 564 additional ROM bytes are required for CCM, GCM and SGCM (which use E_K only) and 3 668 additional ROM bytes are required for OCB3.

Table 3. ROM and RAM (stack) usage of AE modes, in bytes. When using software AES, 1 564 additional ROM bytes are required for CCM, GCM and SGCM and 3 668 bytes for OCB3

Mode	ROM	RAM
CTR	688	124
CCM	1 684	250
GCM	5 602	884
SGCM	2 874	322
OCB3	2 382	538
HB2	3 674	196

5.1 Related work

Unfortunately, we are not aware of any works describing implementations of these modes for the MSP430X (except Hummingbird-2), which prevents further comparisons. However, it is still possible to draw some comparisons with related works, as follows.

In [3], the encryption performance using the AES module present in the CC2420 transceiver is studied, achieving 110 cycles per byte. This is still 5 times slower than our results for the CTR mode, probably because the CC2420 is a peripheral and communicating with it is more expensive.

The Dragon-MAC [10] is based on the Dragon stream cipher. Its authors describe an implementation for the MSP430 that achieves 21.4 cycles per byte for authenticated encryption (applying Dragon then Dragon-MAC), which is faster than all timings in this work. However, it requires 18.9 KB of code. Our CCM implementation using the AES accelerator is 1.7 times slower, but 11 times smaller.

The Hummingbird-2 timings reported for the MSP430 in its paper [4] are about 10% faster than the timings we have obtained. However, its authors do not describe their optimization techniques, nor the exact MSP430 model used and their timing methodology, making it difficult to explain their achieved speed. However, we believe that our implementation is good enough for comparisons.

The work [9], whose approach we have followed, provides timings for CTR, CCM, GCM and OCB3 for many platforms, but not for the MSP430. In order to make a comparison, consider the performance relative to the CTR timings. For the x86-64 platform with AES New Instructions, they obtain a performance of 3.28, 2.94 and 1.16 for CCM, GCM and OCB3 respectively; while our results using the AES accelerator are 1.62, 7.87 and 1.71 (notice that our CCM is much faster); using software AES, they are 2.01, 1.64, 1.06 (notice the same ordering of performance).

6 Conclusion

Authenticated encryption modes are a very useful tool in the development of security solutions for constrained platforms. In this work, we have presented an efficient implementation for the MSP430X family of microcontrollers of two popular and standardized AE modes, CCM and GCM, along with the SGCM and OCB3 modes and the Hummingbird-2 cipher. We have also described how to take full advantage of the AES accelerator present in some MSP430X models, achieving a speedup of around 10 times for CTR encryption and CCM compared to the best known timings for a software implementation.

The CCM and OCB3 modes were found to provide similar speed results using the AES accelerator, with CCM being around 5% faster. While OCB3 is the fastest mode in many platforms, we expect CCM to be faster whenever a non-parallel AES accelerator is available. This is the case for the MSP430X models studied and is also the case for other platforms, for example, the AVR XMEGA microcontroller with has an AES module analogue to the MSP430X AES accelerator.

The CCM appears to be the best choice for MSP430X models with AES accelerator considering that it also consumes less code space and less stack RAM. If one of the undesirable properties of CCM must be avoided (not being online, lack of support for preprocessing of static AD), a good alternative is the EAX

mode [2] which is described as a "cleaned-up CCM" by one of its authors and should have performance similar to CCM. The GCM mode, even though it has many good properties, does not appear to be adequate in software implementation for resource-constrained platforms since it requires very large lookup tables in order to offer performance comparable to other modes.

Some other relevant facts we have found are that Hummingbird-2 provided the fastest performance compared to the other modes using AES in software; that SGCM is 50% faster than GCM when using the AES accelerator and 25% when not; and that OCB3 and Hummingbird-2 in particular have a decryption performance remarkably slower than encryption (up to 50%).

Future work. It would be interesting to implement and compare lightweight encrypt-and-authenticate or authenticated encryption schemes such as LetterSoup [17] and Rabbit-MAC [18] for the MSP430X. Another possible venue for research is to study the efficient implementation of authenticated encryption using the AES accelerator featured in other platforms such as the AVR XMEGA and n devices based on the ARM Cortex such as the EFM32 Gecko, STM32 and LPC1800.

References

1. Bellare, M., Namprempre, C.: Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In: Advances in Cryptology — ASIACRYPT 2000, Lecture Notes in Computer Science, vol. 1976, pp. 531–545. Springer Berlin / Heidelberg (2000)
2. Bellare, M., Rogaway, P., Wagner, D.: The EAX mode of operation. In: Fast Software Encryption, Lecture Notes in Computer Science, vol. 3017, pp. 389–407. Springer Berlin / Heidelberg (2004)
3. Didla, S., Ault, A., Bagchi, S.: Optimizing AES for embedded devices and wireless sensor networks. In: Proceedings of the 4th International ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities. pp. 4:1–4:10 (2008)
4. Engels, D., Saarinen, M.J.O., Smith, E.M.: The Hummingbird-2 lightweight authenticated encryption algorithm. Cryptology ePrint Archive, Report 2011/126 (2011), `http://eprint.iacr.org/`
5. Gladman, B.: AES and combined encryption/authentication modes. `http://gladman.plushost.co.uk/oldsite/AES/` (2008)
6. Gouvêa, C.P.L., López, J.: Software implementation of pairing-based cryptography on sensor networks using the MSP430 microcontroller. In: Progress in Cryptology — INDOCRYPT 2009. Lecture Notes in Computer Science, vol. 5922, pp. 248–262. Springer Berlin / Heidelberg (2009)
7. Großschädl, J., Szekely, A., Tillich, S.: The energy cost of cryptographic key establishment in wireless sensor networks. In: Proceedings of the 2nd ACM symposium on information, computer and communications security. pp. 380–382. ASIACCS '07, ACM, New York, NY, USA (2007)
8. Jovanov, E., Milenkovic, A.: Body area networks for ubiquitous healthcare applications: Opportunities and challenges. Journal of Medical Systems pp. 1–10 (2011)

9. Krovetz, T., Rogaway, P.: The software performance of authenticated-encryption modes. In: Fast Software Encryption, Lecture Notes in Computer Science, vol. 6733, pp. 306–327. Springer Berlin / Heidelberg (2011)
10. Lim, S.Y., Pu, C.C., Lim, H.T., Lee, H.J.: Dragon-MAC: Securing wireless sensor networks with authenticated encryption. Cryptology ePrint Archive, Report 2007/204 (2007), http://eprint.iacr.org/
11. López, J., Dahab, R.: High-speed software multiplication in $\mathbb{F}_{2^m}$. In: Progress in Cryptology — INDOCRYPT 2000. Lecture Notes in Computer Science, vol. 1977, pp. 93–102. Springer Berlin / Heidelberg (2000)
12. McGrew, D., Viega, J.: The security and performance of the Galois/Counter Mode (GCM) of operation. In: Progress in Cryptology — INDOCRYPT 2004, Lecture Notes in Computer Science, vol. 3348, pp. 377–413. Springer Berlin / Heidelberg (2005)
13. Oliveira, L.B., Aranha, D.F., Gouvêa, C.P.L., Scott, M., Câmara, D.F., López, J., Dahab, R.: TinyPBC: Pairings for authenticated identity-based non-interactive key distribution in sensor networks. Computer Communications 34(3), 485–493 (2010)
14. Rogaway, P., Wagner, D.: A critique of CCM. Cryptology ePrint Archive, Report 2003/070 (2003), http://eprint.iacr.org/
15. Rogaway, P.: Method and apparatus for facilitating efficient authenticated encryption (2007), US patent 7200227
16. Saarinen, M.J.O.: SGCM: The Sophie Germain counter mode. Cryptology ePrint Archive, Report 2011/326 (2011), http://eprint.iacr.org/
17. Simplicio Jr, M.A., Barbuda, P.F.F.S., Barreto, P.S.L.M., Carvalho, T.C.M.B., Margi, C.B.: The MARVIN message authentication code and the LETTERSOUP authenticated encryption scheme. Security and Communication Networks 2(2), 165–180 (2009)
18. Tahir, R., Javed, M., Cheema, A.: Rabbit-MAC: Lightweight authenticated encryption in wireless sensor networks. In: Information and Automation, 2008. ICIA 2008. International Conference on. pp. 573–577 (2008)
19. Werner-Allen, G., Lorincz, K., Welsh, M., Marcillo, O., Johnson, J., Ruiz, M., Lees, J.: Deploying a wireless sensor network on an active volcano. IEEE Internet Computing 10, 18–25 (2006)
20. Whiting, D., Housley, R., Ferguson, N.: Counter with CBC-MAC (CCM) (2002), http://csrc.nist.gov/groups/ST/toolkit/BCM/index.html

An Efficient Authentication Protocol Based on Ring-LPN

Stefan Heyse* Eike Kiltz† Vadim Lyubashesvky‡ Christof Paar§
Krzysztof Pietrzak¶

Abstract

We propose a new Hopper-Blum (HB) style authentication protocol that is provably secure based on a *ring* variant of the learning parity with noise (LPN) problem. Our protocol is secure against *active* attacks, consists of only two rounds, has small communication complexity, and has a very small footprint which makes it very applicable in scenarios that involve low-cost, resource-constrained devices.

Performance-wise, our protocol is the most efficient of the HB family of protocols and our implementation results show that it is even comparable to the standard challenge-and-response protocols based on the AES block-cipher. Our basic protocol is roughly 20 times slower than AES, but with the advantage of having 10 times smaller code size. Furthermore, if a few hundred bytes of non-volatile memory are available to allow the storage of some off-line pre-computations, then the online phase of our protocols is only twice as slow as AES.

1 Introduction

Lightweight shared-key authentication protocols, in which a tag authenticates itself to a reader, are extensively used in resource-constrained devices such as radio-frequency identification (RFID) tags or smart cards. The straight-forward approach for constructing secure authentications schemes is to use low-level symmetric primitives such as block-ciphers, e.g. AES [DR02]. In their most basic form, the protocols consist of the reader sending a short challenge c and the tag responding with $\mathsf{AES}_K(c)$, where K is the shared secret key. The protocol is secure if AES fulfills a strong, *interactive* security assumption, namely that it behaves like a strong pseudo-random function.

Authentication schemes based on AES have some very appealing features: they are extremely fast, consist of only 2 rounds, and have very small communication complexities. In certain scenarios, however, such as when low-cost and resource-constrained devices are involved, the relatively large gate-count and code size used to implement AES may pose a problem. One approach to overcome the restrictions presented by low-weight devices is to construct a low-weight block cipher (e.g. PRESENT [BKL+07]), while another approach has been to deviate entirely from block-cipher based constructions and build a *provably-secure* authentication scheme based on the hardness of some mathematical problem. In this work, we concentrate on this second approach.

*Ruhr-Universität Bochum; stefan.heyse@rub.de
†Ruhr-Universität Bochum; eike.kiltz@rub.de
‡INRIA / ENS, Paris; lyubash@di.ens.fr
§Ruhr-Universität Bochum; christof.paar@rub.de
¶IST Austria; krzpie@gmail.com

Ideally, one would like to construct a scheme that incorporates all the beneficial properties of AES-type protocols, while also acquiring the additional provable security and smaller code description characteristics. In the past decade, there have been proposals that achieved some, but not all, of these criteria. The most notable of these proposals fall into the Hopper-Blum (HB) line of protocols, which we will survey in detail below. Our proposal can be seen as a continuation of this line of research that contains all the advantages enjoyed by HB-type protocols, while at the same time, getting even closer to enjoying the benefits of AES-type schemes.

Overview of our Results. In this work we present a new symmetric authentication protocol which (i) is provably-secure against active attacks (as defined in [JW05]) based on the Ring-LPN assumption, a natural variant of the standard LPN (learning parity with noise) assumption; (ii) consists of 2 rounds; (iii) has small communication complexity (approximately 1300 bits); (iv) has efficiency comparable to AES-based challenge-response protocols (depending on the scenario), but with a much smaller code size. To demonstrate the latter we implemented the tag part of our new protocol in a setting of high practical relevance – a low-cost 8-bit microcontroller which is a typical representative of a CPU to be found on lightweight authentication tokens, and compared its performance (code size and running time) with an AES implementation on the same platform.

Previous Works. Hopper and Blum [HB00, HB01] proposed a 2-round authentication protocol that is secure against *passive* adversaries based on the hardness of the LPN problem (we remind the reader of the definition of the LPN problem in Section 1.2). The characteristic feature of this protocol is that it requires very little workload on the part of the tag and the reader. Indeed, both parties only need to compute vector inner products and additions over F_2, which makes this protocol (thereafter named HB) a good candidate for lightweight applications.

Following this initial work, Juels and Weis constructed a protocol called HB^+ [JW05] which they proved to be secure against more realistic, so called *active* attacks. Subsequently, Katz et al. [KS06a, KS06b, KSS10] provided a simpler security proof for HB^+ as well as showed that it remains secure when executed in parallel. Unlike the HB protocol, however, HB^+ requires three rounds of communication between tag and reader. From a practical aspect, 2 round authentication protocols are often advantageous over 3 round protocols. They often show a lower latency which is especially pronounced on platforms where the establishment of a communication in every directions is accompanied by a fixed initial delay. An additional drawback of both HB and HB^+ is that their communication complexity is on the order of hundreds of thousands of bits, which makes them almost entirely impractical for lightweight authentication tokens because of timing and energy constraints. (The contactless transmission of data on RFIDs or smart cards typically requires considerably more energy than the processing of the same data.)

To remedy the overwhelming communication requirement of HB^+, Gilbert et al. proposed the three-round $\mathsf{HB}^\sharp$ protocol [GRS08a]. A particularly practical instantiation of this protocol requires fewer than two thousand bits of communication, but is no longer based on the hardness of the LPN problem. Rather than using independent randomness, the $\mathsf{HB}^\sharp$ protocol utilized a Toeplitz matrix, and is thus based on a plausible assumption that the LPN problem is still hard in this particular scenario.

A feature that the $\mathsf{HB}, \mathsf{HB}^+$, and $\mathsf{HB}^\sharp$ protocols have in common is that at some point the reader sends a random string r to the tag, which then must reply with $\langle r, s\rangle + e$, the inner product of r with the secret s plus some small noise e. The recent work of Kiltz et al. [KPC+11] broke with this approach, and they were able to construct the first 2-round LPN-based authentication protocol (thereafter named HB^2) that is secure against active attacks. In their challenge-response protocol,

the reader sends some challenge bit-string c to the tag, who then answers with a noisy inner product of a random r (which the tag chooses itself) and a session-key $K(c)$, where $K(c)$ selects (depending on c) half of the bits from the secret s. Unfortunately, the HB^2 protocol still inherits the large communication requirement of HB and HB^+. Furthermore, since the session key $K(c)$ is computed using bit operations, it does not seem to be possible to securely instantiate HB^2 over structured (and hence more compact) objects such as Toeplitz matrices (as used in $\mathsf{HB}^\sharp$ [GRS08a]).

1.1 Our contributions

Protocol. In this paper we propose a variant of the HB^2 protocol from [KPC+11] which uses an "algebraic" derivation of the session key $K(c)$, thereby allowing to be instantiated over a carefully chosen ring $\mathsf{R} = \mathsf{F}_2[X]/(f)$. Our scheme is no longer based on the hardness of LPN, but rather on the hardness of a natural generalization of the problem to rings, which we call Ring-LPN(see Section 3 for the definition of the problem.) The general overview of our protocol is quite simple. Given a challenge c from the reader, the tag answers with $(r, z = r \cdot K(c) + e) \in \mathsf{R} \times \mathsf{R}$, where r is a random ring element, e is a low-weight ring element, and $K(c) = sc + s'$ is the session key that depends on the shared secret key $K = (s, s') \in \mathsf{R}^2$ and the challenge c. The reader accepts if $e' = r \cdot K(c) - z$ is a polynomial of low weight, cf. Figure 1 in Section 4. Compared to the HB and HB^+ protocols, ours has one less round and a dramatically lower communication complexity. Our protocol has essentially the same communication complexity as $\mathsf{HB}^\sharp$, but still retains the advantage of one fewer round. And compared to the two-round HB^2 protocol, ours again has the large savings in the communication complexity. Furthermore, it inherits from HB^2 the simple and tight security proof that, unlike three-round protocols, does not use rewinding.

We remark that while our protocol is provably secure against active attacks, we do not have a proof of security against man-in-the-middle ones. Still, as argued in [KSS10], security against active attacks is sufficient for many use scenarios (see also [JW05, KW05, KW06]). We would like to mention that despite man-in-the-middle attacks being outside our "security model", we think that it is still worthwhile investigating whether such attacks do in fact exist, because it presently seems that all previous man-in-the middle attacks against HB-type schemes along the lines of Gilbert et al. [GRS05] and of Ouafi et al. [OOV08] do not apply to our scheme. In Appendix A, however, we do present a man-in-the-middle attack that works in time approximately $n^{1.5} \cdot 2^{\lambda/2}$ (where n is the dimension of the secret and λ is the security parameter) when the adversary can influence on the order of $n^{1.5} \cdot 2^{\lambda/2}$ interactions between the reader and the tag. To resist this attack, one could simply double the security parameter, but we believe that even for $\lambda = 80$ (and $n > 512$, as it is currently set in our scheme) this attack is already impractical because of the extremely large number of interactions that the adversary will have to observe and modify.

Implementation. We demonstrate that our protocol is indeed practical by providing a lightweight implementation of the tag part of the protocol. (The reader is typically not run on a constrained device and therefore we do not consider its performance.) The target platform was an AVR ATmega163 [Atm] based smart card. The ATmega163 is a small 8-bit microcontroller which is a typical representative of a CPU to be found on lightweight authentication tokens. The main metrics we consider are run time and code size. We compare our results with a challenge-response protocol using an AES implementation optimized for the target platform. A major advantage of our protocol is its very small code size. The most compact implementation requires only about 460 bytes of code, which is an improvement by factor of about 10 over AES-based authentication. Given that

Table 1: Summary of implementation results

Protocol	Time (cycles)		Code size
	online	offline	(bytes)
Ours: reducible f (§5.1)	$30,000$	$82,500$	$1,356$
Ours: irreducible f (§5.2)	$21,000$	$174,000$	459
AES-based [LLS09, Tik]	$10,121$	0	$4,644$

EEPROM or FLASH memory is often one of the most precious resources on constrained devices, our protocol can be attractive in certain situations. The drawback of our protocol over AES on the target platform is an increase in clock cycles for one round of authentication. However, if we have access to a few hundred bytes of non-volatile data memory, our protocol allows precomputations which make the on-line phase only a factor two or three slower than AES. But even without precomputations, the protocol can still be executed in a few 100 msec, which will be sufficient for many real-world applications, e.g. remote keyless entry systems or authentication for financial transactions. Table 1 gives a summary of the results, see Section 5 for details.

We would like to stress at this point that our protocol is targeting lightweight tags that are equipped with (small) CPUs. For ultra constrained tokens (such as RFIDs in the price range of a few cents targeting the EPC market) which consist nowadays of a small integrated circuit, even compact AES implementations are often considered too costly. (We note that virtually all current commercially available low-end RFIDs do not have any crypto implemented.) However, tokens which use small microcontrollers are far more common, e.g., low-cost smart cards, and they do often require strong authentication. Also, it can be speculated that computational RFIDs such as the WISP [Wik] will become more common in the future, and hence software-friendly authentication methods that are highly efficient such as the protocol provided here will be needed.

1.2 LPN, Ring-LPN, and Related Problems

The security of our protocols relies on the new Ring Learning Parity with Noise (Ring-LPN) problem which is a natural extension of the standard Learning Parity with Noise (LPN) problem to rings. It can also be seen as a particular instantiation of the Ring-LWE (Learning with Errors over Rings) problem that was recently shown to have a strong connection to lattices [LPR10]. We will now briefly describe and compare these hardness assumptions, and we direct the reader to Section 3 for a formal definition of the Ring-LPN problem.

The decision versions of these problems require us to distinguish between two possible oracles to which we have black-box access. The first oracle has a randomly generated secret vector $s \in \mathsf{F}_2^n$ which it uses to produce its responses. In the LPN problem, each query to the oracle produces a uniformly random matrix[1] $A \in \mathsf{F}_2^{n\times n}$ and a vector $As + e = t \in \mathsf{F}_2^n$ where e is a vector in F_2^n each of whose entries is an independently generated Bernoulli random variable with probability of 1 being some public parameter τ between 0 and 1/2. The second oracle in the LPN problem outputs a uniformly-random matrix $A \in \mathsf{F}_2^{n\times n}$ and a uniformly random vector $t \in \mathsf{F}_2^n$.

The only difference between LPN and Ring-LPN is in the way the matrix A is generated (both by the first and second oracle). While in the LPN problem, all its entries are uniform and independent,

[1] In the more common description of the LPN problem, each query to the oracle produces one random sample in F_2^n. For comparing LPN to Ring-LPN, however, it is helpful to consider the oracle as returning a matrix of n random independent samples on each query.

in the Ring-LPN problem, only its first column is generated uniformly at random in F_2^n. The remaining n columns of A depend on the first column and the underlying ring $\mathsf{R} = \mathsf{F}_2[X]/(f(X))$. If we view the first column of A as a polynomial $r \in \mathsf{R}$, then the ith column (for $0 \leq i \leq n-1$) of A is just the vector representation of rX^i in the ring R. Thus when the oracle returns $As + e$, this corresponds to it returning the polynomial $r \cdot s + e$ where the multiplication of polynomials r and s (and the addition of e) is done in the ring R. The $\mathsf{Ring\text{-}LPN}^{\mathsf{R}}$ assumption states that it is hard to distinguish between the outputs of the first and the second oracle described above. In Section 3, we discuss how the choice of the ring R affects the security of the problem.

While the standard Learning Parity with Noise (LPN) problem has found extensive use as a cryptographic hardness assumption (e.g., [HB01, JW05, GRS08b, GRS08a, ACPS09, KSS10]), we are not aware of any constructions that employed the Ring-LPN problem. There have been some previous works that considered some relatively similar "structured" versions of LPN. The $\mathsf{HB}^\sharp$ authentication protocol of Gilbert et al. [GRS08a] made the assumption that for a random Toeplitz matrix $S \in \mathsf{F}_2^{m \times n}$, a uniformly random vector $a \in \mathsf{F}_2^n$, and a vector $e \in \mathsf{F}_2^m$ whose coefficients are distributed as Ber_τ, the output $(a, Sa + e)$ is computationally indistinguishable from (a, t) where t is uniform over F_2^m.

Another related work, as mentioned above, is the recent result of Lyubashevsky et al. [LPR10], where it is shown that solving the decisional Ring-LWE (Learning with Errors over Rings) problem is as hard as quantumly solving the worst case instances of the shortest vector problem in *ideal* lattices. The Ring-LWE problem is quite similar to Ring-LPN, with the main difference being that the ring R is defined as $\mathsf{F}_q[X]/(f(X))$ where $f(X)$ is a cyclotomic polynomial and q is a prime such that $f(X)$ splits completely into $deg(f(X))$ distinct factors over F_q.

Unfortunately, the security proof of our authentication scheme does not allow us to use a polynomial $f(X)$ that splits into low-degree factors, and so we cannot base our scheme on lattice problems. For a similar reason (see the proof of our scheme in Section 4 for more details), we cannot use samples that come from a Toeplitz matrix as in [GRS08a]. Nevertheless, we believe that the Ring-LPN assumption is very natural and will find further cryptographic applications, especially for constructions of schemes for low-cost devices.

2 Definitions

2.1 Rings and Polynomials

For a polynomial $f(X)$ over F_2, we will often omit the indeterminate X and simply write f. The degree of f is denoted by $deg(f)$. For two polynomials a, f in $\mathsf{F}_2[X]$, $a \bmod f$ is defined to be the unique polynomial r of degree less than $deg(f)$ such that $a = fg + r$ for some polynomial $g \in \mathsf{F}_2[X]$. The elements of the ring $\mathsf{F}_2[X]/(f)$ will be represented by polynomials in $\mathsf{F}_2[X]$ of maximum degree $deg(f) - 1$. In this paper, we will only be considering rings $\mathsf{R} = \mathsf{F}_2[X]/(f)$ where the polynomial f factors into *distinct* irreducible factors over F_2. For an element a in the ring $\mathsf{F}_2[X]/(f)$, we will denote by $\widehat{a}$, the CRT (Chinese Remainder Theorem) representation of a with respect to the factors of f. In other words, if $f = f_1 \ldots f_m$ where all f_i are irreducible, then

$$\widehat{a} \doteq (a \bmod f_1, \ldots, a \bmod f_m).$$

If f is itself an irreducible polynomial, then $\widehat{a} = a$. Note that an element $\widehat{a} \in \mathsf{R}$ has a multiplicative inverse iff, for all $1 \leq i \leq m$, $a \neq 0 \bmod f_i$. We denote by R^* the set of elements in R that have a multiplicative inverse.

2.2 Distributions

For a distribution D over some domain, we write $r \stackrel{\$}{\leftarrow} D$ to denote that r is chosen according to the distribution D. For a domain Y, we write $U(Y)$ to denote the uniform distribution over Y. Let Ber_τ be the Bernoulli distribution over F_2 with parameter (bias) $\tau \in \ ]0, 1/2[$ (i.e., $\Pr[x = 1] = \tau$ if $x \leftarrow \mathsf{Ber}_\tau$). For a polynomial ring $\mathsf{R} = \mathsf{F}_2[X]/(f)$, the distribution $\mathsf{Ber}_\tau^{\mathsf{R}}$ denotes the distribution over the polynomials of R, where each of the $deg(f)$ coefficients of the polynomial is drawn independently from Ber_τ. For a ring R and a polynomial $s \in \mathsf{R}$, we write $\Lambda_\tau^{\mathsf{R},s}$ to be the distribution over $\mathsf{R} \times \mathsf{R}$ whose samples are obtained by choosing a polynomial $r \stackrel{\$}{\leftarrow} U(\mathsf{R})$ and another polynomial $e \stackrel{\$}{\leftarrow} \mathsf{Ber}_\tau^{\mathsf{R}}$, and outputting $(r, rs + e)$.

2.3 Authentication Protocols

An authentication protocol Π is an interactive protocol executed between a Tag $\mathcal{T}$ and a reader $\mathcal{R}$, both PPT algorithms. Both hold a secret x (generated using a key-generation algorithm KG executed on the security parameter λ in unary) that has been shared in an initial phase. After the execution of the authentication protocol, $\mathcal{R}$ outputs either accept or reject. We say that the protocol has completeness error ε_c if for all $\lambda \in \mathbb{N}$, all secret keys x generated by $\mathsf{KG}(1^\lambda)$, the honestly executed protocol returns reject with probability at most ε_c. We now define different security notions of an authentication protocol.

Passive attacks. An authentication protocol is secure against *passive* attacks, if there exists no PPT adversary $\mathcal{A}$ that can make the reader $\mathcal{R}$ return accept with non-negligible probability after (passively) observing any number of interactions between reader and tag.

Active attacks. A stronger notion for authentication protocols is security against *active* attacks. Here the adversary $\mathcal{A}$ runs in two stages. First, she can interact with the honest tag a polynomial number of times (with concurrent executions allowed). In the second phase $\mathcal{A}$ interacts with the reader only, and wins if the reader returns accept. Here we only give the adversary one shot to convince the verifier.[2] An authentication protocol is (t, q, ε)-*secure against active adversaries* if every PPT $\mathcal{A}$, running in time at most t and making q queries to the honest reader, has probability at most ε to win the above game.

3 Ring-LPN and its Hardness

The decisional $\mathsf{Ring\text{-}LPN}^{\mathsf{R}}$ (Ring Learning Parity with Noise in ring R) assumption, formally defined below, states that it is hard to distinguish uniformly random samples in $\mathsf{R} \times \mathsf{R}$ from those sampled from $\Lambda_\tau^{\mathsf{R},s}$ for a uniformly chosen $s \in \mathsf{R}$.

Definition 3.1 ($\mathsf{Ring\text{-}LPN}^{\mathsf{R}}$). *The (decisional)* $\mathsf{Ring\text{-}LPN}_\tau^{\mathsf{R}}$ *problem is* (t, Q, ε)*-hard if for every distinguisher* $\mathcal{D}$ *running in time* t *and making* Q *queries,*

$$\left| \Pr\left[s \stackrel{\$}{\leftarrow} \mathsf{R} \ : \ \mathcal{D}^{\Lambda_\tau^{\mathsf{R},s}} = 1 \right] - \Pr\left[\ \mathcal{D}^{U(\mathsf{R}\times\mathsf{R})} = 1 \right] \right| \leq \varepsilon.$$

[2] By using a hybrid argument one can show that this implies security even if the adversary can interact in $k \geq 1$ independent instances concurrently (and wins if the verifier accepts in at least one instance). The use of the hybrid argument looses a factor of k in the security reduction.

3.1 Hardness of LPN and Ring-LPN

One can attempt to solve Ring-LPN using standard algorithms for LPN, or by specialized algorithms that possibly take advantage of Ring-LPN's additional structure. Some work towards constructing the latter type of algorithm has recently been done by Hanrot et al. [HLPS11], who show that in certain cases, the algebraic structure of the Ring-LPN and Ring-LWE problems makes them vulnerable to certain attacks. These attacks essentially utilize a particular relationship between the factorization of the polynomial $f(X)$ and the distribution of the noise.

3.1.1 Ring-LPN with an irreducible $f(X)$

When $f(X)$ is irreducible over F_2, the ring $\mathsf{F}_2[X]/(f)$ is a field. For such rings, the algorithm of Hanrot et al. does not apply, and we do not know of any other algorithm that takes advantage of the added algebraic structure of this particular Ring-LPN instance. Thus to the best of our knowledge, the most efficient algorithms for solving this problem are the same ones that are used to solve LPN, which we will now very briefly recount.

The computational complexity of the LPN problem depends on the length of the secret n and the noise distribution Ber_τ. Intuitively, the larger the n and the closer τ is to $1/2$, the harder the problem becomes. Usually the LPN problem is considered for constant values of τ somewhere between 0.05 and 0.25. For such constant τ, the fastest asymptotic algorithm for the LPN problem, due to Blum et al. [BKW03], takes time $2^{\Omega(n/\log n)}$ and requires approximately $2^{\Omega(n/\log n)}$ samples from the LPN oracle. If one has access to fewer samples, then the algorithm will perform somewhat worse. For example, if one limits the number of samples to only polynomially-many, then the algorithm has an asymptotic complexity of $2^{\Omega(n/\log\log n)}$ [Lyu05]. In our scenario, the number of samples available to the adversary is limited to n times the number of executions of the authentication protocol, and so it is reasonable to assume that the adversary will be somewhat limited in the number of samples he is able to obtain (perhaps at most 2^{40} samples), which should make our protocols harder to break than solving the Ring-LPN problem. Levieil and Fouque [LF06] made some optimizations to the algorithm of Blum et al. and analyzed its precise complexity. To the best of our knowledge, their algorithm is currently the most efficient one and we will refer to their results when analyzing the security of our instantiations.

In Section 5, we base our scheme on the hardness of the $\mathsf{Ring\text{-}LPN}^{\mathsf{R}}$ problem where $\mathsf{R} = \mathsf{F}_2[X]/(X^{532} + X + 1)$ and $\tau = 1/8$. According to the analysis of [LF06], an LPN problem of dimension 512 with $\tau = 1/8$ would require 2^{77} memory (and thus at least that much time) to solve when given access to approximately as many samples (see [LF06, Section 5.1]). Since our dimension is somewhat larger and the number of samples will be limited in practice, it is reasonable to assume that this instantiation has 80-bit security.

3.1.2 Ring-LPN with a reducible $f(X)$

For efficiency purposes, it is sometimes useful to consider using a polynomial $f(X)$ that is not irreducible over F_2. This will allow us to use the CRT representation of the elements of $\mathsf{F}_2[X]/(f)$ to perform multiplications, which in practice turns out to be more efficient. Ideally, we would like the polynomial f to split into as many small-degree polynomials f_i as possible, but there are some constraints that are placed on the factorization of f both by the security proof, and the possible weaknesses that a splittable polynomial introduces into the Ring-LPN problem.

If the polynomial f splits into $f = \prod_{i=1}^{m} f_i$, then it may be possible to try and solve the Ring-LPN problem modulo some f_i rather than modulo f. Since the degree of f_i is smaller than the degree of f, the resulting Ring-LPN problem may end up being easier. In particular, when we receive a sample $(r, rs + e)$ from the distribution $\Lambda_\tau^{\mathsf{R},s}$, we can rewrite it in CRT form as

$$(\widehat{r}, \widehat{rs + e}) = ((r \bmod f_1, rs + e \bmod f_1), \ldots, \\ (r \bmod f_m, rs + e \bmod f_m)),$$

and thus for every f_i, we have a sample

$$(r \bmod f_i, (r \bmod f_i)(s \bmod f_i) + e \bmod f_i),$$

where all the operations are in the ring (or field) $\mathsf{F}_2[X]/(f_i)$. Thus solving the (decision) Ring-LPN problem in $\mathsf{F}_2[X]/(f)$ reduces to solving the problem in $\mathsf{F}_2[X]/(f_i)$. The latter problem is in a smaller dimension, since $deg(s) > deg(s \bmod f_i)$, but the error distribution of $(e \bmod f_i)$ is quite different than that of e. While each coefficient of e is distributed independently as Ber_τ, each coefficient of $(e \bmod f_i)$ is distributed as the distribution of a sum of certain coefficients of e, and therefore the new error is larger.[3] Exactly which coefficients of e, and more importantly, how many of them, combine to form every particular coefficient of e' depends on the polynomial f_i. For example, if

$$f(X) = (X^3 + X + 1)(X^3 + X^2 + 1)$$

and $e = \sum_{i=0}^{5} e_i X^i$, then,

$$e' = e \bmod (X^3 + X + 1) = (e_0 + e_3 + e_5) + (e_1 + e_3 + e_4 + e_5)X + (e_2 + e_4 + e_5)X^2,$$

and thus every coefficient of the error e' is comprised of at least 3 coefficients of the error vector e, and thus $\tau' > \frac{1}{2} - \frac{(1-2\tau)^3}{2}$.

In our instantiation of the scheme with a reducible $f(X)$ in Section 5, we used the $f(X)$ such that it factors into f_i's that make the operations in CRT form relatively fast, while making sure that the resulting Ring-LPN problem modulo each f_i is still around 2^{80}-hard.

4 Authentication Protocol

In this section we describe our new 2-round authentication protocol and prove its active security under the hardness of the Ring-LPN problem. Detailed implementation details will be given in Section 5.

4.1 The Protocol

Our authentication protocol is defined over the ring $\mathsf{R} = \mathsf{F}_2[X]/(f)$ and involves a "suitable" mapping $\pi : \{0,1\}^\lambda \rightarrow \mathsf{R}$. We call π *suitable* for ring R if for all $c, c' \in \{0,1\}^\lambda$, $\pi(c) - \pi(c') \in \mathsf{R} \setminus \mathsf{R}^*$ iff $c = c'$. We will discuss the necessity and existence of such mappings after the proof of Theorem 4.1

[3] If we have k elements $e_1, \ldots, e_k \overset{\$}{\leftarrow} \mathsf{Ber}_\tau$, then the element $e' = e_1 + \ldots + e_k$ is distributed as $\mathsf{Ber}_{\tau'}$ where $\tau' = \frac{1}{2} - \frac{(1-2\tau)^k}{2}$.

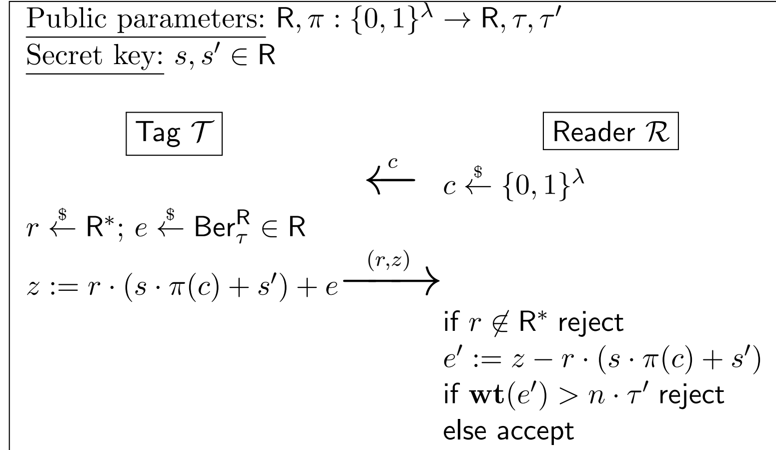


Figure 1: Two-round authentication protocol with active security from the $\mathsf{Ring\text{-}LPN}^{\mathsf{R}}$ assumption.

- Public parameters. The authentication protocol has the following public parameters, where τ, τ' are constants and n depend on the security parameter λ.

R, n	ring $\mathsf{R} = \mathsf{F}_2[X]/(f)$, $\deg(f) = n$
$\pi : \{0,1\}^\lambda \to \mathsf{R}$	mapping
$\tau \in \{0, \dots 1/2\}$	parameter of Bernoulli distribution
$\tau' \in \{\tau, \dots 1/2\}$	acceptance threshold

- Key Generation. Algorithm $\mathsf{KG}(1^\lambda)$ samples $s, s' \xleftarrow{\$} \mathsf{R}$ and returns s, s' as the secret key.
- Authentication Protocol. The Reader $\mathcal{R}$ and the Tag $\mathcal{T}$ share secret value $s, s' \in \mathsf{R}$. To be authenticated by a Reader, the Tag and the Reader execute the authentication protocol from Figure 1.

4.2 Analysis

For our analysis we define for $x, y \in]0,1[$ the following constant:

$$c(x,y) := \left(\frac{x}{y}\right)^x \left(\frac{1-x}{1-y}\right)^{1-x}.$$

We now state that our protocol is secure against active adversaries. Recall that active adversaries can arbitrarily interact with a Tag oracle in the first phase and tries to impersonate the Reader in the 2nd phase.

Theorem 4.1. *If ring mapping π is suitable for ring R and the $\mathsf{Ring\text{-}LPN}_{\mathsf{R}}$ problem is (t, q, ε)-hard then the authentication protocol from Figure 1 is (t', q, ε')-secure against active adversaries, where*

$$t' = t - q \cdot \exp(R) \qquad \varepsilon' = \varepsilon + q \cdot 2^{-\lambda} + c(\tau', 1/2)^{-n} \tag{4.1}$$

and $\exp(R)$ is the time to perform $O(1)$ exponentiations in R. Furthermore, the protocol has completeness error $\varepsilon_{\mathrm{c}}(\tau, \tau', n) \approx c(\tau', \tau)^{-n}$.

Proof. The completeness error $\varepsilon_c(\tau,\tau',n)$ is (an upper bound on) the probability that an honestly generated Tag gets rejected. In our protocol this is exactly the case when the error e has weight $\geq n\cdot\tau'$, i.e.

$$\varepsilon_c(\tau,\tau',n) = \Pr[\mathbf{wt}(e) > n\cdot\tau' \ : \ e \stackrel{\$}{\leftarrow} \mathsf{Ber}_\tau^{\mathsf{R}}]$$

Levieil and Fouque [LF06] show that one can approximate this probability as $\varepsilon_c \approx c(\tau',\tau)^{-n}$.

To prove the security of the protocol against acitve attacks we proceed in sequences of games. Game_0 is the security experiment describing an active attack on our scheme by an adversary $\mathcal{A}$ making q queries and running in time t', i.e.

- Sample the secret key $s, s' \stackrel{\$}{\leftarrow} \mathsf{R}$.
- (1st phase of active attack) $\mathcal{A}$ queries the tag $\mathcal{T}$ on $c \in \{0,1\}^\lambda$ and receives (r,z) computed as illustrated in Figure 1.
- (2nd phase of active attack) $\mathcal{A}$ gets a random challange $c^* \stackrel{\$}{\leftarrow} \{0,1\}^\lambda$ and outputs (r,z). $\mathcal{A}$ wins if the reader $\mathcal{R}$ accepts, i.e. $\mathbf{wt}(z - r\cdot(s\cdot\pi(c^*)+s')) \leq n\cdot\tau'$.

By definition we have $\Pr[\mathcal{A} \text{ wins in } \mathsf{Game}_0] \leq \varepsilon'$.

Game_1 is as Game_0, except that all the values (r,z) returned by the Tag oracle in the first phase (in return to a query $c \in \{0,1\}^\lambda$) are uniform random elements $(r,z) \in \mathsf{R}^2$. We now show that if $\mathcal{A}$ is successful against Game_0, then it will also be successful against Game_1.

Claim 4.2. $|\Pr[\mathcal{A} \textit{ wins in } \mathsf{Game}_1] - \Pr[\mathcal{A} \textit{ wins in } \mathsf{Game}_0]| \leq \varepsilon + q\cdot 2^{-\lambda}$

To prove this claim, we construct an adversary $\mathcal{D}$ (distinguisher) against the Ring-LPN problem which runs in time $t = t' + \exp(\mathsf{R})$ and has advantage

$$\varepsilon \geq |\Pr[\mathcal{A} \text{ wins in } \mathsf{Game}_1] - \Pr[\mathcal{A} \text{ wins in } \mathsf{Game}_0]| - q\cdot 2^{-\lambda}$$

$\mathcal{D}$ has access to a Ring-LPN oracle $\mathcal{O}$ and has to distinguish between $\mathcal{O} = \Lambda_\tau^{\mathsf{R},s}$ for some secret $s \in \mathsf{R}$ and $\mathcal{O} = U(\mathsf{R}\times\mathsf{R})$.

- $\mathcal{D}$ picks a random challenge $c^* \stackrel{\$}{\leftarrow} \{0,1\}^\lambda$ and $a \stackrel{\$}{\leftarrow} \mathsf{R}$. Next, it runs $\mathcal{A}$ and simulates its view with the unknown secret s, s', where $s \in \mathsf{R}$ comes from the oracle $\mathcal{O}$ and s' is implicitly defined as $s' := -\pi(c^*)\cdot s + a \in \mathsf{R}$.
- In the 1st phase, $\mathcal{A}$ can make q (polynomial many) queries to the Tag oracle. On query $c \in \{0,1\}^\lambda$ to the Tag oracle, $\mathcal{D}$ proceeds as follows. If $\pi(c) - \pi(c^*) \notin \mathsf{R}^*$, then abort. Otherwise, $\mathcal{D}$ queries its oracle $\mathcal{O}()$ to obtain $(r',z') \in \mathsf{R}^2$. Finally, $\mathcal{D}$ returns (r,z) to $\mathcal{A}$, where
$$r := r'\cdot(\pi(c)-\pi(c^*))^{-1}, \quad z := z' + ra. \tag{4.2}$$
- In the 2nd phase, $\mathcal{D}$ uses $c^* \in \{0,1\}^\lambda$ to challenge $\mathcal{A}$. On answer (r,z), $\mathcal{D}$ returns 0 to the Ring-LPN game if $\mathbf{wt}(z - r\cdot a) > n\cdot\tau'$ or $r \notin \mathsf{R}^*$, and 1 otherwise. Note that $s\pi(c^*) + s' = (\pi(c^*) - \pi(c^*))s + a = a$ and hence the above check correctly simulates the output of a reader with the simulated secret s, s'.

Note that the running time of $\mathcal{D}$ is that of $\mathcal{A}$ plus $O(q)$ exponentiations in R.

Let bad be the event that for at least one query c made by $\mathcal{A}$ to the Tag oracle, we have that $\pi(c) - \pi(c^*) \notin \mathsf{R}^*$. Since c^* is uniform random in R and hidden from $\mathcal{A}$'s view in the first phase we have by the union bound over the q queries

$$\begin{aligned} \Pr[\mathsf{bad}] &\leq q \cdot \Pr_{c^* \in \{0,1\}^\lambda}[\pi(c) - \pi(c^*) \in \mathsf{R} \setminus \mathsf{R}^*] \\ &= q \cdot 2^{-\lambda}. \end{aligned} \tag{4.3}$$

The latter inequality holds because π is suitable for R.

Let us now assume bad does not happen. If $\mathcal{O} = \Lambda_\tau^{\mathsf{R},s}$ is the real oracle (i.e., it returns (r', z') with $z' = r's + e$) then by the definition of (r, z) from (4.2),

$$z = (r's + e) + ra = r(\pi(c) - \pi(c^*) + a)s + e = r(s\pi(c) + s') + e.$$

Hence the simulation perfectly simulates $\mathcal{A}$'s view in Game_0. If $\mathcal{O} = U(\mathsf{R} \times \mathsf{R})$ is the random oracle then (r, z) are uniformly distributed, as in Game_1. That concludes the proof of Claim 4.2.

We next upper bound the probability that $\mathcal{A}$ can be successful in Game_1. This bound will be information theoretic and even holds if $\mathcal{A}$ is computationally unbounded and can make an unbounded number of queries in the 1st phase. To this end we introduce the minimal soundness error, $\varepsilon_{\mathrm{ms}}$, which is an upper bound on the probability that a tag (r, z) chosen independently of the secert key is valid, i.e.

$$\varepsilon_{\mathrm{ms}}(\tau', n) := \max_{(z,r) \in \mathsf{R} \times \mathsf{R}^*} \Pr_{s,s' \xleftarrow{\$} \mathsf{R}} [\mathbf{wt}(\underbrace{z - r \cdot (s \cdot \pi(c^*) + s')}_{e'}) \leq n\tau']$$

As $r \in \mathsf{R}^*$ and $s' \in \mathsf{R}$ is uniform, also $e' = z - r \cdot (s \cdot \pi(c^*) + s'$ is uniform, thus $\varepsilon_{\mathrm{ms}}$ is simply

$$\varepsilon_{\mathrm{ms}}(\tau', n) := \Pr_{e' \xleftarrow{\$} \mathsf{R}} [\mathbf{wt}(e') \leq n\tau']$$

Again, it was shown in [LF06] that this probability can be approximated as

$$\varepsilon_{\mathrm{ms}}(\tau', n) \approx c(\tau', 1/2)^{-n}. \tag{4.4}$$

Clearly, $\varepsilon_{\mathrm{ms}}$ is a trivial lower bound on the advantage of $\mathcal{A}$ in forging a valid tag, by the following claim in Game_1 one cannot do any better than this.

Claim 4.3. $\Pr[\mathcal{A}$ *wins in* $\mathsf{Game}_1] = \varepsilon_{\mathrm{ms}}(\tau', n)$

To see that this claim holds one must just observe that the answers $\mathcal{A}$ gets in the first phase of the active attack in Game_1 are independent of the secret s, s'. Hence $\mathcal{A}$'s advantage is $\varepsilon_{\mathrm{ms}}(\tau', n)$ by definition.

Claims 4.2 and 4.3 imply (4.1) and conclude the proof of Theorem 4.1. □

We require the mapping $\pi : \{0,1\}^\lambda \to \mathsf{R}$ used in the protocol to be *suitable* for R, i.e. for all $c, c' \in \{0,1\}^\lambda$, $\pi(c) - \pi(c') \in \mathsf{R} \setminus \mathsf{R}^*$ iff $c = c'$. In Section 5 we describe efficient suitable maps for any $\mathsf{R} = \mathsf{F}_2[X]/(f)$ where f has no factor of degree $\leq \lambda$. This condition is necessary, as no suitable mapping exists if f has a factor f_i of degree $\leq \lambda$: in this case, by the pigeonhole principle, there exist distinct $c, c' \in \{0,1\}^\lambda$ such that $\pi(c) = \pi(c') \mod f_i$, and thus $\pi(c) - \pi(c') \in \mathsf{R} \setminus \mathsf{R}^*$.

We stress that for our security proof we need π to be suitable for R, since otherwise (4.3) is no longer guaranteed to hold. It is an interesting question if this is inherent, or if the security of our protocol can be reduced to the $\mathsf{Ring\text{-}LPN}^{\mathsf{R}}$ problem for arbitrary rings $\mathsf{R} = \mathsf{F}_2[X]/(f)$, or even $\mathsf{R} = \mathsf{F}_q[X]/(f)$ (This is interesting since, if f has factors of degree $\ll \lambda$, the protocol could be implemented more efficiently and even become based on the worst-case hardness of lattice problems). Similarly, it is unclear how to prove security of our protocol instantiated with Toeplitz matrices.

5 Implementation

There are two objectives that we pursue with the implementation of our protocol. First, we will show that the protocol is in fact practical with concrete parameters, even on extremely constrained CPUs. Second, we investigate possible application scenarios where the protocol might have additional advantages. From a practical point of view, we are particularly interested in comparing our protocol to classical symmetric challenge-response schemes employing AES. Possible advantages of the protocol at hand are (i) the security properties and (ii) improved implementation properties. With respect to the former aspect, our protocol has the obvious advantage of being provably secure under a reasonable and static hardness assumption. Even though AES is arguably the most trusted symmetric cipher, it is "merely" computationally secure with respect to known attacks.

In order to investigate implementation properties, constrained microprocessors are particularly relevant. We chose an 8-bit AVR ATmega163 [Atm] based smartcard, which is widely used in myriads of embedded applications. It can be viewed as a typical representative of a CPU used in tokens that are in need for an authentication protocol, e.g., computational RFID tags or (contactless) smart cards. The main metrics we consider for the implementation are run-time and code size. We note at this point that in many lightweight crypto applications, code size is the most precious resource once the run-time constraints are fulfilled. This is due to the fact that EEPROM or flash memory is often heavily constrained. For instance, the WISP, a computational RFID tag, has only 8 kBytes of program memory [Wik, MSP].

We implemented two variants of the protocol described in Section 4. The first variant uses a ring $\mathsf{R} = \mathsf{F}_2[X]/(f)$, where f splits into five irreducible polynomials; the second variant uses a field, i.e., f is irreducible. For both implementations, we chose parameters which provide a security level of $\lambda = 80$ bits, i.e., the parameters are chosen such that ε' in (4.1) is bounded by 2^{-80} and the completeness ε_c is bounded by 2^{-40}. This security level is appropriate for the lightweight applications which we are targeting.

5.1 Implementation with a Reducible Polynomial

From an implementation standpoint, the case of reducible polynomial is interesting since one can take advantage of arithmetic based on the Chinese Remainder Theorem.

Parameters. To define the ring $\mathsf{R} = \mathsf{F}_2[X]/(f)$, we chose the reducible polynomial f to be the product of the $m = 5$ irreducible pentanomials specified by the following powers with non-zero coefficients: $(127, 8, 7, 3, 0)$, $(126, 9, 6, 5, 0)$, $(125, 9, 7, 4, 0)$, $(122, 7, 4, 3, 0)$, $(121, 8, 5, 1, 0)$[4]. Hence f is a polynomial of degree $n = 621$. We chose $\tau = 1/6$ and $\tau' = .29$ to obtain minimal soundness error $\varepsilon_{ms} \approx c(\tau', 1/2)^{-n} \leq 2^{-82}$ and completeness error $\varepsilon_c \leq 2^{-42}$. From the discussion of Section

[4] $(127, 8, 7, 3, 0)$ refers to the polynomial $X^{127} + X^8 + X^7 + X^3 + 1$.

3 the best known attack on $\mathsf{Ring\text{-}LPN}_\tau^{\mathsf{R}}$ with the above parameters has complexity $> 2^{80}$. The mapping $\pi : \{0,1\}^{80} \to \mathsf{R}$ is defined as follows. On input $c \in \{0,1\}^{80}$, for each $1 \leq i \leq 5$, pad $c \in \{0,1\}^{80}$ with $\deg(f_i) - 80$ zeros and view the result as coefficients of an element $v_i \in \mathsf{F}_2[X]/(f_i)$. This defines $\pi(c) = (v_1, \ldots, v_5)$ in CRT representation. Note that, for fixed $c, c^* \in \{0,1\}^{80}$, we have that $\pi(c) - \pi(c^*) \in \mathsf{R} \setminus \mathsf{R}^*$ iff $c = c^*$ and hence π is *suitable* for R.

Implementation Details. The main operations are multiplications and additions of polynomials that are represented by 16 bytes. We view the CRT-based multiplication in three stages. In the first stage, the operands are reduced modulo each of the five irreducible polynomials. This part has a low computational complexity. Note that only the error e has to be chosen in the ring and afterwards transformed to CRT representation. It is possible to save the secret key (s, s') and to generate r directly in the CRT representation. This is not possible for e because e has to come from $\mathsf{Ber}_\tau^{\mathsf{R}}$. In the second stage, one multiplication in each of the finite fields defined by the five pentanomials has to be performed. We used the right-to-left comb multiplication algorithm from [HMV03]. For the multiplication with $\pi(c)$ we exploit the fact that only the first 80 coefficients can be non-zero. Hence we wrote one function for *normal* multiplication and one for *sparse* multiplication. The latter is more than twice as fast as the former. The subsequent reduction takes care of the special properties of the pentanomials, thus code reuse is not possible for the different fields. The third stage, constructing the product polynomial in the ring, is shifted to the prover (RFID reader) which normally has more computational power than the tag $\mathcal{T}$. Hence the response (r, z) is sent in CRT form to the reader. If non-volatile storage — in our case we need $2 \cdot 5 \cdot 16 = 160$ bytes — is available we can heavily reduce the response time of the tag. At an arbitrary point in time, choose e and r according to their distribution and precompute $tmp_1 = r \cdot s$ and $tmp_2 = r \cdot s' + e$. When a challenge c is received afterwards, tag $\mathcal{T}$ only has to compute $z = tmp_1 \cdot \pi(c) + tmp_2$. Because $\pi(c)$ is sparse, the tag can use the *sparse* multiplication and response very quickly. The results of the implementation are shown in Table 2 in Section 5.3. Note that all multiplication timings given already include the necessary reductions and addition of a value according to Figure 1.

5.2 Implementation with an Irreducible Polynomial

Parameters. To define the field $\mathsf{F} = \mathsf{F}_2[X]/(f)$, we chose the irreducible trinomial $f(X) = X^{532} + X + 1$ of degree $n = 532$. We chose $\tau = 1/8$ and $\tau' = .27$ to obtain minimal soundness error $\varepsilon_{\mathrm{ms}} \approx c(\tau', 1/2)^{-n} \leq 2^{-80}$ and completeness error $\varepsilon_{\mathrm{c}} \approx 2^{-55}$. From the discussion in Section 3 the best known attack on $\mathsf{Ring\text{-}LPN}_\tau^{\mathsf{F}}$ with the above parameters has complexity $> 2^{80}$. The mapping $\pi : \{0,1\}^{80} \to \mathsf{F}$ is defined as follows. View $c \in \{0,1\}^{80}$ as $c = (c_1, \ldots, c_{16})$ where c_i is a number between 1 and 32. Define the coefficients of the polynomial $v = \pi(c) \in \mathsf{F}$ as zero except all positions i of the form $i = 16 \cdot (j-1) + c_j$, for some $j = 1, \ldots, 16$. Hence $\pi(c)$ is sparse, i.e., it has exactly 16 non-zero coefficients. Since π is injective and F is a field, the mapping π is suitable for F.

Implementation Details. The main operation for the protocol is now a 67-byte multiplication. Again we used the right-to-left comb multiplication algorithm from [HMV03] and an optimized reduction algorithm. Like in the reducible case, the tag can do similar precomputations if $2 \cdot 67 = 134$ bytes non-volatile storage are available. Because of the special type of the mapping $v = \pi(c)$, the gain of the *sparse* multiplication is even larger than in the reducible case. Here we are a factor of 7 faster, making the response time with precomputations faster, although the field is larger. The results are shown in Table 3 in Section 5.3.

5.3 Implementation Results

All results presented in this section consider only the clock cycles of the actual arithmetic functions. The communication overhead and the generation of random bytes is excluded because they occur in every authentication scheme, independent of the underlying cryptographic functions. The time for building e from $\mathsf{Ber}_{\tau}^{\mathsf{R}}$ out of the random bytes and converting it to CRT form is included in *Overhead*. Table 2 and Table 3 shows the results for the ring based and field based variant, respectively.

Table 2: Results for the ring based variant w/o precomputation

Aspect	**time in cycles**	**code size in bytes**
Overhead	$17,500$	264
Mul	$5 \times 13,000$	164
sparse Mul	$5 \times 6,000$	170
total	$112,500$	1356

The overall code size is not the sum of the other values because, as mentioned before, the same multiplication code is used for all *normal* and *sparse* multiplications, respectively, while the reduction code is different for every field ($\approx$ 134 byte each). The same code for reduction is used independently of the type of the multiplication for the same field. If precomputation is acceptable, the tag can answer the challenge after approximately $30,000$ clock cycles, which corresponds to a 15 msec if the CPU is clocked at 2 MHz.

Table 3: Results for the field based variant w/o precomputation

Aspect	**time in cycles**	**code size in bytes**
Overhead	$3,000$	150
Mul	$150,000$	161
sparse Mul	$21,000$	148
total	$174,000$	459

For the field-based protocol, the overall performance is slower due to the large operands used in the multiplication routine. But due to the special mapping $v = \pi(c)$, here the tag can do a sparse multiplications in only $21,000$ clocks cycles. This allows the tag to respond in 10.5 msec at 2 MHz clock rate if non-volatile storage is available.

As mentioned in the introduction, we want to compare our scheme with a conventional challenge-response authentication protocol based on AES. The tag's main operation in this case is one AES encryption. The implementation in [LLS09] states $8,980$ clock cycles for one encryption on a similar platform, but unfortunately no code size is given; [Tik] reports 10121 cycles per encryption and a code size of 4644 bytes.[5] In comparison with these highly optimized AES implementations, our scheme is around eleven times slower when using the ring based variant without precomputations. If non-volatile storage allows precomputations, the ring based variant is only three times slower than AES. But the code size is by a factor of two to three smaller, making it attractive for Flash

[5] An internet source [Poe] claims to encrypt in 3126 cycles with code size of 3098 bytes but since this is unpublished material we do not consider it in our comparison.

constrained devices. The field based variant without precomputations is 17 to 19 times slower than AES, but with precompuations it is only twice as slow as AES, while only consuming one tenths of the code size. From a practical point of view, it is important to note that even our slowest implementation is executed in less than 100 msec if the CPU is clocked at 2 MHz. This response time is sufficient in many application scenarios. (For authentications involving humans, a delay of 1 sec is often considered acceptable.)

The performance drawback compared to AES is not surprising, but it is considerably less dramatic compared to asymmetric schemes like RSA or ECC [GPW+04]. But exploiting the special structure of the multiplications in our scheme and using only a small amount of non-volatile data memory provides a response time in the same order of magnitude as AES, while keeping the code size much smaller.

6 Conclusions and open Problems

We proposed a new HB-style authentication protocol with provable security against active attacks based on the Ring-LPN assumption, consisting of only two rounds, and having small communication complexity. Furthermore, our implementations on an 8-bit AVR ATmega163 based smartcard demonstrated that it has very small code size and its efficiency can be of the same order as traditional AES-based authentication protocols. Overall, we think that its features make it very applicable in scenarios that involve low-cost, resource-constrained devices.

A number of open problems remain. Our protocol cannot be proved secure against man-in-the-middle attacks. It is possible to apply the techniques from [KPC+11] to secure it against such attacks, but the resulting protocol would lose its practical appeal in terms of code size and performance. Finding a truly practical authentication protocol, provably secure against man-in-the-middle attacks from the Ring-LPN assumption (or something comparable) remains a challenging open problem.

We believe that the Ring-LPN assumption is very natural and will find further cryptographic applications, especially for constructions of schemes for low-cost devices. In particular, we think that if the HB line of research is to lead to a practical protocol in the future, then the security of this protocol will be based on a hardness assumption with some "extra algebraic structure", such as Ring-LPN in this work, or LPN with Toeplitz matrices in the work of Gilbert et al. [GRS08a]. More research, however, needs to be done on understanding these problems and their computational complexity. In terms of Ring-LPN, it would be particularly interesting to find out whether there exists an equivalence between the decision and the search versions of the problem similar to the reductions that exist for LPN [BFKL93, Reg09, KS06a] and Ring-LWE [LPR10].

7 Acknowledgements.

We would like to thank the anonymous referees for very useful comments, and in particular for the suggestion that the scheme is vulnerable to a man-in-the-middle attack whenever an adversary observes two reader challenges that are the same. We hope that the attack we described in Appendix A corresponds to what the reviewer had in mind.

References

[ACPS09] Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai, *Fast cryptographic primitives and circular-secure encryption based on hard learning problems*, CRYPTO 2009 (Shai Halevi, ed.), LNCS, vol. 5677, Springer, August 2009, pp. 595–618.

[Atm] Atmel, *ATmega163 datasheet*, "www.atmel.com/atmel/acrobat/doc1142.pdf".

[BFKL93] Avrim Blum, Merrick L. Furst, Michael J. Kearns, and Richard J. Lipton, *Cryptographic primitives based on hard learning problems*, CRYPTO, 1993, pp. 278–291.

[BKL+07] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe, *PRESENT: An ultra-lightweight block cipher*, CHES 2007 (Pascal Paillier and Ingrid Verbauwhede, eds.), LNCS, vol. 4727, Springer, September 2007, pp. 450–466.

[BKW03] Avrim Blum, Adam Kalai, and Hal Wasserman, *Noise-tolerant learning, the parity problem, and the statistical query model*, J. ACM **50** (2003), no. 4, 506–519.

[DR02] Joan Daemen and Vincent Rijmen, *The design of rijndael: AES - the advanced encryption standard*, Springer, 2002.

[GPW+04] Nils Gura, Arun Patel, Arvinderpal W, Hans Eberle, and Sheueling Chang Shantz, *Comparing elliptic curve cryptography and RSA on 8-bit CPUs*, Cryptographic Hardware and Embedded Systems - CHES 2004, 2004, pp. 119–132.

[GRS05] Henri Gilbert, Matt Robshaw, and Herve Sibert, *An active attack against HB+ – a provably secure lightweight authentication protocol*, Cryptology ePrint Archive, Report 2005/237, 2005, http://eprint.iacr.org/.

[GRS08a] Henri Gilbert, Matthew J. B. Robshaw, and Yannick Seurin, *$HB^\sharp$: Increasing the security and efficiency of HB^+*, EUROCRYPT 2008 (Nigel P. Smart, ed.), LNCS, vol. 4965, Springer, April 2008, pp. 361–378.

[GRS08b] ______, *How to encrypt with the LPN problem*, ICALP 2008, Part II (Luca Aceto, Ivan Damgard, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, eds.), LNCS, vol. 5126, Springer, July 2008, pp. 679–690.

[HB00] N. Hopper and M. Blum, *A secure human-computer authentication scheme*, Tech. Report CMU-CS-00-139, Carnegie Mellon University, 2000.

[HB01] Nicholas J. Hopper and Manuel Blum, *Secure human identification protocols*, ASIACRYPT 2001 (Colin Boyd, ed.), LNCS, vol. 2248, Springer, December 2001, pp. 52–66.

[HLPS11] Guillaume Hanrot, Vadim Lyubashevsky, Chris Peikert, and Damien Stehlé, *Personal communication*, 2011.

[HMV03] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone, *Guide to elliptic curve cryptography*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[JW05] Ari Juels and Stephen A. Weis, *Authenticating pervasive devices with human protocols*, CRYPTO 2005 (Victor Shoup, ed.), LNCS, vol. 3621, Springer, August 2005, pp. 293–308.

[KPC+11] Eike Kiltz, Krzysztof Pietrzak, David Cash, Abhishek Jain, and Daniele Venturi, *Efficient authentication from hard learning problems*, EUROCRYPT, 2011, pp. 7–26.

[KS06a] Jonathan Katz and Ji Sun Shin, *Parallel and concurrent security of the HB and HB+ protocols*, EUROCRYPT 2006 (Serge Vaudenay, ed.), LNCS, vol. 4004, Springer, May / June 2006, pp. 73–87.

[KS06b] Jonathan Katz and Adam Smith, *Analyzing the HB and HB+ protocols in the "large error" case*, Cryptology ePrint Archive, Report 2006/326, 2006, `http://eprint.iacr.org/`.

[KSS10] Jonathan Katz, Ji Sun Shin, and Adam Smith, *Parallel and concurrent security of the HB and HB+ protocols*, Journal of Cryptology **23** (2010), no. 3, 402–421.

[KW05] Ziv Kfir and Avishai Wool, *Picking virtual pockets using relay attacks on contactless smartcard*, Security and Privacy for Emerging Areas in Communications Networks, International Conference on **0** (2005), 47–58.

[KW06] Ilan Kirschenbaum and Avishai Wool, *How to build a low-cost, extended-range RFID skimmer*, Proceedings of the 15th USENIX Security Symposium (SECURITY 2006), USENIX Association, August 2006, pp. 43–57.

[LF06] Éric Levieil and Pierre-Alain Fouque, *An improved LPN algorithm*, SCN 06(Roberto De Prisco and Moti Yung, eds.), LNCS, vol. 4116, Springer, September 2006, pp. 348–359.

[LLS09] Hyubgun Lee, Kyounghwa Lee, and Yongtae Shin, *AES implementation and performance evaluation on 8-bit microcontrollers*, CoRR **abs/0911.0482** (2009).

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev, *On ideal lattices and learning with errors over rings*, EUROCRYPT 2010 (Henri Gilbert, ed.), LNCS, vol. 6110, Springer, May 2010, pp. 1–23.

[Lyu05] Vadim Lyubashevsky, *The parity problem in the presence of noise, decoding random linear codes, and the subset sum problem*, APPROX-RANDOM, 2005, pp. 378–389.

[MSP] *MSP430 datasheeet.*

[OOV08] Khaled Ouafi, Raphael Overbeck, and Serge Vaudenay, *On the security of HB# against a man-in-the-middle attack*, ASIACRYPT, 2008, pp. 108–124.

[Poe] B. Poettering, *AVRAES: The AES block cipher on AVR controllers*, "`http://point-at-infinity.org/avraes/`".

[Reg09] Oded Regev, *On lattices, learning with errors, random linear codes, and cryptography*, J. ACM **56** (2009), no. 6.

[Tik] Jeff Tikkanen, *AES implementation on AVR ATmega328p*, "http://cs.ucsb.edu/~koc/cs178/projects/JT/avr_aes.html".

[Wik] WISP Wiki, *WISP 4.0 DL hardware*, "http://wisp.wikispaces.com/WISP+4.0+DL".

A Man-in-the-Middle Attack

In this section, we sketch a man-in-the-middle attack against the protocol in Figure 1 that recovers the secret key in time approximately $O\left(n^{1.5} \cdot 2^{\lambda/2}\right)$ when the adversary is able to insert himself into that many valid interactions between the reader and the tag. For a ring $\mathsf{R} = \mathsf{F}_2[X]/(f)$ and a polynomial $g \in \mathsf{R}$, define the vector $\vec{g}$ to be a vector of dimension $deg(f)$ whose i^{th} coordinate is the X^i coefficient of g. Similarly, for a polynomial $h \in \mathsf{R}$, let $Rot(h)$ be a $deg(f) \times deg(f)$ matrix whose i^{th} column (for $0 \leq i < deg(f)$) is $\overrightarrow{h \cdot X^i}$, or in other words, the coefficients of the polynomial $h \cdot X^i$ in the ring R. From this description, one can check that for two polynomials $g, h \in \mathsf{R}$, the product $\overrightarrow{g \cdot h} = Rot(g) \cdot \vec{h} \bmod 2 = Rot(h) \cdot \vec{g} \bmod 2$.

We now move on to describing the attack. The ith (successful) interaction between a reader $\mathcal{R}$ and a tag $\mathcal{T}$ consists of the reader sending the challenge c_i, and the tag replying with the pair (r_i, z_i) where $z_i - r_i \cdot (s \cdot \pi(c_i) + s')$ is a low-weight polynomial of weight at most $n \cdot \tau'$. The adversary who is observing this interaction will forward the challenge c_i untouched to the tag, but reply to the reader with the ordered pair $(r_i, z_i' = z_i + e_i)$ where e_i is a vector that is strategically chosen with the hope that the vector $z_i' - r_i \cdot (s \cdot \pi(c_i) + s')$ is *exactly* of weight $n \cdot \tau'$. It's not hard to see that it's possible to choose such a vector e_i so that the probability of $z_i' - r_i \cdot (s \cdot \pi(c_i) + s')$ being of weight $n \cdot \tau'$ is approximately $1/\sqrt{n}$. The response (r_i, z_i') will still be valid, and so the reader will accept. By the birthday bound, after approximately $2^{\lambda/2}$ interactions, there will be a challenge c_j that is equal to some previous challenge c_i. In this case, the adversary replies to the reader with (r_i, z_i''), where the polynomial z_i'' is just the polynomial z_i' whose first bit (i.e. the constant coefficient) is flipped. What the adversary is hoping for is that the reader accepted the response (r_i, z_i') but rejects (r_i, z_i''). Notice that the only way this can happen is if the first bit of z_i' is equal to the first bit of $r_i \cdot (s \cdot \pi(c_i) + s')$, and thus flipping it, increases the error by 1 and makes the reader reject. We now explain how finding such a pair of responses can be used to recover the secret key.

Since the polynomial expression $z_i' - r_i \cdot (s \cdot \pi(c_i) + s') = z_i' - r_i \cdot \pi(c_i) \cdot s - r_i \cdot s'$ can be written as matrix-vector multiplications as

$$\vec{z_i'} - Rot(r_i \cdot \pi(c_i)) \cdot \vec{s} - Rot(r_i) \cdot \vec{s'} \bmod 2,$$

if we let the first bit of $\vec{z_i'}$ be β_i, the first row of $Rot(r_i \cdot \pi(c_i))$ be $\vec{a_i}$ and the first row of $Rot(r_i)$ be $\vec{b_i}$, then we obtain the linear equation

$$\langle \vec{a_i}, \vec{s} \rangle + \langle \vec{b_i}, \vec{s'} \rangle = \beta_i.$$

To recover the entire secret s, s', the adversary needs to repeat the above attack until he obtains $2n$ linearly-independent equations (which can be done with $O(n)$ successful attacks), and then use Gaussian elimination to recover the full secret.

The Cryptographic Power of Random Selection

Matthias Krause and Matthias Hamann

Theoretical Computer Science
University of Mannheim
Mannheim, Germany

Abstract. The principle of random selection and the principle of adding biased noise are new paradigms used in several recent papers for constructing lightweight RFID authentication protocols. The cryptographic power of adding biased noise can be characterized by the hardness of the intensively studied Learning Parity with Noise (LPN) Problem. In analogy to this, we identify a corresponding learning problem for random selection and study its complexity. Given L secret $GF(2)$-linear functions $f_1, \ldots, f_L : \{0,1\}^n \longrightarrow \{0,1\}^a$, $\mathsf{RandomSelect}\,(L, n, a)$ denotes the problem of learning $f_1, \ldots, f_L$ from values $(u, f_l\,(u))$, where the secret indices $l \in \{1, \ldots, L\}$ and the inputs $u \in \{0,1\}^n$ are randomly chosen by an oracle. We take an algebraic attack approach to design a nontrivial learning algorithm for this problem, where the running time is dominated by the time needed to solve full-rank systems of linear equations over $O\left(n^L\right)$ unknowns. In addition to the mathematical findings relating correctness and average running time of the suggested algorithm, we also provide an experimental assessment of our results.

Keywords: Lightweight Cryptography, Algebraic Attacks, Algorithmic Learning, Foundations and Complexity Theory

1 Introduction

The very limited computational resources available in technical devices like RFID (radio frequency identification) tags implied an intensive search for lightweight authentication protocols in recent years. Standard block encryption functions like Triple-DES or AES seem to be not suited for such protocols largely because the amount of hardware to implement and the energy consumption to perform these operations is too high (see, e.g., [7] or [17] for more information on this topic).

This situation initiated two lines of research. The first resulted in proposals for new lightweight block encryption functions like PRESENT [4], KATAN and KTANTAN [10] by use of which standard block cipher-based authentication protocols can be made lightweight, too. A second line, and this line we follow in the paper, is to look for new cryptographic paradigms which allow for designing new symmetric lightweight authentication protocols. The two main suggestions discussed so far in the relevant literature are the principle of random selection and the principle of adding biased noise.

The principle of adding biased noise to the output of a linear basis function underlies the HB-protocol, originally proposed by Hopper and Blum [16] and later improved to HB^+ by Juels and Weis [17], as well as its variants $\text{HB}^\#$ and Trusted-HB (see [13] and [6], respectively). The protocols of the HB-family are provably secure against passive attacks with respect to the Learning Parity with Noise Conjecture but the problem to design HB-like protocols which are secure against active adversaries seems to be still unsolved (see, e.g., [14], [20], [12]).

The principle of random selection underlies, e.g., the CKK-protocols of Cichoń, Klonowski, and Kutyłowski [7] as well as the F_f-protocols in [3] and the Linear Protocols in [18]. It can be described as follows.

Suppose that the verifier Alice and the prover Bob run a challenge-response authentication protocol which uses a lightweight symmetric encryption operation $E : \{0,1\}^n \times \mathcal{K} \longrightarrow \{0,1\}^m$ of block length n, where $\mathcal{K}$ denotes an appropriate key space. Suppose further that E is weak in the sense that a passive adversary can efficiently compute the secret key $K \in \mathcal{K}$ from samples of the form $(u, E_K(u))$. This is obviously the case if E is linear.

Random selection denotes a method for compensating the weakness of E by using the following mode of operation. Instead of holding a single $K \in \mathcal{K}$, Alice and Bob share a collection $K_1, \ldots, K_L$ of keys from $\mathcal{K}$ as their common secret information, where $L > 1$ is a small constant. Upon receiving a challenge $u \in \{0,1\}^n$ from Alice, Bob chooses a random index $l \in \{1, \ldots, L\}$ and outputs the response $y = E(u, K_l)$. The verification of y with respect to u can be efficiently done by computing $E_{K_l}^{-1}(y)$ for all $l = 1, \ldots, L$.

The main problem this paper is devoted to is to determine the level of security which can be reached by applying this principle of random selection.

Note that the protocols introduced in [7], [3], and [18] are based on random selection of $GF(2)$-linear functions. The choice of linear basis functions is motivated by the fact that they can be implemented efficiently in hardware and have desirable pseudo-random properties with respect to a wide range of important statistical tests.

It is quite obvious that, with respect to passive adversaries, the security of protocols which use random selection of linear functions can be bounded from above by the complexity of the following learning problem referred to as $\mathsf{RandomSelect}\,(L, n, a)$: Learn $GF(2)$-linear functions $f_1, \ldots, f_L : \{0,1\}^n \longrightarrow \{0,1\}^a$ from values $(u, f_l(u))$, where the secret indices $l \in \{1, \ldots, L\}$ and the inputs $u \in \{0,1\}^n$ are randomly chosen by an oracle. In order to illustrate this notion, we sketch in appendix B how an efficient learning algorithm for $\mathsf{RandomSelect}\,(L, n, a)$ can be used for attacking the linear $(n, k, L)^+$-protocol described by Krause and Stegemann [18]. In the course of the respective arguments, it will also become clear why we just changed our notation from $\{0,1\}^m$ to $\{0,1\}^a$, as, unlike m, the parameter a is controlled by the attacker.

In this paper, we present an algebraic attack approach for solving the above learning problem $\mathsf{RandomSelect}\,(L, n, a)$. The running time of our algorithm is dominated by the effort necessary to solve a full-rank system of linear equa-

tions of $O(n^L)$ unknowns over the field $GF(2^a)$. Note that trivial approaches for solving $\mathsf{RandomSelect}\,(L,n,a)$ lead to a running time exponential in n.

In recent years, people from cryptography as well as from complexity and coding theory devoted much interest to the solution of learning problems around linear structures. Prominent examples in the context of lightweight cryptography are the works by Goldreich and Levin [15], Regev [21], and Arora and Ge [2]. But all these results are rather connected to the Learning Parity with Noise Problem. To the best of our knowledge, there are currently no nontrivial results with respect to the particular problem of learning randomly selected linear functions, which is studied in the present paper.

We are strongly convinced that the complexity of RandomSelect also defines a lower bound on the security achievable by protocols using random selection of linear functions, e.g., the improved $(n,k,L)^{++}$-protocol in [18]. Thus, the running time of our algorithm hints at how the parameters n, k, and L should be chosen in order to achieve an acceptable level of cryptographic security. Note that even for moderate choices like $n = 128$ and $L = 8$ or $n = 256$ and $L = 4$, solving $\mathsf{RandomSelect}\,(L,n,a)$ by means of our algorithm implies solving a system of around 2^{28} unknowns, which should be classified as sufficiently difficult in many practical situations.

The paper is organized as follows. In sections 2, 3, and 4, our learning algorithm, which conducts an algebraic attack in the spirit of [22], will be described in full detail. We represent the L linear basis functions as assignments A to a collection $X = \left(x_i^l\right)_{i=1,\ldots,n,l=1,\ldots,L}$ of variables taking values from the field $K = GF(2^a)$. We will then see that each example $(u, f_l\,(u))$ induces a degree-L equation of a certain type in the X-variables, which allows for reducing the learning problem $\mathsf{RandomSelect}\,(L,n,a)$ to the problem of solving a system of degree-L equations over K. While, in general, the latter problem is known to be NP-hard, we can show an efficient way to solve this special kind of systems.

One specific problem of our approach is that, due to inherent symmetries of the degree-L equations, we can never reach a system which has full linear rank with respect to the corresponding monomials. In fact, this is the main difference between our learning algorithm and the well-known algebraic attack approaches for cryptanalyzing LFSR-based keystream generators (see, e.g., [19], [8], [9], [1]).

We circumvent this problem by identifying an appropriate set $T(n,L)$ of basis polynomials of degree at most L which allow to express the degree-L equations as linear equations over $T(n,L)$. The choice of $T(n,L)$ will be justified by Theorem 2 saying that if $|K| \geq L$, then the system of linear equations over $T(n,L)$ induced by all possible examples has full rank $|T(n,L)|$ (note that according to Theorem 1, this is not true if $|K| < L$). Our experiments, which are presented in section 5, indicate that if $|K| \geq L$, then with probability close to one, the number of examples needed to get a full rank system over $T(n,L)$ exceeds $|T(n,L)|$ only by a small constant factor. This implies that the effort to compute the unique *weak* solution $t\,(A) = \left(t_*\,(A)\right)_{t_* \in T(n,L)}$ corresponding to the *strong* solution A equals the time needed to solve a system of $|T(n,L)|$ linear equations over K.

But in contrast to the algebraic attacks in [19], [8], [9], [1], we still have to solve another nontrivial problem, namely, to compute the *strong* solution A, which identifies the secret functions $f_1, \ldots, f_L$, from the unique weak solution. An efficient way to do this will complete our learning algorithm for $\mathsf{RandomSelect}\,(L, n, a)$ in section 4. Finally, we also provide an experimental evaluation of our estimates using the computer algebra system Magma [5] in section 5 and conclude this paper with a discussion of the obtained results as well as an outlook on potentially fruitful future work in section 6.

2 The Approach

We fix positive integers n, a, L and secret $GF(2)$-linear functions $f_1, \ldots, f_L : \{0,1\}^n \longrightarrow \{0,1\}^a$. The learner seeks to deduce specifications of $f_1, \ldots, f_L$ from an oracle which outputs in each round an example $(u, w) \in \{0,1\}^n \times \{0,1\}^a$ in the following way. The oracle chooses independently and uniformly a random input $u \in_U \{0,1\}^n$ and a secret random index $l \in_U [L]$[1], computes $w = f_l(u)$ and outputs (u, w).

It is easy to see that RandomSelect can be efficiently solved in the case $L = 1$ by collecting examples $(u^1, w_1), \ldots, (u^m, w_m)$ until $\{u^1, \ldots, u^m\}$ contains a basis of $GF(2)^n$. The expected number of iterations until the above goal is reached can be approximated by $n + 1.61$ (see, e.g., the appendix in [11]).

We will now treat the case $L > 1$, which immediately yields a sharp rise in difficulty. First we need to introduce the notion of a *pure basis*.

Definition 1. *Let us call a set* $\mathcal{V} = \{(u^1, w_1), \ldots, (u^n, w_n)\}$ *of* n *examples a pure basis, if* $\{u^1, \ldots, u^n\}$ *is a basis of* $GF(2)^n$ *and there exists an index* $l \in [L]$ *such that* $w_i = f_l(u^i)$ *is satisfied for all* $i = 1, \ldots, n$.

Recalling our preliminary findings, we can easily infer that for $m \in Ln + \Omega(1)$, a set of m random examples contains such a pure basis with high probability. Moreover, note that for a given set $\tilde{\mathcal{V}} = \{(\tilde{u}^1, \tilde{w}_1), \ldots, (\tilde{u}^n, \tilde{w}_n)\}$ the pure basis property can be tested efficiently. The respective strategy makes use of the fact that in case of a random example (u, w), where $u = \bigoplus_{i \in I} \tilde{u}^i$ and $I \subseteq [n]$[2], the probability p that $w = \bigoplus_{i \in I} \tilde{w}_i$ holds is approximately L^{-1} if $\tilde{\mathcal{V}}$ is pure and at most $(2 \cdot L)^{-1}$ otherwise. The latter estimate is based on the trivial observation that if $\tilde{\mathcal{V}}$ is not a pure basis, it contains at least one tuple $(\tilde{u}^j, \tilde{w}_j)$, $j \in [n]$, which would have to be exchanged to make the set pure. As $j \in I$ holds true for half of all possible (but valid) examples, the probability that $w = \bigoplus_{i \in I} \tilde{w}_i$ is fulfilled although $\tilde{\mathcal{V}}$ is not pure can be bounded from above by $(2 \cdot L)^{-1}$.

[1] For a positive integer N, we denote by $[N]$ the set $\{1, \ldots, N\}$.

[2] Let $B = \{v^1, \ldots, v^n\}$ denote a basis spanning the vector space V. It is a simple algebraic fact that every vector $v \in V$ has a unique representation $I \subseteq [n]$ over B, i.e., $v = \bigoplus_{i \in I} v^i$.

However, it seems to be nontrivial to extract a pure basis from a set of $m \in Ln + \Omega(1)$ examples. Exhaustive search among all subsets of size n yields a running time exponential in n. This can be shown easily by applying Stirling's formula[3] to the corresponding binomial coefficient $\binom{m}{n}$.

We exhibit the following alternative idea for solving $\mathsf{RandomSelect}(L, n, a)$ for $L > 1$. Let $e^1, \ldots, e^n$ denote the standard basis of the $GF(2)$-vector space $\{0,1\}^n$ and keep in mind that $\{0,1\}^n = GF(2)^n \subseteq K^n$, where K denotes the field $GF(2^a)$. For all $i = 1, \ldots, n$ and $l = 1, \ldots, L$ let us denote by x_i^l a variable over K representing $f_l(e^i)$. Analogously, let A denote the $(n \times L)$-matrix with coefficients in K completely defined by $A_{i,l} = f_l(e^i)$. Henceforth, we will refer to A as a *strong solution* of our learning problem, thereby indicating the fact that its coefficients fully characterize the underlying secret $GF(2)$-linear functions $f_1, \ldots, f_L$.

Observing an example (u, w), where $u = \bigoplus_{i \in I} e^i$, the only thing we know is that there is some index $l \in [L]$ such that $w = \bigoplus_{i \in I} A_{i,l}$. This is equivalent to the statement that A is a solution of the following degree-L equation in the x_i^l-variables.

$$\left(\bigoplus_{i \in I} x_i^1 \oplus w\right) \cdot \ldots \cdot \left(\bigoplus_{i \in I} x_i^L \oplus w\right) = 0. \tag{1}$$

Let g be a partial mapping from $[L]$ to $[n]$, whose domain and image will henceforth be denoted by $\mathsf{dom}(g)$ and $\mathsf{im}(g)$, respectively. Note that equation (1) can be rewritten as

$$\bigoplus_{J \subseteq I, 1 \leq |J| \leq L'} \bigoplus_{j=|J|}^{L} w^{L-j} t_{J,j} = w^L, \tag{2}$$

$L' = \min\{L, |I|\}$, where the basis polynomials $t_{J,j}$ are defined as

$$t_{J,j} = \bigoplus_{g, |\mathsf{dom}(g)|=j, \mathsf{im}(g)=J} m_g$$

for all $J \subseteq [n]$, $1 \leq |J| \leq L$, and all j, $|J| \leq j \leq L$. The corresponding monomials m_g are in turn defined as

$$m_g = \prod_{l \in \mathsf{dom}(g)} x_{g(l)}^l$$

for all partial mappings g (as introduced above).

[3] Stirling's formula is an approximation for large factorials and commonly written $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$.

Let $T(n,L) = \{t_{J,j} \mid J \subseteq [n], 1 \leq |J| \leq L, |J| \leq j \leq L\}$ denote the set of all basis polynomials $t_{J,j}$ which may appear as part of equation (2). Moreover, we define

$$\Phi(a,b) = \sum_{i=0}^{b} \binom{a}{i}$$

for integers $0 \leq b \leq a$ and write

$$\begin{aligned} |T(n,L)| &= \sum_{j=1}^{L} \binom{n}{j} (L-j+1) \\ &= (L+1)(\Phi(n,L)-1) - \sum_{j=1}^{L} n \binom{n-1}{j-1} \\ &= (L+1)(\Phi(n,L)-1) - n\Phi(n-1,L-1). \end{aligned} \tag{3}$$

Consequently, each set of examples $\mathcal{V} = \{(u^1, w_1), \ldots, (u^m, w_m)\}$ yields a system of m degree-L equations in the x_i^l-variables, which can be written as m K-linear equations in the $t_{J,j}$-variables. In particular, the strong solution $A \in K^{n \times L}$ satisfies the relation

$$M(\mathcal{V}) \circ t(A) = W(\mathcal{V}), \tag{4}$$

where

- $K^{n \times L}$ denotes the set of all $(n \times L)$-matrices with coefficients from K,
- $M(\mathcal{V})$ is an $(m \times |T(n,L)|)$-matrix built by the m linear equations of type (2) corresponding to the examples in $\mathcal{V}$,
- $W(\mathcal{V}) \in K^m$ is defined by $W(\mathcal{V})_i = w_i^L$[4] for all $i = 1, \ldots, m$,
- $t(A) \in K^{T(n,L)}$ is defined by $t(A) = (t_{J,j}(A))_{J \subseteq [n], 1 \leq |J| \leq L, |J| \leq j \leq L}$.

Note that in section 3, we will treat the special structure of $M(\mathcal{V})$ in further detail. Independently, it is a basic fact from linear algebra that if $M(\mathcal{V})$ has full column rank, then the linear system (4) has the unique solution $t(A)$, which we will call the *weak solution*.

Our learning algorithm proceeds as follows:

(1) Grow a set of examples $\mathcal{V}$ until $M(\mathcal{V})$ has full column rank $|T(n,L)|$.
(2) Compute the unique solution $t(A)$ of system (4), i.e., the weak solution of our learning problem, by using an appropriate algorithm which solves systems of linear equations over K.
(3) Compute the strong solution A from $t(A)$.

We discuss the correctness and running time of steps (1) and (2) in section 3 and an approach for step (3) in section 4.

[4] Keep in mind that, unlike for the previously introduced K-variables $x_s^1, \ldots, x_s^L$, $s \in [n]$, the superscripted L in case of w_i^L is not an index but an exponent. See, e.g., equation (2).

3 On Computing a Weak Solution

Let n and L be arbitrarily fixed such that $2 \leq L \leq n$ holds. Moreover, let $\mathcal{V} \subseteq \{0,1\}^n \times K$ denote a given set of examples obtained through linear functions $f_1, \ldots, f_L : \{0,1\}^n \longrightarrow K$, where $K = GF(2^a)$. By definition, for each tuple $(u,w) \in \mathcal{V}$, where $u = \bigoplus_{i \in I} e^i$ and $I \subseteq [n]$ denotes the unique representation of u over the standard basis $e^1, \ldots, e^n$ of $\{0,1\}^n$, the relation $w = f_{l'}(u) = \bigoplus_{i \in I} f_{l'}(e^i)$ is bound to hold for some $l' \in [L]$. We denote by $K^{\min} \subseteq K$ the subfield of K generated by all values $f_l(e^i)$, where $l \in [L]$ and $i \in [n]$. Note that $w \in K^{\min}$ for all examples (u,w) induced by $f_1, \ldots, f_l$.

In the following, we show that our learning algorithm is precluded from succeeding if the secret linear functions $f_1, \ldots, f_L$ happen to be of a certain type or if K itself lacks in size.

Theorem 1 *If $|K^{\min}| < L$, then the columns of $M(\mathcal{V})$ are linearly dependent for any set $\mathcal{V}$ of examples, i.e., a unique weak solution does not exist.*

Proof: Let n, K, L, and $f_1, \ldots, f_L$ be arbitrarily fixed such that $2 \leq |K^{\min}| < L \leq n$ holds and let $\mathcal{V}$ denote a corresponding set of examples. Obviously, for each tuple $(u,w) \in \mathcal{V}$, where $u = \bigoplus_{i \in I} e^i$ and $I \subseteq [n]$, the two cases $1 \in I$ and $1 \notin I$ can be differentiated.

If $1 \in I$ holds, then it follows straightforwardly from equation (2) that the coefficient with coordinates (u,w) and $t_{\{1\},(L-1)}$ in $M(\mathcal{V})$ equals $w^{L-(L-1)} = w^1$. Analogously, the coefficient with coordinates (u,w) and $t_{\{1\},(L-|K^{\min}|)}$ in $M(\mathcal{V})$ equals $w^{L-(L-|K^{\min}|)} = w^{|K^{\min}|}$. Note that $t_{\{1\},(L-|K^{\min}|)}$ is a valid (and different) basis polynomial as

$$|\{1\}| = 1 \leq \left(L - |K^{\min}|\right) \leq (L-2) < (L-1) < L$$

holds for $2 \leq |K^{\min}| < L$. As $K^{\min} \subseteq K$ is a finite field of characteristic 2, we can apply Lagrange's theorem and straightforwardly conclude that the relation $z^1 = z^{|K^{\min}|}$ holds for all $z \in K^{\min}$ (including $0 \in K^{\min}$). Hence, if $1 \in I$ holds for an example (u,w), then in the corresponding row of $M(\mathcal{V})$ the two coefficients indexed by $t_{\{1\},(L-1)}$ and $t_{\{1\},(L-|K^{\min}|)}$ are always equal.

If $1 \notin I$ holds for an example (u,w), then the coefficient with coordinates (u,w) and $t_{\{1\},(L-1)}$ in $M(\mathcal{V})$ as well as the coefficient with coordinates (u,w) and $t_{\{1\},(L-|K^{\min}|)}$ in $M(\mathcal{V})$ equals 0.

Consequently, if $|K^{\min}| < L$ holds, then the column of $M(\mathcal{V})$ indexed by $t_{\{1\},(L-1)}$ equals the column indexed by $t_{\{1\},(L-|K|)}$ for any set $\mathcal{V}$ of examples, i.e., $M(\mathcal{V})$ can never achieve full column rank. □

Corollary 1 *If K is chosen such that $|K| < L$, then the columns of $M(\mathcal{V})$ are linearly dependent for any set $\mathcal{V}$ of examples, i.e., a unique weak solution does not exist.* □

While we are now aware of a lower bound for the size of K, it yet remains to prove that step (1) of our learning algorithm is, in fact, correct.

This will be achieved by introducing the $((2^n|K|) \times |T(n,L)|)$-matrix $M^* = M(\{0,1\}^n \times K)$, which clearly corresponds to the set of *all* possible examples, and showing that M^* has full column rank $|T(n,L)|$ if $L \leq |K|$ holds.

However, be careful not to misinterpret this finding, which is presented below in the form of Theorem 2. The fact that M^* has full column rank $|T(n,L)|$ by no means implies that, eventually, this will also hold for $M(\mathcal{V})$ if only the corresponding set of observations $\mathcal{V}$ is large enough (let alone the treacherous delusion that, given enough observations, even the equation $M(\mathcal{V}) = M^*$ will eventually hold, which would trivially imply that $M(\mathcal{V})$ could always reach full column rank). In particular, the experimental results summarized in section 5 (see, e.g., table 1) show that there are cases in which the rank of $M(\mathcal{V})$ is always smaller than $|T(n,L)|$, even if $L \leq |K|$ is satisfied and $\mathcal{V}$ equals the set $\{(u, f_l(u)) \mid u \in \{0,1\}^n, l \in [L]\} \subseteq \{0,1\}^n \times K$[5] of all possible *valid* examples.

Still, as a counterpart of Theorem 1, the following theorem proves the possibility of existence of a unique weak solution for arbitrary parameters n and L satisfying $2 \leq L \leq n$. In other words, choosing $T(n,L)$ to be the set of basis polynomials does not necessarily lead to systems of linear equations which cannot be solved uniquely.

Theorem 2 *Let n and L be arbitrarily fixed such that $2 \leq L \leq n$ holds. If K satisfies $L \leq |K|$, then M^* has full column rank $|T(n,L)|$.*

Proof: We denote by $\mathcal{Z}(n)$ the set of monomials $z_0^{d_0} \cdot \ldots \cdot z_n^{d_n}$, where $0 \leq d_i \leq |K| - 1$ for $i = 0, \ldots, n$. Obviously, the total number of such monomials is $|\mathcal{Z}(n)| = |K|^{n+1}$. Let us recall the aforementioned fact that the relation $z^1 = z^{|K|}$ holds for all $z \in K$ (including $0 \in K$). This straightforwardly implies that each monomial in the variables $z_0, \ldots, z_n$ is (as a function from K^{n+1} to K) equivalent to a monomial in $\mathcal{Z}(n)$. Let $\mu_{J,j}$ denote the monomial $\mu_{J,j} = z_0^{L-j} \prod_{r \in J} z_r$ for all $J \subseteq [n]$ and j, $0 \leq j \leq L$. The following lemma can be easily verified:

Lemma 2.1 *For all $J \subseteq [n]$, $1 \leq |J| \leq L$, and j, $|J| \leq j \leq L$, and examples $(u,w) \in \{0,1\}^n \times K$, it holds that $\mu_{J,j}(w,u)$ equals the coefficient in M^* which has the coordinates (u,w) and $t_{J,j}$.* □

For $i = 1, \ldots, |K|$, we denote by k_i the i-th element of the finite field K. Moreover, we suppose the convention that $0^0 = 1$ in K. Let (u,w) be an example defined as above and keep in mind that we are treating the case $L \leq |K|$. It should be observed that the coefficients in the corresponding equation of type (2) are given by w^{L-j}, where $1 \leq j \leq L$. Thus, the set of possible exponents $\{L - j \mid 1 \leq j \leq L\}$ is bounded from above by $(L-1) < L \leq |K|$. It follows straightforwardly from Lemma 2.1 that the (distinct) columns of M^* are columns

[5] It can be seen easily that for random linear functions $f_1, \ldots, f_L$, the relation $\{(u, f_l(u)) \mid u \in \{0,1\}^n, l \in [L]\} \neq \{0,1\}^n \times K$ will always hold if $L < |K|$ and is still very likely to hold if $L = |K|$.

of the matrix $W \otimes B^{\otimes n}$, where

$$W = \left(k_i^j\right)_{i=1,\ldots,|K|,j=0,\ldots,|K|-1} \quad \text{and} \quad B = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}.$$

As W and B are regular, $W \otimes B^{\otimes n}$ is regular, too. This, in turn, implies that the columns of M^* are linearly independent, thus proving Theorem 2. □

We will see in section 4 that for $|K| \in O\left(dnL^4\right)$, the strong solution can be reconstructed from the weak solution in time $n^{O(L)}$ with error probability at most d^{-1}. Furthermore, section 5 will feature an experimental assessment of the number of random (valid) observations needed until $M(\mathcal{V})$ achieves full column rank $|T(n,L)|$ for various combinations of n, L, and K (see table 2).

4 On Computing a Strong Solution from the Unique Weak Solution

Let n, K, L, and $f_1, \ldots, f_L$ be defined as before. Remember that the goal of our learning algorithm is to compute a strong solution fully characterized by the L sets $\left\{\left(e^i, f_l\left(e^i\right)\right) \mid i \in [n]\right\}$, $l = 1, \ldots, L$, where e^i denotes the i-th element of the standard basis of $GF(2)^n$ and $f_l\left(e^i\right) = x_i^l \in K$. Obviously, this information can equivalently be expressed as a matrix $A \in K^{n \times L}$ defined by $A_{i,\cdot} = \left(x_i^1, \ldots, x_i^L\right)$ for all $i = 1, \ldots, n$.

Hence, we have to solve the following problem: Compute the matrix $A \in K^{n \times L}$ from the information $t(A)$, where

$$t(A) = \left(t_{J,j}(A)\right)_{J \subseteq [n], 1 \leq |J| \leq L, |J| \leq j \leq L}$$

is the unique weak solution determined previously. But before we lay out how (and under which conditions) a strong solution A can be found, we need to introduce the following two definitions along with an important theorem linking them:

Definition 2. *Let for all vectors $x \in K^L$ the signature $sgt(x)$ of x be defined as $sgt(x) = \left(|x|_k\right)_{k \in K}$, where $|x|_k$ denotes the number of components of x which equal k.*

Furthermore, consider the following new family of polynomials:

Definition 3. *a) For all $L \geq 1$ and $j \geq 0$ let the simple symmetric polynomial s_j over the variables $x_1, \ldots, x_L$ be defined by $s_0 = 1$ and*

$$s_j = \bigoplus_{S \subseteq [L], |S| = j} m_S,$$

where $m_S = \prod_{i \in S} x_i$ for all $S \subseteq [L]$. Moreover, we denote

$$s(x) = \left(s_0(x), s_1(x), \ldots, s_L(x)\right)$$

for all $x \in K^L$.

b) Let n, L, $1 \leq L \leq n$, hold as well as j, $0 \leq j \leq L$, and $J \subseteq [n]$. The symmetric polynomial $s_{J,j} : K^{n \times L} \longrightarrow K$ is defined by

$$s_{J,j}(A) = s_j \left(\bigoplus_{i \in J} A_{i,\cdot} \right)$$

for all matrices $A \in K^{n \times L}$. Moreover, we denote

$$s_J(A) = (s_{J,0}(A), \ldots, s_{J,L}(A)).$$

The concept of signatures introduced in Definition 2 and the family of simple symmetric polynomials described in Definition 3 will now be connected by the following theorem:

Theorem 3 *For all $L \geq 1$ and $x, x' \in K^L$ it holds that $s(x) = s(x')$ if and only if $sgt(x) = sgt(x')$.*

Proof: See appendix A.

Building on this result, we can then prove the following proposition, which is of vital importance for computing the strong solution A on the basis of the corresponding weak solution $t(A)$:

Theorem 4 *Let $A \in K^{n \times L}$ and $t(A)$ be defined as before. For each subset $I \subseteq [n]$ of rows of A, the signature of the sum of these rows, i.e., $sgt\left(\bigoplus_{i \in I} A_{i,\cdot}\right)$, can be computed by solely using information derived from $t(A)$, in particular, without knowing the underlying matrix A itself.*

Proof: We first observe that the s-polynomials can be written as linear combinations of the t-polynomials. Trivially, the relation $t_{\{i\},j} = s_{\{i\},j}$ holds for all $i \in [n]$ and j, $1 \leq j \leq L$. Moreover, for all $I \subseteq [n]$, $|I| > 1$, it holds that

$$s_{I,j} = \bigoplus_{Q \subseteq I, 1 \leq |Q| \leq j} \left(\bigoplus_{g:[L] \longrightarrow [n], |dom(g)|=j, im(g)=Q} m_g \right) = \bigoplus_{Q \subseteq I, 1 \leq |Q| \leq j} t_{Q,j}. \quad (5)$$

Note that for all $J \subseteq [n]$ and j, $|J| \leq j \leq L$, relation (5) implies

$$t_{J,j} = s_{J,j} \oplus \bigoplus_{Q \subset J} t_{Q,j}. \quad (6)$$

By an inductive argument, we obtain from relation (6) that the converse is also true, i.e., the t-polynomials can be written as linear combinations of the s-polynomials.

We have seen so far that given $t(A)$, we are able to compute $s_{I,j}$ for all j, $1 \leq j \leq L$, and each subset $I \subseteq [n]$ of rows of A. Recall

$$s_{I,j}(A) = s_j \left(\bigoplus_{i \in I} A_{i,\cdot} \right) \quad \text{and} \quad s_I(A) = (s_{I,0}(A), \ldots, s_{I,L}(A))$$

from Definition 3 and let $x \in K^L$ be defined by $x = \bigoplus_{i \in I} A_{i,\cdot}$. It can be easily seen that $s_I(A) = s(x)$ holds.

In conjunction with Theorem 3, this straightforwardly implies the validity of Theorem 4. □

Naturally, it remains to assess the degree of usefulness of this information when it comes to reconstructing the strong solution $A \in K^{n \times L}$. In the following, we will prove that if K is large enough, then with high probability, A can be completely (up to column permutations) and efficiently derived from the signatures of all single rows of A and the signatures of all sums of pairs of rows of A:

Theorem 5 *Let $K = GF(2^a)$ fulfill $|K| \geq \frac{1}{4} \cdot d \cdot n \cdot L^4$, i.e., $a \geq \log(n) + \log(d) + 4\log(L) - 2$. Then, for a random matrix $A \in_U K^{n \times L}$, the following is true with a probability of approximately at least $\left(1 - \frac{1}{d}\right)$: A can be completely reconstructed from the signatures $sgt(A_{i,\cdot})$, $1 \leq i \leq n$, and $sgt(A_{i,\cdot} \oplus A_{j,\cdot})$, $1 \leq i < j \leq n$.*

Proof: See appendix A.

As we have seen now that, under certain conditions, it is possible to fully reconstruct the strong solution A by solely resorting to information obtained from the weak solution $t(A)$, we can proceed to actually describe a conceivable approach for step (3) of the learning algorithm:

We choose a constant error parameter d and an exponent a, i.e., $K = GF(2^a)$, in such a way that Theorem 5 can be applied. Note that $L \leq n$ and $|K| \in n^{O(1)}$. In a pre-computation, we generate two databases DB_1 and DB_2 of size $n^{O(L)}$. While DB_1 acts as a lookup table with regard to the one-to-one relation between $s(x)$ and $sgt(x)$ for all $x \in K^L$, we use DB_2 to store all triples of signatures $S, S', \tilde{S}$ for which there is exactly one solution pair $x, y \in K^L$ fulfilling $sgt(x) = S$ and $sgt(y) = S'$ as well as $sgt(x \oplus y) = \tilde{S}$.

Given $t(A)$, i.e., the previously determined weak solution, we then compute $sgt(A_{i,\cdot})$ for all i, $1 \leq i \leq n$, and $sgt(A_{i,\cdot} \oplus A_{j,\cdot})$ for all i, j, $1 \leq i < j \leq n$, in time $n^{O(1)}$ by using DB_1 and relation (5), which can be found in the proof of Theorem 4. According to Theorem 5, it is now possible to reconstruct A by the help of database DB_2 with probability at least $1 - \frac{1}{d}$.

5 Experimental Results

To showcase the detailed workings of our learning algorithm as well as to evaluate its efficiency at a practical level, we created a complete implementation using the computer algebra system Magma. In case of success, it takes approximately 90 seconds on standard PC hardware (Intel i7, 2.66 GHz, with 6 GB RAM) to compute the unique strong solution on the basis of a set of 10,000 randomly generated examples for $n = 10$, $a = 3$ (i.e., $K = GF(2^a)$), and $L = 5$. Relating to this, we performed various simulations in order to assess the corresponding

Parameters			Performed Iterations				
			Rank of $M(\mathcal{V}) < \|T(n,L)\|$		Rank of $M(\mathcal{V}) = \|T(n,L)\|$		Total
n	K	L	Number	Ratio	Number	Ratio	Number
4	$GF(2^2)$	2	37	0.37 %	9,963	99.63 %	10,000
4	$GF(2^2)$	3	823	8.23 %	9,177	91.77 %	10,000
4	$GF(2^2)$	4	7,588	75.88 %	2,412	24.12 %	10,000
5	$GF(2^2)$	4	4,556	45.56 %	5,444	54.44 %	10,000
5	$GF(2^2)$	5	10,000	100.00 %	0	0.00 %	10,000
6	$GF(2^3)$	4	0	0.00 %	1,000	100.00 %	1,000
8	$GF(2^3)$	4	0	0.00 %	1,000	100.00 %	1,000
8	$GF(2^3)$	6	0	0.00 %	100	100.00 %	100
8	$GF(2^3)$	7	0	0.00 %	100	100.00 %	100
8	$GF(2^3)$	8	0	0.00 %	100	100.00 %	100
9	$GF(2^3)$	8	0	0.00 %	10	100.00 %	10
9	$GF(2^3)$	9	10	100.00 %	0	0.00 %	10

Table 1. An estimate of the rank of $M(\mathcal{V})$ on the basis of all possible valid observations for up to 10,000 randomly generated instances of $\mathsf{RandomSelect}(L, n, a)$. For each choice of parameters, $|T(n, L)|$ denotes number of columns of $M(\mathcal{V})$ as defined in section 2 and listed in table 2.

probabilities, which were already discussed in sections 3 and 4 from a theoretical point of view.

The experimental results summarized in table 1 clearly suggest that if $|K|$ is only slightly larger than the number L of secret linear functions, then in all likelihood, $M(\mathcal{V})$ will eventually reach full (column) rank $|T(n, L)|$, thus allowing for the computation of a unique weak solution. Moreover, in accordance with Corollary 1, the columns of $M(\mathcal{V})$ were always linearly dependent in the case of $n = 5$, $K = GF(2^2)$ and $L = 5$, i.e., $|K| = 4 < 5 = L$. A further analysis of the underlying data revealed in addition that, for arbitrary combinations of n, K, and L, the matrix $M(\mathcal{V})$ never reached full column rank if at least two of the corresponding L random linear functions $f_1, \ldots, f_L$ were identical during an iteration of our experiments. Note that, on the basis of the current implementation, it was not possible to continue table 1 for larger parameter sizes because, e.g., in the case of $n = 8$, $K = GF(2^3)$ and $L = 7$, performing as few as 100 iterations already took more than 85 minutes on the previously described computer system.

Table 2 features additional statistical data with respect to the number of examples needed (in case of success) until the matrix $M(\mathcal{V})$ reaches full column rank $|T(n, L)|$. Please note that, in contrast to the experiments underlying table 1, such examples $(u, f_l(u))$ are generated iteratively and independently choosing random pairs $u \in_U \{0,1\}^n$ and $l \in_U [L]$, i.e., they are not processed in their canonical order but observed randomly (and also repeatedly) to simulate a practical passive attack. While we have seen previously that for most choices of n, K and L, the matrix $M(\mathcal{V})$ is highly likely to eventually reach full column rank, the experimental results summarized in table 2, most notably the observed p-

Parameters				Number of Random Examples until Rank $(M(\mathcal{V})) = \lvert T(n,L) \rvert$							
n	K	L	$\lvert T(n,L) \rvert$	Avg.	Max.	Min.	$Q_{0.1}$	$Q_{0.25}$	$Q_{0.5}$	$Q_{0.75}$	$Q_{0.9}$
4	$GF(2^2)$	1	4	5.5	18	4	4	4	5	6	8
4	$GF(2^2)$	2	14	24.4	93	14	18	20	23	27	32
4	$GF(2^2)$	3	28	71.8	273	33	51	58	67	81	99
4	$GF(2^2)$	4	43	226.2	701	95	147	175	211	261	317
5	$GF(2^2)$	4	75	218.5	591	140	176	192	211	237	263
6	$GF(2^3)$	4	124	201.6	318	162	184	192	200	211	220
8	$GF(2^3)$	4	298	378.7	419	345	365	371	378	386	393
8	$GF(2^3)$	6	762	1401.6	1565	1302	1342	1364	1405	1427	1458
8	$GF(2^3)$	7	1016	2489.7	2731	2275	2369	2417	2477	2547	2645
8	$GF(2^3)$	8	1271	5255.3	7565	4302	4706	4931	5227	5557	5706
9	$GF(2^3)$	8	2295	6266.1	6553	6027	6078	6136	6199	6415	6504

Table 2. An estimate of the number of randomly generated examples $(u, f_l(u))$ which have to be processed (in case of success) until the matrix $M(\mathcal{V})$ reaches full column rank $\lvert T(n,L) \rvert$. Given a probability p, we denote by Q_p the p-quantile of the respective sample.

quantiles, strongly suggest that our learning algorithm for $\mathsf{RandomSelect}(L, n, a)$ will also be able to efficiently construct a corresponding LES which allows for computing a unique weak solution.

Parameters			Performed Iterations (i.e., randomly chosen $A \in_U K^{n \times L}$)				
			A **not** $sgt(2)$-identifiable		A **was** $sgt(2)$-identifiable		Total
n	K	L	Number	Ratio	Number	Ratio	Number
4	$GF(2^2)$	2	0	0.00 %	10,000	100.00 %	10,000
4	$GF(2^2)$	3	69	0.69 %	9,931	99.31 %	10,000
4	$GF(2^2)$	4	343	3.43 %	9,657	96.57 %	10,000
6	$GF(2^3)$	4	0	0.00 %	10,000	100.00 %	10,000
8	$GF(2^3)$	4	0	0.00 %	10,000	100.00 %	10,000
8	$GF(2^3)$	6	0	0.00 %	1,000	100.00 %	1,000
8	$GF(2^3)$	7	0	0.00 %	1,000	100.00 %	1,000
8	$GF(2^3)$	8	0	0.00 %	100	100.00 %	100
9	$GF(2^3)$	8	0	0.00 %	100	100.00 %	100

Table 3. An estimate of the ratio of $sgt(2)$-identifiable $(n \times L)$-matrices over K.

It remains to clear up the question, to what extent Theorem 5 reflects reality concerning the probability of a random $(n \times L)$-matrix over K being $sgt(2)$-identifiable (see Definitions 5.1 and 5.2 in the proof of Theorem 5), which is necessary and sufficient for the success of step (3) of our learning algorithm. Our corresponding simulations yielded table 3, which immediately suggests that even for much smaller values of $\lvert K \rvert$ than those called for in Theorem 5, a strong solution $A \in_U K^{n \times L}$ can be completely reconstructed from the signatures $sgt(A_{i,\cdot})$,

$1 \leq i \leq n$, and $sgt(A_{i,\cdot} \oplus A_{j,\cdot})$, $1 \leq i < j \leq n$. In conjunction with the experimental results concerning the rank of $M(\mathcal{V})$, this, in turn, implies that our learning algorithm will efficiently lead to success in the vast majority of cases.

6 Discussion

The running time of our learning algorithm for $\mathsf{RandomSelect}(L, n, a)$ is dominated by the complexity of solving a system of linear equations with $|T(n, L)|$ unknowns. Our hardness conjecture is that this complexity also constitutes a lower bound to the complexity of $\mathsf{RandomSelect}(L, n, a)$ itself, which would imply acceptable cryptographic security for parameter choices like $n = 128$ and $L = 8$ or $n = 256$ and $L = 6$. The experimental results summarized in the previous section clearly support this view. Consequently, employing the principle of random selection to design new symmetric lightweight authentication protocols might result in feasible alternatives to current HB-based cryptographic schemes.

A problem of independent interest is to determine the complexity of reconstructing an $sgt(r)$-identifiable matrix A from the signatures of all sums of at most r rows of A. Note that this problem is wedded to determining the complexity of $\mathsf{RandomSelect}(L, n, a)$ with respect to an *active* learner, who is able to receive examples (u, w) for inputs u of his choice, where $w = f_l(u)$ and $l \in_U [L]$ is randomly chosen by the oracle. It is easy to see that such learners can efficiently compute $sgt(f_1(u), \ldots, f_L(u))$ by repeatedly asking for u. As the approach for reconstructing A which was outlined in section 4 needs a data structure of size exponential in L, it would be interesting to know if there are corresponding algorithms of time and space costs polynomial in L.

From a theoretical point of view, another open problem is to determine the probability that a random $(n \times L)$-matrix over K is $sgt(r)$-identifiable for some r, $2 \leq r \leq L$. Based on the results of our computer experiments, it appears more than likely that the lower bound derived in Theorem 5 is far from being in line with reality and that identifiable matrices occur with much higher probability for fields K of significantly smaller size.

References

1. F. Armknecht and M. Krause. Algebraic attacks on combiners with memory. In *Proceedings of Crypto 2003*, volume 2729 of *LNCS*, pages 162–176. Springer, 2003.
2. S. Arora and R. Ge. New algorithms for learning in presence of errors. Submitted, 2010. http://www.cs.princeton.edu/~rongge/LPSN.pdf.
3. E.-O. Blass, A. Kurmus, R. Molva, G. Noubir, and A. Shikfa. The F_f-family of protocols for RFID-privacy and authentication. In *5th Workshop on RFID Security, RFIDSec'09*, 2009.
4. A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. H. Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In *Proceedings of Cryptographic Hardware and Embedded Systems (CHES) 2007*, volume 4727 of *LNCS*, pages 450–466. Springer, 2007.

5. W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997.
6. J. Bringer and H. Chabanne. Trusted-HB: A low cost version of HB^+ secure against a man-in-the-middle attack. *IEEE Trans. Inform. Theor.*, 54:4339–4342, 2008.
7. J. Cichoń, M. Klonowski, and M. Kutyłowski. Privacy protection for RFID with hidden subset identifiers. In *Proceedings of Pervasive 2008*, volume 5013 of *LNCS*, pages 298–314. Springer, 2008.
8. N. Courtois. Fast algebraic attacks on stream ciphers with linear feedback. In *Proceedings of Crypto 2003*, volume 2729 of *LNCS*, pages 176–194. Springer, 2003.
9. N. Courtois and W. Meier. Algebraic attacks on stream ciphers with linear feedback. In *Proceedings of Eurocrypt 2003*, volume 2656 of *LNCS*, pages 345–359. Springer, 2003.
10. C. De Cannière, O. Dunkelman, and M. Knežević. KATAN and KTANTAN – A family of small and efficient hardware-oriented block ciphers. In *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems (CHES) 2009*, volume 5747 of *LNCS*, pages 272–288. Springer, 2009.
11. Z. Gołebiéwski, K. Majcher, and F. Zagórski. Attacks on CKK family of RFID authentication protocols. In *Proceedings Adhoc-now 2008*, volume 5198 of *LNCS*, pages 241–250. Springer, 2008.
12. D. Frumkin and A. Shamir. Untrusted-HB: Security vulnerabilities of Trusted-HB. Cryptology ePrint Archive, Report 2009/044, 2009. `http://eprint.iacr.org`.
13. H. Gilbert, M. J. B. Robshaw, and Y. Seurin. $HB^\#$: Increasing the security and efficiency of HB^+. In *Proceedings of Eurocrypt 2008*, volume 4965 of *LNCS*, pages 361–378, 2008.
14. H. Gilbert, M. J. B. Robshaw, and H. Sibert. Active attack against HB^+: A provable secure lightweight authentication protocol. *Electronic Letters*, 41:1169–1170, 2005.
15. O. Goldreich and L. A. Levin. A hard-core predicate for all one-way functions. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing (STOC)*, pages 25–32. ACM Press, 1989.
16. N. J. Hopper and M. Blum. Secure human identification protocols. In *Proceedings of Asiacrypt 2001*, volume 2248 of *LNCS*, pages 52–66. Springer, 2001.
17. A. Juels and S. A. Weis. Authenticating pervasive devices with human protocols. In *Proceedings of Crypto 2005*, volume 3621 of *LNCS*, pages 293–308. Springer, 2005.
18. M. Krause and D. Stegemann. More on the security of linear RFID authentication protocols. In *Proceedings of SAC 2009*, volume 5867 of *LNCS*, pages 182–196. Springer, 2009.
19. W. Meier, E. Pasalic, and C. Carlet. Algebraic attacks and decomposition of boolean functions. In *Proceedings of Eurocrypt 2004*, volume 3027 of *LNCS*, pages 474–491. Springer, 2004.
20. K. Ouafi, R. Overbeck, and S. Vaudenay. On the security of $HB^\#$ against a man-in-the-middle attack. In *Proceedings of Asiacrypt 2008*, volume 5350 of *LNCS*, pages 108–124. Springer, 2008.
21. O. Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing (STOC)*, pages 84–93. ACM Press, 2005.
22. A. Shamir, J. Patarin, N. Courtois, and A. Klimov. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *Proceedings of Eurocrypt 2000*, volume 1807 of *LNCS*, pages 474–491. Springer, 2000.

A The Proofs of Theorems 3 and 5

A.1 The Proof of Theorem 3

Theorem 3 *For all $L \geq 1$ and $x, x' \in K^L$ it holds that $s(x) = s(x')$ if and only if $sgt(x) = sgt(x')$.*

Proof: The *if*-direction of Theorem 3 follows directly from the definitions of $sgt(x)$ and $s(x)$.

We will prove the *only-if*-direction of Theorem 3 by induction on L. The case $L = 1$ is obvious. Let us fix an arbitrary $L > 1$ and let us suppose that the following is true for all $L' < L$ and all $x, x' \in K^{L'}$: if $s(x) = s(x')$ then $sgt(x) = sgt(x')$.

Lemma 3.1 *For all $x \in K^{L-1}$, $k \in K$ and j, $1 \leq j \leq L$, the following is true: $s_j(x, k) = s_j(x) \oplus k \cdot s_{j-1}(x)$.* □

Henceforth, for all $k \in K$, $r \geq 1$ and $x \in K^r$, we write $k \in x$ if some component of x equals k.

Lemma 3.2 *For all $y, y' \in K^L$, the following is true: if $s(y) = s(y')$ and there is some $k \in K$ with $k \in y$ and $k \in y'$, then $sgt(y) = sgt(y')$.*

Proof of Lemma 3.2: Suppose, w.l.o.g., that $y = (x, k)$ and $y' = (x', k)$, where $x, x' \in K^{L-1}$. It suffices to prove that $s(x) = s(x')$ as this implies by induction hypothesis that $sgt(x) = sgt(x')$ and, consequently, $sgt(y) = sgt(y')$. We prove $s(x) = s(x')$ by showing via induction on j that $s_j(x) = s_j(x')$ holds for all j, $0 \leq j \leq L$. The case $j = 0$ follows straightforwardly from Definition 3. Let us fix some $j > 0$ and suppose that $s_r(x) = s_r(x')$ holds for all non-negative integers $r < j$. As $s_j(y) = s_j(y')$ is satisfied, it follows from Lemma 3.1, in conjunction with the induction hypothesis, that

$$s_j(x) + k \cdot s_{j-1}(x) = s_j(x') + k \cdot s_{j-1}(x') = s_j(x') + k \cdot s_{j-1}(x)$$

and, consequently, $s_j(x) = s_j(x')$ holds. □

Lemma 3.3 *For all $y, y' \in K^L$, the following is true: if $s(y) = s(y')$ and $0 \in y$, then $sgt(y) = sgt(y')$.*

Proof of Lemma 3.3: Trivially, $s(y) = s(y')$ implies that $s_L(y) = s_L(y')$. As $0 \in y$, it follows directly from Definition 3 that $s_L(y) = 0$, which, in turn, implies that $s_L(y') = 0$ and, consequently, $0 \in y'$. The proof now follows from Lemma 3.2. □

Finally, we have to consider the last remaining case of $y, y' \in K^L$ given by

- $s(y) = s(y')$,
- $0 \notin y$ and $0 \notin y'$,

- $Y \cap Y' = \emptyset$,

where Y (Y') denotes the set of components of y (y').

In order to proceed, we need the following technical definition, accompanied by two technical lemmas:

Definition 3.1 *For all $r \geq 1$ and $x \in K^r$, we denote*

$$S(x) = \bigoplus_{j=0}^{r} s_j(x).$$

Lemma 3.4 *For all $r \geq 1$ and $x \in K^r$, the following is true: $S(x) = 0$ if and only if $1 \in x$.*

Proof of Lemma 3.4: We prove the lemma by induction on r. The case $r = 1$ can be easily verified. Let us fix some $r > 1$ and suppose that for all q, $1 \leq q \leq (r-1)$, and all $z \in K^q$, the following is true: $S(z) = 0$ if and only if $1 \in z$. Furthermore, let us fix some $x \in K^r$ satisfying $S(x) = 0$ and suppose that $x = (z, k)$ for $z \in K^{r-1}$ and $k \in K$. In conjunction with Lemma 3.1, this yields the following equation:

$$\begin{aligned} 0 = S(x) = 1 \oplus \bigoplus_{j=1}^{r} s_j(x) &= 1 \oplus \bigoplus_{j=1}^{r} (s_j(z) \oplus k \cdot s_{j-1}(z)) \\ &= 1 \oplus \bigoplus_{j=1}^{r-1} s_j(z) \oplus k \cdot \bigoplus_{j=1}^{r} s_{j-1}(z) \\ &= \bigoplus_{j=0}^{r-1} s_j(z) \oplus k \cdot \bigoplus_{j=0}^{r-1} s_j(z) \\ &= (1 \oplus k) \cdot S(z) \end{aligned}$$

Consequently, either $k = 1$ or, by induction hypothesis, $1 \in z$. □

Lemma 3.5 *For all $r \geq 1$ and $x, x' \in K^r$, the following is true: if $s(x) = s(x')$, then $s(k \cdot x) = s(k \cdot x')$.*

Proof of Lemma 3.5: This follows straightforwardly from the simple fact that $s_j(k \cdot x) = k^j \cdot s_j(x)$ holds for all j, $0 \leq j \leq r$. □

The finding given below will complete the proof of Theorem 3:

Lemma 3.6 *For all $y, y' \in K^L$, the following is true: if $0 \notin y$ as well as $0 \notin y'$ and the sets of components of y and y' are disjoint, then $s(y) \neq s(y')$.*

Proof of Lemma 3.6: Due to Lemma 3.5, we can, w.l.o.g., suppose that $y_L = 1$. Let us denote $d = y'_L$, $y = (x, 1)$, $y' = (x', d)$ and keep in mind that $d \notin \{0, 1\}$. We will prove this lemma by contradiction. Hence, let us assume that $s(y) = s(y')$ holds.

By applying Lemma 3.1 to the assumption, we can deduce that

$$s_j(x) \oplus 1 \cdot s_{j-1}(x) = s_j(x') \oplus d \cdot s_{j-1}(x')$$

holds for all $j = 1, \ldots, L$. This implies

$$\begin{aligned} & s_1(x) \oplus 1 = s_1(x') \oplus d \\ \Leftrightarrow \quad & s_1(x') = s_1(x) \oplus d \oplus 1. \end{aligned}$$

Analogously, we obtain

$$s_2(x) \oplus s_1(x) = s_2(x') \oplus d \cdot s_1(x'),$$

i.e.,

$$\begin{aligned} s_2(x') &= s_2(x) \oplus s_1(x) \oplus d(s_1(x) \oplus d \oplus 1) \\ &= s_2(x) \oplus (d \oplus 1) s_1(x) \oplus d(d \oplus 1). \end{aligned}$$

Iterating this, one can easily show that

$$s_j(x') = s_j(x) \oplus \bigoplus_{r=1}^{j} \left(d^{r-1}(d \oplus 1) s_{j-r}(x)\right) \tag{7}$$

holds for all j, $1 \le j \le (L-1)$. In conjunction with the fact that

$$1 \cdot s_{L-1}(x) = s_L(y) = s_L(y') = d \cdot s_{L-1}(x')$$

in the case of $j = L$, relation (7) implies

$$\begin{aligned} & d^{-1} s_{L-1}(x) = s_{L-1}(x) \oplus \bigoplus_{r=1}^{L-1} \left(d^{r-1}(d \oplus 1) s_{L-1-r}(x)\right) \\ \Leftrightarrow \quad & 0 = d^{-1}(1 \oplus d) s_{L-1}(x) \oplus \bigoplus_{r=1}^{L-1} \left(d^{r-1}(d \oplus 1) s_{L-1-r}(x)\right) \\ \Leftrightarrow \quad & 0 = \bigoplus_{r=0}^{L-1} \left(d^{r-1}(d \oplus 1) s_{L-1-r}(x)\right). \end{aligned}$$

Multiplying this by $\left(d^{-(L-2)}\left(d\oplus 1\right)^{-1}\right)$, where $d\notin\{0,1\}$, yields

$$\begin{aligned} & 0=\bigoplus_{r=0}^{L-1}\left(d^{-((1-r)+(L-2))}s_{L-1-r}\left(x\right)\right)\\ \Leftrightarrow\quad & 0=\bigoplus_{r=0}^{L-1}\left(d^{-((L-1)-r)}s_{(L-1)-r}\left(x\right)\right)\\ \Leftrightarrow\quad & 0=\bigoplus_{j=0}^{L-1}\left(d^{-j}s_{j}\left(x\right)\right)\\ \Leftrightarrow\quad & 0=S\left(d^{-1}\cdot x\right). \end{aligned}$$

In conjunction with Lemma 3.4, this implies that $1\in\left(d^{-1}\cdot x\right)$, which, in turn, means that $d\in x$. Consequently, $d\in y$ and also $d\in y'$, which violates the condition that the sets of components of y and y' are disjoint. Hence, the assumption $s\left(y\right)=s\left(y'\right)$ must be false. □

To conclude, Theorem 3 now follows straightforwardly from Lemma 3.2, Lemma 3.3 and Lemma 3.6. □

A.2 The Proof of Theorem 5

Theorem 5 *Let $K=GF\left(2^{a}\right)$ fulfill $|K|\geq\frac{1}{4}\cdot d\cdot n\cdot L^{4}$, i.e., $a\geq\log\left(n\right)+\log\left(d\right)+4\log\left(L\right)-2$. Then, for a random matrix $A\in_{U}K^{n\times L}$, the following is true with a probability of approximately at least $\left(1-\frac{1}{d}\right)$: A can be completely reconstructed from the signatures $sgt\left(A_{i,\cdot}\right)$, $1\leq i\leq n$, and $sgt\left(A_{i,\cdot}\oplus A_{j,\cdot}\right)$, $1\leq i<j\leq n$.*

Proof: In order to prove the theorem, we first need the following definition:

Definition 5.1 *a) Two matrices $A,B\in K^{n\times L}$ are called column-equivalent if A can be obtained from B by permuting the columns.*
b) Two matrices $A,B\in K^{n\times L}$ are called $sgt\left(r\right)$-equivalent if

$$sgt\left(\bigoplus_{i\in I}A_{i,\cdot}\right)=sgt\left(\bigoplus_{i\in I}B_{i,\cdot}\right)$$

holds for all $I\subseteq[n]$, $1\leq|I|\leq r$.

Clearly, if the matrices A and B are column-equivalent, then they are also $sgt\left(r\right)$-equivalent for all r, $1\leq r\leq L$. The converse is not necessarily true as can be seen from the following example, where $A,B\in GF\left(8\right)^{2\times 3}$:

$$A=\begin{bmatrix}1+z & 1+z^{2} & 0\\ z^{2} & 1 & z\end{bmatrix}\quad\text{and}\quad B=\begin{bmatrix}1+z & 1+z^{2} & 0\\ 1 & z & z^{2}\end{bmatrix}.$$

This crucial observation leads to the following definition:

Definition 5.2 *A matrix $A \in K^{n \times L}$ is called $sgt(r)$-identifiable if $sgt(r)$-equivalence to A implies column-equivalence to A.*

We show Theorem 5 by proving a lower bound on the probability of a random matrix $A \in_U K^{n \times L}$ being $sgt(2)$-identifiable. In order to do so, we will now introduce a sufficient condition for the $sgt(2)$-identifiability of an $(n \times L)$-matrix over K and further show that with high probability, it is fulfilled if $|K|$ is large enough.

Definition 5.3 *a) For all $L \geq 1$ and vectors $x \in K^L$, we denote by $\{x\}$ the set of all $k \in K$ occurring in x, i.e., $\{x\} = \{k \in K \mid |x|_k > 0\}$.*
b) For all subsets $M \subseteq K$, we denote by $\Delta(M)$ the set of differences generated by M, i.e., $\Delta(M) = \{k \oplus k' \mid k \neq k' \in M\}$.
c) Two subsets $M, M' \subseteq K$ are called diff.disjoint if $\Delta(M) \cap \Delta(M') = \emptyset$.
d) A matrix $A \in K^{n \times L}$ is called strongly diff.disjoint if there is some $i \in [n]$ such that $|\{A_{i,\cdot}\}| = L$ and, for all $j \in [n] \setminus \{i\}$, $\{A_{i,\cdot}\}$ and $\{A_{j,\cdot}\}$ are diff.disjoint.

In addition, we need the following technical lemma:

Lemma 5.1 *Let $M, M' \subseteq K$ be two given subsets which are diff.disjoint. For all $m_1, m_2 \in M$ and $m'_1, m'_2 \in M'$, the following is true: if $m_1 \oplus m'_1 = m_2 \oplus m'_2$ then $m_1 = m_2$ and $m'_1 = m'_2$.*

Proof of Lemma 5.1: Trivially, $m_1 \oplus m'_1 = m_2 \oplus m'_2$ can be transformed into $m_1 \oplus m_2 = m'_1 \oplus m'_2$. The latter relation would obviously violate the condition of M and M' being diff.disjoint if $m_1 \neq m_2$ (and thus $m'_1 \neq m'_2$) held. □

The following lemma states a sufficient condition for the $sgt(2)$-identifiability of an $(n \times L)$-matrix over K:

Lemma 5.2 *If a matrix $A \in K^{n \times L}$ is strongly diff.disjoint, then A is also $sgt(2)$-identifiable.*

Proof of Lemma 5.2: Let us consider a strongly diff.disjoint matrix $A \in K^{n \times L}$ and suppose that, w.l.o.g., $|\{A_{1,\cdot}\}| = L$ holds (i.e., the first row of A contains the maximum number L of different elements). Furthermore, let us fix some matrix $B \in K^{n \times L}$ which is $sgt(2)$-equivalent to A. In order to prove the lemma, we have to show that A and B are column-equivalent.

As A and B are $sgt(2)$-equivalent, we know that $sgt(A_{1,\cdot}) = sgt(B_{1,\cdot})$ holds, implying the existence of some column-permutation $\rho \in \mathcal{S}_L$ such that $A_{1,\cdot} = \rho(B_{1,\cdot})$. Now let us fix some arbitrary j, $1 < j \leq n$. From $sgt(A_{j,\cdot})$, we learn which elements occur in row $B_{j,\cdot}$ and from $sgt(A_{1,\cdot} \oplus A_{j,\cdot})$, we learn which elements occur in $B_{1,\cdot} \oplus B_{j,\cdot}$. As $\{B_{1,\cdot}\}$ and $\{B_{j,\cdot}\}$ are diff.disjoint, Lemma 5.1 implies that for each element occurring in $B_{1,\cdot} \oplus B_{j,\cdot}$, there is exactly one possibility of writing it as the sum of an element from $B_{1,\cdot}$ and an element from $B_{j,\cdot}$. Moreover, these two elements have to be in the same column of B.

Due to this and the fact that all components of $B_{1,\cdot}$ are different, the positions of all elements occurring in $B_{j,\cdot}$ are uniquely determined. In particular, the aforementioned column-permutation ρ not only satisfies $A_{1,\cdot} = \rho(B_{1,\cdot})$ but also $A_{j,\cdot} = \rho(B_{j,\cdot})$ for all $1 < j \leq n$. Clearly, this proves the column-equivalence of A and B, thus implying the correctness of the lemma. □

Consequently, Theorem 5 can be shown by proving an appropriate lower bound on the probability of a random matrix $A \in_U K^{n \times L}$ being strongly diff.disjoint. Our argument will be based on the following lemma:

Lemma 5.3 *Given a subset $M \subseteq K$ such that $|M| = L$ holds and a sequence $x = (x_1, \ldots, x_L)$ chosen randomly from K^L with respect to the uniform distribution, the lower bound on the probability of $\{x\}$ being diff.disjoint from M can be approximated by $1 - \frac{L^4}{4|K|}$.*

Proof of Lemma 5.3: In the course of this proof, we will make use of the approximation $\prod_{i=1}^{q-1}\left(1 - \frac{i}{p}\right) \approx e^{-\frac{q^2}{2p}}$, commonly found in the context of the well-known birthday paradox, as well as the approximations $e^{-\frac{1}{x}} \approx \left(1 - \frac{1}{x}\right)$ and $\left(1 - \frac{1}{x}\right)^x \approx e^{-1}$. In order to obtain a sequence $x \in K^L$ whose set of components is diff.disjoint from M, for all i, $1 < i \leq L$, the element x_i needs to be chosen in such a way that

$$\Delta(M) \cap \{x_i \oplus x_1, \ldots, x_i \oplus x_{i-1}\} = \emptyset.$$

The probability of this being fulfilled for a randomly (i.e., independently and uniformly) chosen x_i can be bounded from below by $\frac{|K|-(i-1)\cdot|\Delta(M)|}{|K|}$. Hence, in case of a random sequence $x \in K^L$, the probability of $\{x\}$ being diff.disjoint from M is at least

$$\frac{(|K| - |\Delta(M)|) \cdot (|K| - 2|\Delta(M)|) \cdot \ldots \cdot (|K| - (L-1)|\Delta(M)|)}{|K|^{L-1}}$$
$$= \left(1 - \frac{1}{|K| / |\Delta(M)|}\right) \cdot \left(1 - \frac{2}{|K| / |\Delta(M)|}\right) \cdot \ldots \cdot \left(1 - \frac{L-1}{|K| / |\Delta(M)|}\right)$$
$$= \prod_{i=1}^{L-1}\left(1 - \frac{i}{|K| / |\Delta(M)|}\right).$$

By applying the above-mentioned approximations, we obtain that the lower bound on the probability of $\{x\}$ being diff.disjoint from M is around

$$e^{-\frac{L^2}{2(|K|/|\Delta(M)|)}} \approx 1 - \frac{L^2}{2(|K| / |\Delta(M)|)} = 1 - \frac{L^2 \cdot |\Delta(M)|}{2|K|}.$$

As $|\Delta(M)| \leq \binom{L}{2}$, an even coarser approximation can be given by

$$1 - \frac{L^2 \cdot \binom{L}{2}}{2|K|} = 1 - \frac{L^2 \cdot \frac{L\cdot(L-1)}{2}}{2|K|} > 1 - \frac{L^4}{4|K|},$$

which proves the lemma. □

Now let $A \in_U K^{n \times L}$ be a random $(n \times L)$-matrix over K. Similarly to the argument in the previous proof, we can learn that with probability around $1 - \frac{L^2}{2|K|}$, the first row of A contains L different coefficients. In this particular case, it follows straightforwardly from Lemma 5.3 that for all j, $2 \leq j \leq n$, the lower bound on the probability of $\{A_{1,\cdot}\}$ and $\{A_{j,\cdot}\}$ being diff.disjoint can be approximated by $1 - \frac{L^4}{4|K|}$.

Consequently, if K satisfies

$$\frac{L^2}{2\,|K|} \leq \frac{L^4}{4\,|K|} \leq \frac{1}{dn}, \tag{8}$$

then due to the implication

$$\left(1 - \frac{1}{dn}\right)^n \leq \left(1 - \frac{L^2}{2\,|K|}\right) \cdot \left(1 - \frac{L^4}{4\,|K|}\right)^{n-1},$$

in conjunction with Lemmata 5.2 and 5.3, the probability that A is $sgt\,(2)$-identifiable can be bounded from below by approximately

$$\left(1 - \frac{1}{dn}\right)^n \approx e^{-\frac{1}{d}} \approx 1 - \frac{1}{d}.$$

Observing that relation (8) holds if

$$|K| \geq \frac{1}{4} \cdot (dn) \cdot L^4,$$

i.e.,

$$a \geq \log(n) + \log(d) + 4\log(L) - 2,$$

completes the proof of Theorem 5. □

B On attacking the $(n, k, L)^+$-protocol by solving RandomSelect (L, n, a)

The following outline of an attack on the $(n, k, L)^+$-protocol by Krause and Stegemann [18] is meant to exemplify the immediate connection between the previously introduced learning problem RandomSelect (L, n, a) and the security of this whole new class of lightweight authentication protocols. Similar to the basic communication mode described in the introduction, the $(n, k, L)^+$-protocol is based on L n-dimensional, injective linear functions $F_1, \ldots, F_L : GF(2)^n \longrightarrow GF(2)^{n+k}$ (i.e., the secret key) and works as follows.

Each instance is initiated by the verifier Alice, who chooses a random vector $a \in_U GF(2)^{n/2}$ and sends it to Bob, who then randomly (i.e., independently and

uniformly) chooses $l \in_U [L]$ along with an additional value $b \in_U GF(2)^{n/2}$, in order to compute his response $w = F_l(a,b)$. Finally, Alice accepts $w \in GF(2)^{n+k}$ if there is some $l \in [L]$ with $w \in V_l$ and the prefix of length $n/2$ of $F_l^{-1}(w)$ equals a, where V_l denotes the n-dimensional linear subspace of $GF(2)^{n+k}$ corresponding to the image of F_l.

This leads straightforwardly to a problem called *Learning Unions of L Linear Subspaces* (LULS), where an oracle holds the specifications of L secret n-dimensional linear subspaces $V_1, \ldots, V_L$ of $GF(2)^{n+k}$, from which it randomly chooses examples $v \in_U V_l$ for $l \in_U [L]$ and sends them to the learner. Knowing only n and k, he seeks to deduce the specifications of $V_1, \ldots, V_L$ from a sufficiently large set $\{w_1, \ldots, w_s\} \subseteq \bigcup_{l=1}^{L} V_l$ of such observations. It is easy to see that this corresponds to a passive key recovery attack against (n,k,L)-type protocols. Note that there is a number of exhaustive search strategies to solve this problem, e.g., the generic exponential time algorithm called search-for-a-basis heuristic, which was presented in the appendix of [18].

It should be noted that an attacker who is able to solve the LULS problem needs to perform additional steps to fully break the $(n,k,L)^+$-protocol as impersonating the prover requires to send responses $w \in GF(2)^{n+k}$ which not only fulfill $w \in \bigcup_{l=1}^{L} V_l$ but also depend on some random nonce $a \in GF(2)^{n/2}$ provided by the verifier. However, having successfully obtained the specifications of the secret subspaces $V_1, \ldots, V_L$ allows in turn for generating a specification of the image of $F_l(a,\cdot)$ for each $l \in [L]$ by repeatedly sending an arbitrary but fixed (i.e., selected by the attacker) $a \in GF(2)^{n/2}$ to the prover. Remember that, although the prover chooses a random $l \in_U [L]$ each time he computes a response w based on some fixed a, an attacker who has determined $V_1, \ldots, V_L$ will know which subspace the vector w actually belongs to. Krause and Stegemann pointed out that this strategy allows for efficiently constructing specifications of linear functions $G_1, \ldots, G_L : GF(2)^n \longrightarrow GF(2)^{n+k}$ and bijective linear functions $g_1, \ldots, g_L : GF(2)^{n/2} \longrightarrow GF(2)^{n/2}$ such that

$$F_l(a,b) = G_l(a, g_l(b))$$

for all $l \in [L]$ and $a, b \in GF(2)^{n/2}$ [18]. Hence, the efficiently obtained specifications of the functions $((G_1, \ldots, G_L), (g_1, \ldots, g_L))$ are equivalent to the actual secret key $(F_1, \ldots, F_L)$. However, keep in mind that the running time of this attack is dominated by the effort needed to solve the LULS problem first and that $\mathsf{RandomSelect}(L,n,a)$ in fact refers to a special case of the LULS problem, which assumes that the secret subspaces have the form

$$V_l = \{(v, f_l(v)) \mid v \in GF(2)^n\} \subseteq GF(2)^{n+k}$$

for all $l \in [L]$ and secret $GF(2)$-linear functions $f_1, \ldots, f_L : GF(2)^n \longrightarrow GF(2)^k$. This is true with probability $p(n) \approx 0.2887$ as, given an arbitrary $((n+k) \times n)$-matrix A over $GF(2)$, the general case $V = \{A \circ v \mid v \in GF(2)^n\}$ can be written in the special form iff the first n rows of A are linearly independent (see, e.g., [11]).

In order to solve this special problem efficiently, we suggest the following approach, which makes use of our learning algorithm for RandomSelect (L, n, a) and works by

- determining an appropriate number $a \in O(\log(n))$ which, w.l.o.g., divides k (i.e., $k = \gamma \cdot a$ for some $\gamma \in \mathbb{N}$),
- identifying vectors $w \in \{0,1\}^k$ with vectors $w = (w_1, \ldots, w_\gamma) \in GF(2^a)^\gamma$ and functions $f : \{0,1\}^n \longrightarrow \{0,1\}^k$ with γ-tuples $\left(f^1, \ldots, f^\gamma\right)$ of component functions $f^1, \ldots, f^\gamma : \{0,1\}^n \longrightarrow GF(2^a)$ based on the following rule: $f^i(u) = w_i$ for all $i = 1, \ldots, \gamma$ if and only if $f(u) = (w_1, \ldots, w_\gamma)$,
- learning $f_1, \ldots, f_L : \{0,1\}^n \longrightarrow \{0,1\}^k$ by learning each of the corresponding sets of component functions $f_1^i, \ldots, f_L^i : \{0,1\}^n \longrightarrow GF(2^a)$ in time $n^{O(L)}$ for $i = 1, \ldots, \gamma$.

Clearly, for efficiency reasons, a should be as small as possible. However, in section 4 we show that a needs to exceed a certain threshold, which can be bounded from above by $O(\log(n))$, to enable our learning algorithm to find a unique solution with high probability.

Please note that, throughout this paper, a is assumed to be fixed as we develop a learning algorithm for sets of secret $GF(2)$-linear functions $f_1, \ldots, f_L : \{0,1\}^n \longrightarrow K$, where $K = GF(2^a)$. In particular, for the sake of simplicity, we write $f_1, \ldots, f_L$ while actually referring to a set of component functions as explained above.

TWINE: A Lightweight, Versatile Block Cipher

Tomoyasu Suzaki, Kazuhiko Minematsu, Sumio Morioka, and Eita Kobayashi

NEC Corporation, 1753 Shimonumabe, Nakahara-Ku, Kawasaki, Japan
t-suzaki@cb.jp.nec.com, k-minematsu@ah.jp.nec.com,
s-morioka@ak.jp.nec.com, e-kobayashi@fg.jp.nec.com

Abstract. This paper presents a 64-bit lightweight block cipher TWINE supporting 80 and 128-bit keys. It enables quite small hardware implementation similar to the previous proposals, yet enables efficient implementations on embedded software. Moreover, it allows a compact implementation of unified encryption and decryption. This characteristic mainly originates from the use of generalized Feistel with many sub-blocks combined with a recent improvement on the diffusion layer.
Keywords: lightweight block cipher, generalized Feistel, block shuffle

1 Introduction

Recent advances in tiny computing devices, such as RFID and sensor network nodes, give rise to the need of symmetric encryption with highly-limited resources, called lightweight encryption. While we have AES it is often inappropriate for such devices due to their size/power/memory constraints, even though there are constant efforts for small-footprint AES, e.g., [14, 31, 37]. To fill the gap, a number of hardware-oriented lightweight block ciphers have been recently proposed; for instance, DESL [26], HIGHT [21], PRESENT [9], and KATAN/KTANTAN [13], PRINTcipher [25], and many more.

In this paper, we propose a new lightweight 64-bit block cipher TWINE. It supports 80 and 128-bit keys. Our purpose is to achieve hardware efficiency while minimizing the hardware-oriented design choices, such as a bit permutation. The avoidance of such options may be beneficial to software implementation and yield a balanced performance on both software and hardware. Lightweight blockciphers from a similar motivation are also seen in, e.g., KLEIN [19], LBlock [43], and most recently, LED [18] and Piccolo [40].

For this purpose, we employ Type-2 generalized Feistel structure (GFS) proposed[1] by Zheng et al. [44] with 16 nibble-blocks. The drawback of such design is poor (slow) diffusion property, leading to a slow cipher with quite many rounds. To overcome the problem, we employ the idea of Suzaki and Minematsu at FSE '10 [41] which substantially improves diffusion by using a different block shuffle from the original (cyclic shift). As a result, TWINE is also efficient on (embedded) software and enables compact unification of encryption and decryption.

[1] Zheng et al. called it Type-2 Feistel-Type Transformation. The generalized Feistel structure is an alias taken by, e.g., [39, 41].

The features of our proposal are (1) no bit permutation, (2) generalized Feistel-based, and (3) no Galois-Field matrix. The components are only one 4-bit S-box and 4-bit XOR. As far as we know, this is the first attempt that combines these three features. There is a predecessor called LBlock [43] which has some resemblances to ours, however TWINE is an independent work and has several concrete design advantages (See Section 3).

We implemented TWINE on hardware (ASIC and FPGA) and software (8-bit microcontroller). We did not take the fixed key setting, hence keys can be updated. Our hardware implementations suggest that the encryption-only TWINE can be implemented with about $1,500$ Gate Equivalent (GE), and when encryption and decryption are unified, it can be implemented within $1,800$ GEs. We are also trying a serialized implementation and a preliminary result suggests $1,116$ GEs. For software, TWINE is implemented within 0.8 to 1.5 Kbytes ROM. The speed is relatively fast compared to other lightweight ciphers. Though the hardware size is not the smallest, we think the performance of TWINE is well-balanced for both hardware and software.

For security, TWINE employs a technique to enhance the diffusion of GFS, however, it is definitely important to evaluate the security against attacks that exploit the diffusion property of generalized Feistel, such as the impossible differential attack and the saturation attack. We perform a thorough analysis (for a new cipher proposal) on TWINE and present the impossible differential attack against 23-round TWINE-80 and 24-round TWINE-128 as the most powerful attacks we have found so far. The attack is ad-hoc and fully exploits the key schedule, which can be of independent interest as an example of highly-optimized impossible differential attack against GFS-based ciphers.

The organization of the paper is as follows. In Section 2 we describe the specification of TWINE. Section 3 explains the design rationale for TWINE. In Section 4 we present the result of security evaluation, and in section 5 we present the implementation results of both hardware and software. Section 6 concludes the paper.

2 Specification of TWINE

2.1 Notations

The basic mathematical operations we use are as follows. $\oplus$ denotes bitwise exclusive-OR. For binary strings, x and y, $x \| y$ denotes the concatenation of x and y. Let $|x|$ denote the bit length of a binary string x. If $|x| = m$, x is also written as $x_{(m)}$ to emphasize its bit length. If $|x| = 4c$ for some positive integer c, we write $x \rightarrow (x_0 \| x_1 \| \ldots \| x_{c-1})$, where $|x_i| = 4$, is the partition operation into the 4-bit subsequences. The opposite operation, $(x_0 \| x_1 \| \ldots \| x_{c-1}) \rightarrow x$, is similarly defined. The partition operation may be implicit, i.e., we may simply write x_i to denote the i-th 4-bit subsequence for any $4c$-bit string x.

2.2 Data Processing Part (Encryption Process)

TWINE is a 64-bit block cipher with two supported key lengths, 80 and 128 bits. If the key length is needed to be specified, we write TWINE-80 or TWINE-128 to denote the corresponding version. The global structure of TWINE is a variant of Type-2 GFS [44] [38] with 16 4-bit sub-blocks. Given a 64-bit plaintext, $P_{(64)}$, and a round key, $RK_{(32\times 36)}$, the cipher produces the ciphertext $C_{(64)}$. Round key $RK_{(32\times 36)}$ is derived from the secret key, $K_{(n)}$ with $n \in \{80, 128\}$, using the key schedule. A round function of TWINE consists of a nonlinear layer using 4-bit S-boxes and a diffusion layer, which permutes the 16 blocks. Unlike Type-2 GFS, the diffusion layer is not a circular shift and is designed to provide a better diffusion that the circular shift, according the result of [41]. This round function is iterated for 36 times for both key lengths, where the diffusion layer of the last round is omitted. The encryption process can be written as Algorithm 2.1.

The S-box, S, is a 4-bit permutation defined as Table 1. The permutation of block indexes, $\pi : \{0, \ldots, 15\} \to \{0, \ldots, 15\}$, where j-th sub-block (for $j = 0, \ldots, 15$) is mapped to $\pi[j]$-th sub-block, is depicted at Table 2.

The figure of the round function is in Fig. 1.

Algorithm 2.1: TWINE.Enc($P_{(64)}, RK_{(32\times 36)}, C_{(64)}$)

$X^1_{(64)} \leftarrow P$
$RK^1_{(32)} \| \ldots \| RK^{35}_{(32)} \leftarrow RK_{(32\times 36)}$
for $i \leftarrow 1$ **to** 35
do $\begin{cases} X^i_{0(4)} \| X^i_{1(4)} \| \ldots \| X^i_{14(4)} \| X^i_{15(4)} \leftarrow X^i_{(64)} \\ RK^i_{0(4)} \| RK^i_{1(4)} \| \ldots \| RK^i_{6(4)} \| RK^i_{7(4)} \leftarrow RK^i_{(32)} \\ \textbf{for } j \leftarrow 0 \textbf{ to } 7 \\ \quad \textbf{do } X^i_{2j+1} \leftarrow S(X^i_{2j} \oplus RK^i_j) \oplus X^i_{2j+1} \\ \textbf{for } h \leftarrow 0 \textbf{ to } 15 \\ \quad \textbf{do } X^{i+1}_{\pi[h]} \leftarrow X^i_h \\ X^{i+1} \leftarrow X^{i+1}_0 \| X^{i+1}_1 \| \ldots \| X^{i+1}_{14} \| X^{i+1}_{15} \end{cases}$
for $j \leftarrow 0$ **to** 7
  do $X^{36}_{2j+1} \leftarrow S(X^{36}_{2j} \oplus RK^{36}_j) \oplus X^{36}_{2j+1}$
$C \leftarrow X^{36}$

Table 1. S-box Mapping in the Hexadecimal Notation.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S(x)$	C	0	F	A	2	B	9	5	8	3	D	7	1	E	6	4

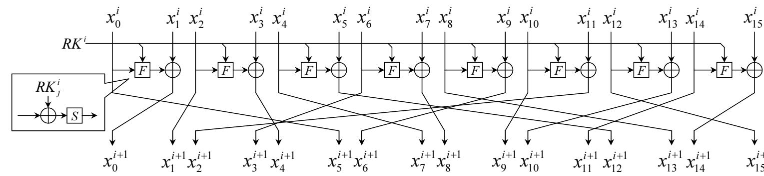


Fig. 1. Round function of TWINE. Data path is 4-bit and each F function is of the form $y = S(x \oplus k)$ with a 4-bit S-box.

Table 2. Block Shuffle π and its Inverse π^{-1}.

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[j]$	5	0	1	4	7	12	3	8	13	6	9	2	15	10	11	14
$\pi^{-1}[j]$	1	2	11	6	3	0	9	4	7	10	13	14	5	8	15	12

2.3 Key Schedule Part

The key schedule produces $RK_{(32\times36)}$ from the secret key, $K_{(n)}$, where $n \in \{80, 128\}$. As well as the data processing part, it is a variant of GFS but with much sparser nonlinear functions. The pseudocode of key schedule for 80-bit key is in Algorithm 2.2 and the figure is in Appendix C. For 128-bit key, see Appendix A. Round constant, $\mathrm{CON}^i_{(6)} = \mathrm{CON}^i_{H(3)} \| \mathrm{CON}^i_{L(3)}$, is defined as 2^i in $\mathrm{GF}(2^6)$ with primitive polynomial $z^6 + z + 1$. The exact values are listed at Table 3.

Table 3. Round Constants. CON^i is the rightmost 6-bit (of 8 bits expressed in hexadecimal notation).

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
CON^i	01	02	04	08	10	20	03	06	0C	18	30	23	05	0A	14	28	13	26

i	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
CON^i	0F	1E	3C	3B	35	29	11	22	07	0E	1C	38	33	25	09	12	24	0B

Algorithm 2.2: TWINE.KeySchedule-80($K_{(80)}, RK_{(32\times 36)}$)

$$
\begin{array}{l}
\mathrm{WK}_{(80)} \leftarrow K \\
\mathrm{WK}_{0(4)} \| \mathrm{WK}_{1(4)} \| \ldots \| \mathrm{WK}_{18(4)} \| \mathrm{WK}_{19(4)} \leftarrow WK \\
\mathrm{RK}^1_{0(4)} \leftarrow \mathrm{WK}_1,\ \ \mathrm{RK}^1_{1(4)} \leftarrow \mathrm{WK}_3,\ \ \mathrm{RK}^1_{2(4)} \leftarrow \mathrm{WK}_4,\ \ \mathrm{RK}^1_{3(4)} \leftarrow \mathrm{WK}_6 \\
\mathrm{RK}^1_{4(4)} \leftarrow \mathrm{WK}_{13},\ \mathrm{RK}^1_{5(4)} \leftarrow \mathrm{WK}_{14},\ \mathrm{RK}^1_{6(4)} \leftarrow \mathrm{WK}_{15},\ \mathrm{RK}^1_{7(4)} \leftarrow \mathrm{WK}_{16} \\
\mathrm{RK}^1_{(32)} \leftarrow \mathrm{RK}^1_0 \| \mathrm{RK}^1_1 \| \ldots \| \mathrm{RK}^1_6 \| \mathrm{RK}^1_7 \\
\textbf{for } i \leftarrow 2 \textbf{ to } 36 \\
\quad \textbf{do} \begin{cases}
\mathrm{WK}_1 \leftarrow \mathrm{WK}_1 \oplus S(\mathrm{WK}_0) \\
\mathrm{WK}_4 \leftarrow \mathrm{WK}_4 \oplus S(\mathrm{WK}_{16}) \\
\mathrm{WK}_7 \leftarrow \mathrm{WK}_7 \oplus 0 \| CON_H^{i-1} \\
\mathrm{WK}_{19} \leftarrow \mathrm{WK}_{19} \oplus 0 \| CON_L^{i-1} \\
\mathrm{tmp}_0 \leftarrow \mathrm{WK}_0,\ \mathrm{tmp}_1 \leftarrow \mathrm{WK}_1,\ \mathrm{tmp}_2 \leftarrow \mathrm{WK}_2,\ \mathrm{tmp}_3 \leftarrow \mathrm{WK}_3 \\
\textbf{for } j \leftarrow 0 \textbf{ to } 3 \\
\quad \textbf{do} \begin{cases} \mathrm{WK}_{j*4} \leftarrow \mathrm{WK}_{j*4+4}, & \mathrm{WK}_{j*4+1} \leftarrow \mathrm{WK}_{j*4+5} \\ \mathrm{WK}_{j*4+2} \leftarrow \mathrm{WK}_{j*4+6}, & \mathrm{WK}_{j*4+3} \leftarrow \mathrm{WK}_{j*4+7} \end{cases} \\
\mathrm{WK}_{16} \leftarrow \mathrm{tmp}_1, \mathrm{WK}_{17} \leftarrow \mathrm{tmp}_2, \mathrm{WK}_{18} \leftarrow \mathrm{tmp}_3, \mathrm{WK}_{19} \leftarrow \mathrm{tmp}_0 \\
\mathrm{RK}^i_0 \leftarrow \mathrm{WK}_1,\ \ \mathrm{RK}^i_1 \leftarrow \mathrm{WK}_3,\ \ \mathrm{RK}^i_2 \leftarrow \mathrm{WK}_4,\ \ \mathrm{RK}^i_3 \leftarrow \mathrm{WK}_6 \\
\mathrm{RK}^i_4 \leftarrow \mathrm{WK}_{13},\ \mathrm{RK}^i_5 \leftarrow \mathrm{WK}_{14},\ \mathrm{RK}^i_6 \leftarrow \mathrm{WK}_{15},\ \mathrm{RK}^i_7 \leftarrow \mathrm{WK}_{16} \\
\mathrm{RK}^i_{(32)} \leftarrow \mathrm{RK}^i_{0(4)} \| \mathrm{RK}^i_{1(4)} \| \ldots \| \mathrm{RK}^i_{6(4)} \| \mathrm{RK}^i_{7(4)}
\end{cases} \\
RK_{(32\times 36)} \leftarrow \mathrm{RK}^1_{(32)} \| \mathrm{RK}^2_{(32)} \| \ldots \| \mathrm{RK}^{35}_{(32)} \| \mathrm{RK}^{36}_{(32)}
\end{array}
$$

2.4 Decryption Process

The decryption of TWINE is quite similar to the encryption; we use the same S-box and key schedule as used in the encryption, with the inverse block shuffle. See Algorithm 2.3.

Algorithm 2.3: TWINE.Dec($C_{(64)}, RK_{(32\times 36)}, P_{(64)}$)

$$
\begin{array}{l}
X^{36}_{(64)} \leftarrow C \\
\mathrm{RK}^0_{(32)} \| \ldots \| \mathrm{RK}^{35}_{(32)} \leftarrow RK_{(32\times 36)} \\
\textbf{for } i \leftarrow 36 \textbf{ to } 2 \\
\quad \textbf{do} \begin{cases}
X^i_{0(4)} \| X^i_{1(4)} \| \ldots \| X^i_{14(4)} \| X^i_{15(4)} \leftarrow X^i_{(64)} \\
\mathrm{RK}^i_{0(4)} \| \mathrm{RK}^i_{1(4)} \| \ldots \| \mathrm{RK}^i_{6(4)} \| \mathrm{RK}^i_{7(4)} \leftarrow \mathrm{RK}^i_{(32)} \\
\textbf{for } j \leftarrow 0 \textbf{ to } 7 \\
\quad \textbf{do } X^i_{2j+1} \leftarrow S(X^i_{2j} \oplus \mathrm{RK}^i_j) \oplus X^i_{2j+1} \\
\textbf{for } h \leftarrow 0 \textbf{ to } 15 \\
\quad \textbf{do } X^{i-1}_{\pi^{-1}[h]} \leftarrow X^i_h \\
X^{i-1} \leftarrow X^{i-1}_0 \| X^{i-1}_1 \| \ldots \| X^{i-1}_{14} \| X^{i-1}_{15}
\end{cases} \\
\textbf{for } j \leftarrow 0 \textbf{ to } 7 \\
\quad \textbf{do } X^1_{2j+1} \leftarrow S(X^1_{2j} \oplus \mathrm{RK}^1_j) \oplus X^1_{2j+1} \\
P \leftarrow X^1
\end{array}
$$

3 Design Rationale

3.1 Basic Objective

We focus on the mixed environments of resource-constrained hardware and software, and aim at building a block cipher with a balanced performance under such environments. Specifically, our goals are

1. small footprint in hardware implementation (e.g. under 2,000 GE [23, 33]),
2. small ROM/RAM consumption in software implementation,
3. these goals achieved for the unified encryption/decryption functionality.

The importance of the last item is also pointed out by [1].

On LBlock. We remark that LBlock [43], proposed independently of ours, is quite similar to our proposal. It is a 64-bit block cipher based on the balanced Feistel whose round function consists of 8 4-bit S-boxes followed by a 4-bit block-wise permutation (hence no matrix operation as used by SPN). LBlock also performs 8-bit rotation to the round function's output. Such a structure can be transformed into a further generalized Type-2 GFS proposed by [41], though we do not know whether the authors of LBlock are aware of it. We investigated LBlock in this respect and found that the LBlock's diffusion layer is equivalent to that of the decryption of our proposal. Note that this choice is quite reasonable from Table 6 of [41], as it satisfies both of the fastest diffusion and the highest immunities against linear and differential attacks among other block shuffles.

Nevertheless, there are some important differences between TWINE and LBlock, as follows;

1. LBlock uses ten distinct S-boxes while ours uses one S-box. The use of single S-box rather than multiple ones can contribute to smaller (serialized) hardware and software implementations.
2. LBlock uses a bit permutation in its key scheduling, while ours is completely bit permutation-free, including the key schedule. Hence the design of LBlock does not meet our criteria mentioned at Introduction.

We also would like to point out that the security evaluation of LBlock is insufficient. We already found a saturation attack against 22-round version without considering the key schedule, thus the security margin is smaller than the claimed by the authors (20-round). Using the techniques presented at Section 4, we expect further improvements on the attack.

3.2 Parameters and Components

Considering the basic design goals as above, we choose the 64-bit block size with 80 and 128-bit keys, which is compatible to many previous lightweight blockciphers. The number of rounds is determined from our security analysis. As far as we investigated, the most powerful attack against TWINE is a dedicated

impossible differential attack, which breaks 23-round TWINE-80 and 24-round TWINE-128. From this, we consider 36-round TWINE-128 has a sufficient security margin. When keeping the same size of margin for TWINE-80 and TWINE-128 is our sole goal, the 80-bit key version could reduce the number of rounds from 36. However, considering the security margin and the implementation merit (i.e. 36 has many factors, which enables various multiple-round hardware implementations with a small overhead), we employ the 36-round structure for both key lengths.

Block Shuffle. The block shuffle π comes from a result of FSE '10 [41]. In [41], it was reported that by changing the block shuffle different from the ordinal cyclic shift one can greatly improve the diffusion of Type-2 GFS. Here, goodness-of-diffusion is measured by the minimum number of rounds that diffuses any input sub-block difference to all output sub-blocks, called DRmax. Smaller DRmax means a faster diffusion. DRmax of cyclic shift with k sub-blocks is k, while there exist shuffles with DRmax $= 2\log_2 k$, called "optimum block shuffle" [41]. Our π is such one[2] with $k = 16$, hence DRmax $= 8$ while DRmax $= 16$ for the cyclic shift. DRmax is connected to the resistance against various attacks. For example, Type-2 GFS with 16 sub-blocks has 33-round impossible differential characteristics and 32-round saturation characteristics. However, by using π of Table 2 they can be reduced to 14 and 15 rounds.

There exist multiple optimum block shuffles [41]. Hence π was chosen considering other aspects which is not (directly) related to DRmax. In particular, we focus on the resistance against differential and linear cryptanalysis, i.e., the number of active S-boxes.

S-box. The 4-bit S-box is chosen to satisfy

1. The maximum differential and linear probabilities are 2^{-2}, which is theoretically the minimum for invertible S-box,
2. The boolean degree is 3,
3. The interpolation polynomial contains many terms and has degree 14.

Following the AES S-box design, we searched S-boxes satisfying the above while being representable as a composition of Galois field inversion and an affine transformation. More precisely, our S-box is defined as $y = S(x) = f((x \oplus b)^{-1})$, where a^{-1} denotes the inverse of a in GF(2^4) (the zero element is mapped to itself.) with irreducible polynomial $z^4 + z + 1$, and $b = 1$ is a constant, and $f(\cdot)$ is an affine function defined as

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 0\,0\,1\,1 \\ 1\,0\,0\,1 \\ 1\,0\,0\,0 \\ 0\,1\,0\,0 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad (1)$$

for $y = f(x)$ with $y = (y_0 \| y_1 \| y_2 \| y_3)$ and $x = (x_0 \| x_1 \| x_2 \| x_3)$.

[2] More precisely, an isomorphic shuffle in Appendix B ($k = 16$, No. 10) of [41].

Key Schedule. The key schedule has many design options. We choose one which enables (1) on-the-fly operations and (2) produces each round key via sequential update of a key state, that is, there is no intermediate key. As mentioned, it uses no bit permutation. As hardware efficiency is not our ultimate goal, the design is rather conservative compared to the recent hardware-oriented ones [13,34,40], yet quite simple. For security, we want our key schedule to have sufficient resistance against slide, meet-in-the-middle, and related-key attacks.

4 Security Evaluation

4.1 Overview

We examined the security of TWINE against various attacks for both 80 and 128-bit keys. Since it is hard to describe all the evaluation results due to the page limit, we focus on the most critical attacks in our evaluation; the impossible differential and saturation attacks. For simplicity we only describe the details of the attacks against TWINE-80; the results on TWINE-128 will be briefly described in the summary. The short summary on other attacks, such as differential and linear attacks, will also be given.

In this section, we use the following notations. $\bar{S}^c$ denotes the sequence of c symbols S, e.g. $\bar{0}^3$ means $(0,0,0)$ and $\bar{U}^3$ means (U,U,U). The F function in the i-th round is labeled as $F^i_0,\dots,F^i_{15}$, where F^i_0 is the leftmost one. We let $\mathrm{RK}^i_{[j_1,\dots,j_h]}$ to denote the vector $(\mathrm{RK}^i_{j_1},\dots,\mathrm{RK}^i_{j_h})$. Since RK^i_j is the j-th 4-bit subsequence of RK^i (for $j=0,1,\dots,15$), this means $F^i_j(x) = S(\mathrm{RK}^i_j \oplus x)$. $X^i_{[j_1,\dots,j_h]}$ is similarly defined.

4.2 Impossible Differential Attack

Generally, impossible differential attack [6] is one of the most powerful attack against Feistel and GFS-based ciphers, as demonstrated by (e.g.) [15, 32, 42]. We searched impossible differential characteristics (IDCs) using Kim et al.'s method [24], and found 64 14-round IDCs. We here present a highly-optimized attack against 23-round TWINE-80. It exploits the key schedule and is based on the following 14-round IDC (for 4-bit blocks);

$$(0,\alpha,\bar{0}^{14}) \overset{14r}{\nrightarrow} (\bar{0}^8,\beta,\bar{0}^7),\ \text{where } \alpha \neq 0 \text{ and } \beta \neq 0. \qquad (2)$$

Our attack uses the above IDC to 5-th to 18-th rounds of TWINE, and tries to recover the subkeys of the first 4 rounds and last 5 rounds, 144 bits in total. These subkey bits are uniquely determined via its 80-bit subsequence; see Appendix D.

The details of our attack are as follows.

Data Collection. We call a set of 2^{32} plaintexts a *structure* if its i-th sub-blocks are fixed to a constant for all $i = 2,4,5,6,7,8,9,14 \in \{0,\dots,15\}$ and the remaining 8 sub-blocks take all 2^{32} values. Suppose we have one structure. From it we extract plaintext pairs having the differential

$$(\alpha_1,\alpha_2,0,\alpha_3,\bar{0}^6,\alpha_4,\alpha_5,\alpha_6,\alpha_7,0,\alpha_0), \qquad (3)$$

where α_i is a non-zero 4-bit value. For such a plaintext pair, we want to make sure that the differential of the internal state after the first 4 rounds to match the left hand side of Eq. (2). For this, the output differentials with respect to some F functions (in the first 4 rounds) have to be canceled out. For example, α_2 must be the differential of F with input differential α_1, as shown by Fig. 2. Here, we use the following observation;

Proposition 1. *Let $y = F^i_j(x) = S(\mathrm{RK}^i_j \oplus x)$ and $y' = F^i_j(x')$. For fixed $\Delta_x = x \oplus x' \neq 0$, $\Delta_y = y \oplus y'$ has always 7 possible values, for any i and j. Moreover, for a fixed $\Delta_x \neq 0$ let $\tau(\Delta_y)$ be the function of Δ_y which represents the number of possible RK^i_j values. Then $\tau(\Delta_y)$ equals 2 for some 6 values of Δ_y and 4 for the remaining one.*

Hence we have a set of 7 possible values for α_2, which is determined by α_1. Considering this restriction, we can extract $2^{54.56}$ plaintext pairs from a structure with its differential being Eq (3).

Key Elimination. After the plaintext pairs have been generated, we encrypt them and seek the ciphertext pairs with a differential

$$(0, \beta_1, 0, \beta_2, \beta_3, \beta_4, \beta_0, \beta_5, \beta_6, \beta_7, \beta_8, \beta_9, \beta_{10}, \beta_{11}, 0, 0),\ {}^{\forall}\beta_i \neq 0. \tag{4}$$

For each ciphertext pair with differential Eq. (4), we try to eliminate the wrong (impossible) candidates for the 80-bit (sub)key vector $(\mathcal{K}_1 \| \mathcal{K}_2 \| \mathcal{K}_3)$, where

$$\begin{aligned} &\mathcal{K}_1 = (\mathrm{RK}^1_{[1,2,3,7]}, \mathrm{RK}^{23}_0), \mathcal{K}_2 = (\mathrm{RK}^1_{[0,5,6]}, \mathrm{RK}^2_{[2,4,6,7]}, \mathrm{RK}^{23}_{[2,4,5]}\mathrm{RK}^{22}_{[1,3,4]}),\\ &\mathcal{K}_3 = (\mathrm{RK}^{22}_{0,2}), \end{aligned} \tag{5}$$

using the plaintext pair of differential Eq. (3) and the ciphertext pair of differential Eq. (4). This can be done as follows. First, we guess the 20-bit $\mathcal{K}_1$ (which can take all possible values). After $\mathcal{K}_1$ is guessed, the number of each 4-bit subkey candidates in $\mathcal{K}_2$ is $(2 \cdot 6 + 4)/7 \approx 2.28$ on average from Proposition 1. Moreover, once $\mathcal{K}_1$ and $\mathcal{K}_2$ are fixed, each 4-bit subkey of $\mathcal{K}_3$ will have $(2 \cdot 6 + 4)/15 \approx 1.07$ candidates, as we have no restrictions on the input differential for Fs relating to these subkeys. From this observation, we can expect to eliminate $2^{20} \cdot (16/7)^{13} \cdot (16/15)^2 \approx 2^{35.69}$ candidates from a set of 2^{80} values for each plaintext-ciphertext pair, i.e. a guess of 80-bit $(\mathcal{K}_1 \| \mathcal{K}_2 \| \mathcal{K}_3)$ is eliminated with probability $2^{-44.31}$. The detail of the above procedure is depicted at Table 12 in Appendix D.

From this, to determine $(\mathcal{K}_1 \| \mathcal{K}_2 \| \mathcal{K}_3)$ with probability almost one, we need N ciphertext pairs, where N satisfies $2^{80}(1 - 2^{-44.31})^N \approx 1$. This implies $N \approx 2^{50.11}$. Assuming the ciphertext's randomness, we can expect a ciphertext pair of differential Eq (4) with probability $(2^{-4})^4 \cdot (2^{-1})^8 = 2^{-24}$. This implies that we basically need $2^{74.11}$ ciphertext pairs. But in fact we need some more. In the key elimination we need to compute some other subkeys (64 bits in total), which is uniquely determined by the key of Eq. (5). These keys contain RK^{19}_4,RK^{21}_4, and RK^{23}_6 and they can cause a contradiction with other keys. If this event occurs,

the corresponding plaintext/ciphertext pair turns out to be useless. Considering the probability of this event we need 2^{10} times more pairs, thus we eventually need $2^{84.11}$ ciphertext pairs.

Since one structure enables to produce $2^{54.56}$ plaintext pairs of the desired difference, we need to generate $2^{29.55}$ structures (by using $2^{29.55}$ distinct constants) and run the above key elimination procedure for all structures.

Details of Key Elimination. In the key elimination we combine several techniques to reduce the complexity. In particular we use the Difference Table. Its entry is indexed by $(x, x', y) \in (\{0,1\}^4)^3$ and the entry is a set $\mathcal{K} = \{k : y = S(k \oplus x) \oplus S(k \oplus x')\}$. We also use the relationships between subkeys induced from the key schedule, or we can directly guess the key if input and output pairs of the corresponding F are fixed (not only their differentials).

Complexity Estimation. For each plaintext-ciphertext pair, the procedure regarding $\mathcal{K}_2$ and $\mathcal{K}_3$ requires the 17 evaluations of F function, shown on the dotted lines in Fig.s 2 and 3, and $F^{23}_{[2,3,4,5,6]}$, total 22 functions. This amounts to $22/(23 \cdot 8)$ encryptions of 22-round TWINE. Consequently, we can attack 23-round TWINE-80 with the time complexity $2^{50.11+10} \cdot 2^{20} \cdot 22/(23 \cdot 8) = 2^{77.04}$ encryptions, and the memory complexity $2^{80}/64 = 2^{74}$ blocks.

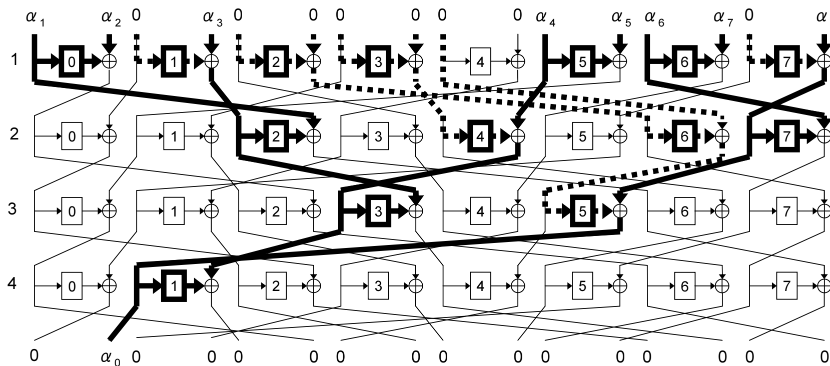


Fig. 2. The First 4 Rounds in the Impossible Differential Attack.

4.3 Saturation Attack

Saturation attack [16] is also a powerful attack against GFS-based ciphers. We consider 4-bit-wise saturations. The state consists of 2^4 variables, denoted by $S = (S_0, \ldots, S_{15})$, where S_i has the following four status (here X is the plaintext):

Constant (C) : $\forall i, j \;\; X_i = X_j$ **All** (A) : $\forall i, j \;\; i \neq j \Leftrightarrow X_i \neq X_j$

Balance (B) : $\bigoplus_i^{2^4-1} X_i = 0$ **Unknown** (U) : Others

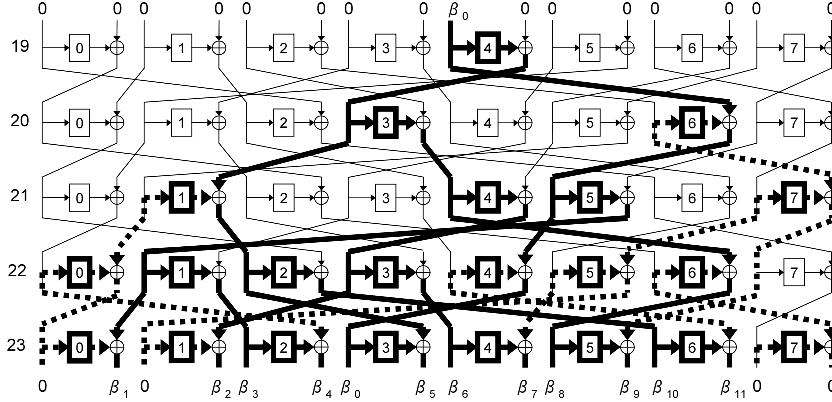


Fig. 3. The Last 5 Rounds in the Impossible Differential Attack.

Let $\alpha = (\alpha_0, \ldots, \alpha_{15})$ and $\beta = (\beta_0, \ldots, \beta_{15})$, $\alpha_i, \beta_i \in \{C, A, B, U\}$, be the initial state and the t-round state holding with probability 1. If we have $\alpha_i = A$ and $\beta_j \neq U$ for some i and j, we call $\alpha \xrightarrow{tr} \beta$ an t-round saturation characteristic (SC).

TWINE has 15-round SC with input consisting of one C and fifteen As and output consisting of 4 Bs (the remaining consists of U), for example;

$$(\bar{A}^{12}, C, \bar{A}^3) \xrightarrow{15r} (\bar{U}^3, B, \bar{U}^5, B, \bar{U}^3, B, U, B), \quad (6)$$

$$(\bar{A}^{6}, C, \bar{A}^9) \xrightarrow{15r} (U, B, \bar{U}^3, B, U, B, \bar{U}^3, B, \bar{U}^4). \quad (7)$$

Suppose we use SC of Eq. (7) to break 22-round TWINE-80. Let S-*structure* denote a set of 2^{60} plaintexts induced from Eq. (7), i.e., the input block X_6 is fixed to a constant and the other blocks take all combinations.

Our attack recovers a 72-bit subkey vector

$$\mathcal{K}_{\text{target}} = (\mathrm{RK}^{22}, \mathrm{RK}^{21}_{[0,2,3,4,5,6,7]}, \mathrm{RK}^{20}_{[6,7]}, \mathrm{RK}^{16}_0)$$

based on the fact that (the state of) X^{15}_1 is B with these 15-round SCs. Here, the state of X^{15}_1 being B implies the coincidence of states between X^{16}_0 and F^{16}_0, the output of the leftmost F function in the round 16, computed from the S-structure. From this, we calculate the sums of F^{16}_0 and X^{16}_0 independently, and choose a key that makes these values the same as a candidate for the correct key. The basic procedure for one S-structure is as follows.

1. Encrypt an S-structure and obtain 2^{60} ciphertexts. As our target is the 22-round version, the ciphertext is written as X^{22}.
2. List all ciphertexts except their rightmost (16-th) block, X^{22}_{15}. The result is denoted by List $\mathcal{L}_1$. It is merely a set of 60-bit values and the values with even appearances need not be stored. We then guess

$$\mathcal{K}_1 = (\mathrm{RK}^{22}_{[0,1,2,3,4,5,6]}, \mathrm{RK}^{21}_{[2,3,4,5,7]}, \mathrm{RK}^{20}_6, \mathrm{RK}^{17}_2, \mathrm{RK}^{16}_0)$$

and compute the sum of outputs of F_0^{16} (the leftmost F function in the round 16) using all entries in $\mathcal{L}_1$ with each guess for $\mathcal{K}_1$[3]. We remark that a pair of an entry of $\mathcal{L}_1$ and a guess for $\mathcal{K}_1$ uniquely determines the output of F_0^{16}. The key guesses are grouped according to the sum of F_0^{16}'s outputs. Let $\mathcal{G}_1(s)$ be the key group with F_0^{16} output sum $s \in \{0,1\}^4$.

3. Count the appearance of 48-bit ciphertext subsequences, $X^{22}_{[0,2,4,5,6,7,8,9,10,11,14,15]}$, and list those have odd counts to form the list $\mathcal{L}_2$. We then guess

$$\mathcal{K}_2 = (\mathrm{RK}^{22}_{[2,3,4,5,7]}, \mathrm{RK}^{21}_{[0,5,6]}, \mathrm{RK}^{20}_7, \mathrm{RK}^{18}_2)$$

and compute the sum of X_0^{16} using all entries in $\mathcal{L}_2$ with each guess for $\mathcal{K}_2$. The key guesses are grouped according to the sum of X_0^{16}. The key group with X_0^{16} sum being $s' \in \{0,1\}^4$ is denoted by $\mathcal{G}_2(s')$.

4. Extract the all "consistent" combinations from $\mathcal{G}_1(s)$ and $\mathcal{G}_2(s)$ for all $s \in \{0,1\}^4$, and output them as the set of valid key candidates for $\mathcal{K}_{\text{target}}$. This can be done as follows. Let $v \in \mathcal{G}_1(0000)$ and $w \in \mathcal{G}_2(0000)$. We denote the guess for RK^i_j in v and w by $\mathrm{RK}^i_j(v)$ and $\mathrm{RK}^i_j(w)$. Both v and w contain guesses of $\mathrm{RK}^{22}_{[2,3,4,5]}$ and RK^{21}_5. If the following four equations,

$$\mathrm{RK}^{22}_{[2,3,4,5]}(v) = \mathrm{RK}^{22}_{[2,3,4,5]}(w), \quad \mathrm{RK}^{21}_5(v) = \mathrm{RK}^{21}_5(w), \tag{8}$$

$$\mathrm{RK}^{21}_6(w) = S(\mathrm{RK}^{21}_2(v)) \oplus \mathrm{RK}^{18}_2(w), \tag{9}$$

$$\mathrm{RK}^{22}_7(w) = S(S(\mathrm{RK}^{20}_7(w)) \oplus S^{-1}(\mathrm{RK}^{20}_6(v) \oplus \mathrm{RK}^{17}_2(v))) \oplus \mathrm{RK}^{21}_0(w) \tag{10}$$

hold true, a valid key candidate for $\mathcal{K}_{\text{target}}$ is obtained by combining v and w. The check is done for all pairs from $\mathcal{G}_1(s) \times \mathcal{G}_2(s)$, and for all $s \in \{0,1\}^4$.

Taking Step 2 for example, we explain the detailed procedure. We first guess RK^{22}_0 and compute $X^{21}_1 (= F^{22}_0(X^{22}_0) = S(\mathrm{RK}^{22}_0 \oplus X^{22}_0))$ using $\mathcal{L}_1$ with 2^{64} F evaluations. Then we substitute $X^{22}_{[0,1]}$ (8 bits) written in $\mathcal{L}_1$ with X^{21}_1 (4 bits) and obtain a list of 56-bit sequences, and collect the values with odd appearance to form a new list, called $\mathcal{L}_{1,1}$. Next, we guess RK^{22}_2 and compute X^{21}_5 based on the guess with 2^{64} F evaluations. We then substitute $X^{22}_{[4,5]}$ with X^{21}_5 in $\mathcal{L}_{1,1}$ and obtain the list of 52-bit sequences and collect the values with odd appearance to form a new list, called List $\mathcal{L}_{1,2}$. The above procedure is repeated to gradually reduce the list size. Eventually the computation of the sum of F_0^{16} outputs requires $2^{73.80}$ F evaluations (equivalently $2^{66.34}$ encryptions of 22-round TWINE). A similar complexity reduction can also be applied to Step 3, however, the computation of Step 3 is much smaller than that of Step 2 (due to the small space for the key guess) in any case.

As checks are done w.r.t. 4-bit internal values, the above procedure with one S-structure rejects a wrong key candidate for $\mathcal{K}_{\text{target}}$ with probability $1 - 2^{-4}$ (i.e. the size of candidates is reduced to 1/16), hence we basically need at least

[3] More precisely, $\mathrm{RK}^{20}_{[0,5]}$,$\mathrm{RK}^{19}_{[1,7]}$,RK^{18}_3 are required to compute F_0^{16} outputs. Also RK^{20}_1 and RK^{19}_3 are required to compute X_0^{16} in Step 3. These RKs can be computed from $\mathcal{K}_1$ or $\mathcal{K}_2$. See Appendix E.

18 S-structures to identify the right key. However, this is impossible as each sub-block is 4-bit. To elude the problem, we exploit the key schedule; the structure of the key schedule allows us to derive the 80-bit key with 2^{68} candidates for 72-bit subkey, and an exhaustive search for the remaining 8-bit subkey, and the final key check, which is trivial.

Summarizing, the attack with an S-structure requires 2^{60} plaintexts to be encrypted, and 2^{77} (which follows from $2^{66.34} + 2^{76} + \rho$, where ρ denotes the computation of Step 3, which is negligible) encryptions. The memory complexity is 2^{67} (64-bit) blocks. If we want to further reduce the complexity, using multiple S-structures (using distinct constants with the same SC) can help. The result is shown by Table 4. According to our investigation, the attack with 5 or more structures has higher time complexity than that with 4 structures, hence the best one is with 4 structures.

Table 4. Complexity of Saturation Attack.

# of Struct.s	Data	Time (Enc)	Memory (Block)
1	2^{60}	$2^{77}(\approx 2^{60} + 2^{66.34} + \rho + 2^{76} + 2^{12})$	2^{67}
2	2^{61}	$2^{73}(\approx 2^{61} + (2^{66.34} + \rho) \cdot 2 + 2^{72} + 2^{8})$	2^{67}
3	$2^{61.59}$	$2^{68.97}(\approx 2^{61.59} + (2^{66.34} + \rho) \cdot 3 + 2^{68} + 2^{4})$	2^{67}
4	2^{62}	$2^{68.43}(\approx 2^{62} + (2^{66.34} + \rho) \cdot 4 + 2^{64})$	2^{67}

4.4 Differential / Linear Cryptanalysis

To evaluate the resistance against differential cryptanalysis (DC) [5] and linear cryptanalysis (LC) [29], we need to know the number of differentially and linearly active S-boxes, denoted by AS_D and AS_L, respectively. We performed a computer-based search for differential and linear paths, and evaluated AS_D and AS_L for each round. As a result, the numbers of AS_D and AS_L are the same, as shown by Table 5. Since our S-box has 2^{-2} maximum differential and linear probabilities, the maximum differential and linear characteristic probabilities are both 2^{-64} for 15 rounds. Examples of 14-round differential (Δ) and linear (Γ) characteristics having the minimum I/O weights are as follows. They involve 30 active S-boxes, and thus the characteristic probability is 2^{-60}.

$$\begin{aligned} \Delta &= (0^9, 1, 0, 1, 0, 1, 0, 0) \xrightarrow{14r} (0^3, 1, 0^4, 1, 0, 0, 1, 0, 0, 1, 1), \\ \Gamma &= (0^6, 1, 1, 0^3, 1, 0, 0, 1, 1) \xrightarrow{14r} (0^9, 1, 0^3, 1, 0, 1). \end{aligned} \tag{11}$$

Here, 1 denotes an arbitrary non-zero difference (mask) and 0 denotes the zero difference (mask) for Δ (Γ). Compared to the impossible differential attack, we naturally expect the key recovery attacks exploiting the key schedule with these

differential/linear characteristics are less powerful, since 14-round impossible differential characteristic has much fewer (only 2) weights, and fewer weights imply the more attackable rounds in the key guessing. We remark that a computer-based search for the maximum differential probability (rather than the characteristic probability) of GFS was proposed by [30]. However, applying their algorithm to our 16-block case seems infeasible due to the computational complexity.

Table 5. Number of Differentially and Linearly Active S-boxes.

Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
AS_D, AS_L	0	1	2	3	4	6	8	11	14	18	22	24	27	30	32

4.5 Key Schedule-based Attacks

Related-Key Differential Attacks. The related-key attack, proposed by Biham [4], is an attack applicable to the environment where the adversary can somehow modify the key input. We focus on the typical setting, that is, the adversary is allowed to insert a key differential. In order to evaluate the resistance of TWINE against the related-key attack, we implemented the search method proposed by Biryukov et al. [8], which counts the number of active S-boxes for combined data processing and key schedule parts. See [8] for the detail of the algorithm. We (naturally) searched 4-bit truncated differential paths. As S-box has maximum differential probability being 2^{-2}, we needed 40 (64) active S-boxes for TWINE-80 (TWINE-128).

Due to the computational constraint the full-search is only feasible for TWINE-80. As a result, we confirmed that the number of active S-boxes reaches 40 for the 22-round. Appendix F provides the number of active S-boxes and the corresponding truncated differential paths.

Other Attacks. For the slide attack [7], the key schedule of TWINE inserts distinct constants for each round. This is a typical way to thwart slide attacks and hence we consider TWINE is immune to the slide attack.

For Meet-In-The-Middle (MITM) attack, we confirmed that the round keys for the first 3 (5) rounds contain all key bits for the 80-bit (128-bit) key case. Thus, we consider it is difficult to mount the basic MITM attack against the full-round TWINE. Note that the recently-proposed MITM variant, called biclique attack [11], may work even when all key bits are used in the relatively small number of rounds. The evaluation of such attack against TWINE is a future topic.

The result of our security evaluation for TWINE is summarized at Table 6.

Table 6. Summary of Attacks on TWINE.

Key (bits)	Attack	Rounds	Data (blocks)	Time (encryption)	Memory (blocks)
80	Impossible Diff.	23	$2^{61.39}$	$2^{76.88}$	2^{74}
	Saturation	22	2^{62}	$2^{68.43}$	2^{67}
128	Impossible Diff.	24	$2^{52.21}$	$2^{115.10}$	2^{118}
	Saturation	23	$2^{62.81}$	$2^{106.14}$	2^{103}

5 Implementation

5.1 Hardware

We implemented TWINE on ASIC using a 90*nm* standard cell library with logic synthesis done by *Synopsys Design Compiler Version D-2010.03-SP1-1*. Following the recent trend in the lightweight implementations [9,13], the figures are shown for the case when Scan Flip-Flops (FFs) are used. In our library, a D-FF and 2-to-1 MUX cost 5.5 GE and 2.25 GE, and a Scan FF costs 6.75 GE: hence this technique saves 1.0 GE per 1-bit storage.

The data path of TWINE-80 encryption circuit is in Fig. 4, and the implementation result is shown by Table 7. We also show the detail of TWINE-80 encryption implementation in Table 8. The figures must be taken with cares, because they are related to the type of memory unit (FF), technology, library, etc, as pointed out by [13]. Even single XOR gate has several grades, from fast-but-large and slow-but-small. As suggested by [13], we list Gates/Memory Bit in the table, which denotes the size (in Gate Equivalent (GE)) of 1-bit memory device used for the key and states.

We did not perform a thorough logic minimization of the S-box circuit, which currently costs 30 GEs. The S-box logic minimization can further reduce the size.

We also tried a serialized implementation. Though it is not yet finished, the preliminary result indicates that encryption-only TWINE-80 can be implemented 1, 116 GEs. The details of our serialized implementation will be given in a forthcoming paper.

5.2 Software

To evaluate the performance on embedded software, we implement TWINE on Atmel AVR 8-bit Micro-controller. The target device is ATmega163, which has 16K bytes Flash, 512 bytes EEPROM and 1024 bytes SRAM. We implemented the four versions: speed-first, ROM-first (minimizing the consumption), and RAM-first, and the double-block implementation, where two message blocks are processed in parallel. Such an implementation works for parellelizable modes of operations, e.g., the counter mode and PMAC. All implementations include the precomputation of round keys, i.e. they do not use an on-the-fly key schedule.

Table 7. ASIC Implementation Results. For some implementations, the figures of Throughput and Cycles/Block is an estimated value.

Algorithm	Function	Block size (bit)	Key size (bit)	Cycles/ block	Throughput (Kbps@100KHz)	Area (GE†)	Gates / Memory bit	Type
TWINE	Enc	64	80	36	178	1503	6.75	round
TWINE	Enc+Dec	64	80	36	178	1799	6.75	round
TWINE	Enc	64	128	36	178	1866	6.75	round
TWINE	Enc+Dec	64	128	36	178	2285	6.75	round
TWINE	Enc	64	80	540	11.8	1116	6.75	serial
PRESENT [36]	Enc	64	80	547	11.4	1000	n/a	serial
PRESENT [9]	Enc	64	80	32	200	1570	6	round
AES [31]	Enc	128	128	226	57	2400	6	serial
mCRYPTON [27]	Enc	64	64	13	492.3	2420	5	round
SEA [28]	Enc+Dec	96	96	93	103	3758	n/a	round
HIGHT [21]	Enc+Dec	64	128	34	188.25	3048	n/a	round
KLEIN [19]	Enc	64	80	17	376.4	2629	n/a	round
KLEIN [19]	Enc	64	80	271	23.6	1478	n/a	serial
DES [26]	Enc	64	56	144	44.4	2309	12.19	serial
DESL [26]	Enc	64	56	144	44.4	1848	12.19	serial
KATAN [13]	Enc	64	80	254	25.1	1054	6.25	serial
Piccolo [40]	Enc	64	80	27	237	1496¶	6.25	round
Piccolo [40]	Enc+Dec	64	80	27	237	1634¶	6.25	round
Piccolo [40]	Enc	64	80	432	14.8	1043¶	6.25	serial
Piccolo [40]	Enc+Dec	64	80	432	14.8	1103¶	6.25	serial
LED [18]	Enc	64	80	1872	3.4	1040	6/4.67◇	serial
PRINTcipher [25]	Enc	48	80	48	12.5	503⋆	n/a	round

† Gate Equivalent : cell area/2-input NAND gate size (2.82).
¶ Includes a key register that costs 360 GEs; Piccolo can be implemented without a key register if key signal holds while encryption.
◇ Mixed usage of two memory units.
⋆ Hardwired key.

Table 8. TWINE-80 Encryption Hardware Implementation.

Data Processing Part (GE)		Key Scheduling Part (GE)	
Data Register	432	Key Register	540
S-box	240	round const comp.	2
Round Key XOR	64	round const XOR	12
S-box out XOR	64	S-box	60
		S-box out XOR	16
		RC register	33
		State register	6
		Others/Control	34
		Total	1503

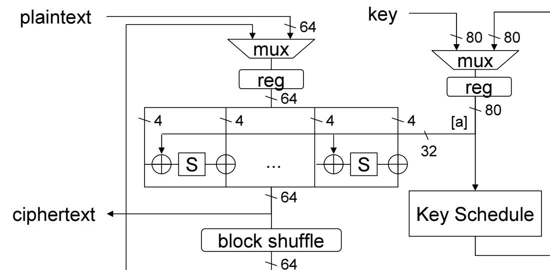


Fig. 4. Data path of TWINE-80 encryption, when Scan FF is not used (i.e. the case with MUX and D-FF). The bit boundary $[a]$ indicates certain 32 bits of 80-bit key state specified from the key schedule.

In the speed-first implementation, two rounds are processed in one loop. This removes the block shuffle between the first and second rounds. A further speeding up is possible if more rounds are contained in one loop at the cost of increased memory. Our program keeps all 4-bit blocks in the distinct registers. RAM load instruction (LD) is faster than ROM load instruction (LPM), hence the S-box and the constants are stored at RAM. The data arrangement is carefully considered to avoid carry in the address computation. The double-block implementation stores the S-boxes in ROM.

Our result is in Table. 9, and a comparison is in Table 10. In Table 10 we list the (scaled) throughput/code ratio for a performance measure (See Table 10 for the formula), following [35]. AES's performance is still quite impressive, however, one can also observe a good performance of TWINE.

One might be interested in the performance of TWINE under 32/64-bit CPU. We are currently working on this, in particular using the vector permutation instructions, which was shown to be very powerful for AES [20].

Table 9. Software Implementation of TWINE on ATmega163.

Target	Key schedule (cycles)	Encryption (cycles/block)	Decryption (cycles/block)	ROM (bytes)	RAM (bytes)
Speed-first	2,170	2,165	2,166	1,304	414
ROM-first	12,022	18,794	18,689	728	335
RAM-first	12,058	18,794	18,688	792	191
Double-block	1,887	1,301	1,302	2,294	386

Table 10. Comparison of Software Implementation on AVR.

Algorithm	Key (bits)	Block (bits)	Language	ROM (bytes)	RAM (bytes)	Enc (cyc/byte)	Dec (cyc/byte)	ETput /Code†	DTput /Code‡
TWINE	80	64	asm	1,304	414	271	271	2.14	2.14
PRESENT [33]	80	64	asm	2,398	528	1,199	1,228	0.28	0.28
PRESENT [17]	80	64	N/A	936	0	1,340	1,405	0.80	0.76
DES [17]	56	64	N/A	4,314	0	1,079	1,019	0.21	0.22
DESXL [17]	184	64	N/A	3,192	0	1,066	995	0.29	0.31
HIGHT [17]	128	64	N/A	5,672	0	371	371	0.48	0.48
IDEA [17]	128	64	N/A	596	0	338	1,924	4.97	0.87
TEA [17]	128	64	N/A	1,140	0	784	784	1.11	1.11
SEA [17]	96	96	N/A	2,132	0	805	805	0.58	0.58
AES [12]	128	128	asm	1,912	432	125	181	3.42	2.35

† Encryption Throughput per Code: (1/Enc)/(ROM + RAM) (scaled by 10^6)
‡ Decryption Throughput per Code: (1/Dec)/(ROM + RAM) (scaled by 10^6)

6 Conclusions

We have presented a lightweight block cipher TWINE, which has 64-bit block and 80 or 128-bit key. It is primary designed to fit extremely-small hardware, yet provides a notable performance under embedded software. This characteristic mainly originates from the Type-2 generalized Feistel with a highly-diffusive block shuffle. We performed a thorough security analysis, in particular for the impossible differential and saturation attacks. Although the result implies the sufficient security of full-round TWINE, its security naturally needs to be studied further.

Acknowledgments. The authors would like to thank the anonymous reviewers for many useful comments. We thank Maki Shigeri, Etsuko Tsujihara, and Teruo Saito for discussions, and Daisuke Ikemura for investigation on hardware-related issues.

References

1. Final report of European project IST-1999-12324, New European Schemes for Signatures, Integrity, and Encryption. 2004.
2. `http://www.lightweightcrypto.org/implementations.php`
3. `http://cis.sjtu.edu.cn/index.php /Software_Implementation_of_Block_Cipher_PRESENT_for_8-Bit_Platforms`
4. E. Biham. "New Types of Cryptanalytic Attacks Using Related Keys." Journal of Cryptology, vol. 7, no. 4, pp. 229-246, 1994.
5. E. Biham and A. Shamir. "Differential Cryptanalysis of the Data Encryption Standard." Springer-Verlag, 1993.

6. E. Biham, A. Biryukov and A. Shamir, "Cryptanalysis of skipjack reduced to 31 rounds using impossbile differentials." *Advances in Cryptology* - EUROCRYPT '99, LNCS 1592, pp.12-23, 1999.
7. A. Biryukov and D. Wagner. "Slide Attacks." FSE'99, LNCS 1636, pp.245-259.
8. A. Biryukov and I. Nikoliċ. "Automatic Search for Related-Key Differential Characteristics in Byte-Oriented Block Ciphers: Application to AES, Camellia, Khazad, and Others." EUROCRYPT 2010.
9. A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M.J.B. Robshaw, Y. Seurin and C. Vikkelsoe, "PRESENT: An Ultra-Lightweight Block Cipher", CHES 2007, LNCS 4727, pp. 450-466, 2007.
10. A. Bogdanov, L.R. Knudsen, G. Leander, C. Paar, A. Poschmann, M.J.B. Robshaw, Y. Seurin, and C. Vikkelsoe. "Small-Footprint Block Cipher Design -How far can you go?" 3rd Conference on RFID Security, 2007.
11. A. Bogdanov, D. Khovratovich and C. Rechberger. "Biclique Cryptanalysis of the Full AES." cryptology eprint archive 2011/449.
12. J. W. Bos, D. A. Osvik, D. Stefan. "Fast Implementations of AES on Various Platforms." SPEED-CC – Software Performance Enhancement for Encryption and Decryption and Cryptographic Compilers, 2009.
13. C. D. Canniere, O. Dunkelman and M. Knezevic. "KATAN and KTANTAN - A Family of Small and Efficient Hardware-Oriented Block Ciphers." CHES '09, pp. 272-288, 2009.
14. D. Canright. "A Very Compact S-Box for AES." *Cryptographic Hardware and Embedded Systems*- CHES'05, LNCS 3659, pp. 441-455, 2005.
15. J. Chen, K. Jia, H. Yu, and X. Wang. "New Impossible Differential Attacks of Reduced-Round Camellia-192 and Camellia-256." ACISP 2011, LNCS 6812, pp.16-33, 2011
16. J. Daemen, L. R. Knudsen, and V. Rijmen, "The Block Cipher SQUARE." *Fast Software Encryption*-FSE'97, LNCS 1267, pp.149-165, 1997.
17. T. Eisenbarth, S. Kumar, C. Paar, A. Poschmann, and L. Uhsadel. "A Survey of Lightweight Cryptography Implementations." IEEE Design & Test of Computers – Special Issue on Secure ICs for Secure Embedded Computing, 24(6):522-533, November/December 2007.
18. J. Guo, T. Peyrin, A. Poschmann, M. J. B. Robshaw. "The LED Block Cipher.", CHES'10, LNCS 6225, pp. 326-341.
19. Z. Gong, S. Nikova and Y.-W. Law. "KLEIN: A New Family of Lightweight Block Ciphers." RFIDsec 2011.
20. M. Hamburg. "Accelerating AES with Vector Permute Instructions." CHES 2009, LNCS 5747, pp. 18-32.
21. D. Hong, J. Sung, S. Hong, J. Lim, S. Lee, B. Koo, C. Lee, D. Chang, J. Lee, K. Jeong, H. Kim, J. Kim and S. Chee, "HIGHT: A New Block Cipher Suitable for Low-Resource Device." CHES 2006, LNCS 4249, pp. 46-59, 2006.
22. M. Izadi, B. Sadeghiyan, S. S. Sadeghian, and H. A. Khanooki. "MIBS: A New Lightweight Block Cipher." CANS'09, LNCS 5888, pp. 334-348, 2009.
23. A. Juels and S. A. Weis. "Authenticating Pervasive Devices with Human Protocols." CRYPTO'05, LNCS 3126, pp. 293-198, 2005.
24. J. Kim, S. Hong, J. Sung, C. Lee, S. Lee, "Impossible Differential Cryptanalysis for Block Cipher Structures." INDOCRYPT 2003, LNCS 2904, pp.82-96, 2003.
25. L. R. Knudsen, G. Leander, A. Poschmann and M. J. B. Robshaw. "PRINTcipher: A Block Cipher for IC-Printing." CHES'10, LNCS 6225, pp. 16-32.
26. G. Leander, C. Paar, A. Poschmann, and K. Schramm. "New Lightweight DES Variants." *Fast Software Encryption*-FSE'07, LNCS 4593, pp. 196-210, 2007.

27. C. H. Lim and T. Korkishko. "mCrypton - A Lightweight Block Cipher for Security of Low-Cost RFID Tags and Sensors". *Information Security Applications*-WISA'05, LNCS 3786, pp. 243-258, 2005.
28. F. Mace, F.-X. Standaert, and J.-J. Quisquater. "Implementations of the Block Cipher SEA for Constrained Applications." Proceedings of the Third International Conference on RFID Security, RFIDSec 2007, pp.103-114.
29. M. Matsui, "Linear cryptanalysis of the data encryption standard." EUROCRYPT'93, LNCS 765, pp.386-397, Springer-Verlag, 1994.
30. K. Minematsu, T. Suzaki, and M. Shigeri. "On Maximum Differential Probability of Generalized Feistel." ACISP 2011, LNCS 6812, pp. 89-105, 2011.
31. A. Moradi, A. Poschmann, S. Ling, C. Paar, and H. Wang. "Pushing the Limits: A Very Compact and a Threshold Implementation of AES." Eurocrypt 2011, LNCS 6632, pp. 69-88.
32. O. Özen, K. Varici, C. Tezcan, Ç. Kocair. "Lightweight Block Ciphers Revisited: Cryptanalysis of Reduced Round PRESENT and HIGHT." ACISP 2009, LNCS 5594, pp. 90-107, 2009.
33. A. Poschmann. "Lightweight Cryptography - Cryptographic Engineering for a Pervasive World." Cryptology ePrint Archive, Report 2009/516, 2009.
34. A. Poschmann, S. Ling, and H. Wang. "256 Bit Standardized Crypto for 650 GE - GOST Revisited." CHES'10, LNCS 6225, pp. 219-233.
35. S. Rinne, T. Eisenbarth, and C. Paar. "Performance Analysis of Contemporary Lightweight Block Ciphers on 8-bit Microcontrollers." SPEED – Software Performance Enhancement for Encryption and Decryption, 2007.
36. C. Rolfes, A. Poschmann, G. Leander, and C. Paar. "Ultra-Lightweight Implementations for Smart Devices - Security for 1000 Gate Equivalents." Smart Card Research and Advanced Application Conference (CARDIS 2008), LNCS 5189, pp. 89-103, 2008.
37. Akashi Satoh, Sumio Morioka, Kohji Takano and Seiji Munetoh, "A Compact Rijndael Hardware Architecture with S-Box Optimization." ASIACRYPT2001, LNCS Vol.2248, pp.239-254, Dec 2001.
38. T. Shirai, K. Shibutani, T. Akishita, S. Moriai and T. Iwata, "The 128-bit Blockcipher CLEFIA." *Fast Software Encryption*-FSE'07, LNCS 4593, pp. 181-195, 2007.
39. K. Shibutani. "On the Diffusion of Generalized Feistel Structures Regarding Differential and Linear Cryptanalysis." SAC'10, LNCS 6544, pp. 211-228, 2011.
40. K. Shibutani, T. Isobe, H. Hiwatari, A. Mitsuda, T. Akishita, and T. Shirai. "Piccolo: An Ultra-Lightweight Blockcipher." CHES 2011, LNCS 6917, pp. 342-357, 2011.
41. T. Suzaki and K. Minematsu. "Improving the Generalized Feistel." *Fast Software Encryption*-FSE'10, LNCS 6147, pp.19-39, 2010.
42. Y. Tsunoo, E. Tsujihara, M. Shigeri, T. Saito, T. Suzaki, and H. Kubo. "Impossible Differential Cryptanalysis of CLEFIA." FSE 2008. LNCS 5086, pp.398-411, 2008.
43. W. Wu and L. Zhang. "LBlock: A Lightweight Block Cipher. " 9th International Conference on Applied Cryptography and Network Security, ACNS '11, LNCS 6715, pp. 327-344.
44. Y. Zheng, T. Matsumoto, and H. Imai. "On the Construction of Block Ciphers Provably Secure and Not Relying on Any Unproved Hypotheses." *Advances in Cryptology* - CRYPTO '89, LNCS 435, pp. 461-480, 1989.

A Key Schedule for 128-bit Key

Algorithm A.1: TWINE.KeySchedule-128($K_{(128)}, RK_{(32\times 36)}$)

$\mathrm{WK}_{(128)} \leftarrow K$
$\mathrm{WK}_{0(16)} \| \mathrm{WK}_{1(16)} \| \dots \| \mathrm{WK}_{6(16)} \| \mathrm{WK}_{7(16)} \leftarrow WK$
$\mathrm{RK}^1_{0(4)} \leftarrow \mathrm{WK}_2,\ \ \mathrm{RK}^1_{1(4)} \leftarrow \mathrm{WK}_3,\ \ \mathrm{RK}^1_{2(4)} \leftarrow \mathrm{WK}_{12},\ \ \mathrm{RK}^1_{3(4)} \leftarrow \mathrm{WK}_{15}$
$\mathrm{RK}^1_{4(4)} \leftarrow \mathrm{WK}_{17},\ \mathrm{RK}^1_{5(4)} \leftarrow \mathrm{WK}_{18},\ \mathrm{RK}^1_{6(4)} \leftarrow \mathrm{WK}_{28},\ \mathrm{RK}^1_{7(4)} \leftarrow \mathrm{WK}_{31}$
$\mathrm{RK}^1_{(32)} \leftarrow \mathrm{RK}^1_0 \| \mathrm{RK}^1_1 \| \dots \| \mathrm{RK}^1_6 \| \mathrm{RK}^1_7$
for $i \leftarrow 2$ **to** 36

$$
\textbf{do}
\begin{cases}
\mathrm{WK}_1 \leftarrow \mathrm{WK}_1 \oplus S(\mathrm{WK}_0) \\
\mathrm{WK}_4 \leftarrow \mathrm{WK}_4 \oplus S(\mathrm{WK}_{16}) \\
\mathrm{WK}_{23} \leftarrow \mathrm{WK}_{23} \oplus S(\mathrm{WK}_{30}) \\
\mathrm{WK}_7 \leftarrow \mathrm{WK}_7 \oplus 0 \| CON^{i-1}_H \\
\mathrm{WK}_{19} \leftarrow \mathrm{WK}_{19} \oplus 0 \| CON^{i-1}_L \\
\mathrm{tmp}_0 \leftarrow \mathrm{WK}_0,\ \ \mathrm{tmp}_1 \leftarrow \mathrm{WK}_1,\ \ \mathrm{tmp}_2 \leftarrow \mathrm{WK}_2,\ \ \mathrm{tmp}_3 \leftarrow \mathrm{WK}_3 \\
\textbf{for } j \leftarrow 0 \textbf{ to } 6 \\
\quad \textbf{do} \begin{cases} \mathrm{WK}_{j*4} \leftarrow \mathrm{WK}_{j*4+4}, & \mathrm{WK}_{j*4+1} \leftarrow \mathrm{WK}_{j*4+5} \\ \mathrm{WK}_{j*4+2} \leftarrow \mathrm{WK}_{j*4+6}, & \mathrm{WK}_{j*4+3} \leftarrow \mathrm{WK}_{j*4+7} \end{cases} \\
\mathrm{WK}_{28} \leftarrow \mathrm{tmp}_1, \mathrm{WK}_{29} \leftarrow \mathrm{tmp}_2, \mathrm{WK}_{30} \leftarrow \mathrm{tmp}_3, \mathrm{WK}_{31} \leftarrow \mathrm{tmp}_0 \\
\mathrm{RK}^i_0 \leftarrow \mathrm{WK}_2,\ \ \mathrm{RK}^i_1 \leftarrow \mathrm{WK}_3,\ \ \mathrm{RK}^i_2 \leftarrow \mathrm{WK}_{12},\ \ \mathrm{RK}^i_3 \leftarrow \mathrm{WK}_{15} \\
\mathrm{RK}^i_4 \leftarrow \mathrm{WK}_{17},\ \mathrm{RK}^i_5 \leftarrow \mathrm{WK}_{18},\ \mathrm{RK}^i_6 \leftarrow \mathrm{WK}_{28},\ \mathrm{RK}^i_7 \leftarrow \mathrm{WK}_{31} \\
\mathrm{RK}^i_{(32)} \leftarrow \mathrm{RK}^i_{0(4)} \| \mathrm{RK}^i_{1(4)} \| \dots \| \mathrm{RK}^i_{6(4)} \| \mathrm{RK}^i_{7(4)}
\end{cases}
$$

$RK_{(32\times 36)} \leftarrow \mathrm{RK}^1_{(32)} \| \mathrm{RK}^2_{(32)} \| \dots \| \mathrm{RK}^{35}_{(32)} \| \mathrm{RK}^{36}_{(32)}$

B Test Vectors

Table 11. Test Vectors in the Hexadecimal Notation.

key length	80-bit	128-bit
key	00112233 44556677 8899	00112233 44556677 8899AABB CCDDEEFF
plaintext	01234567 89ABCDEF	01234567 89ABCDEF
ciphertext	7C1F0F80 B1DF9C28	979FF9B3 79B5A9B8

C 80-bit Key Schedule

Figure 5 depicts the key schedule for TWINE-80.

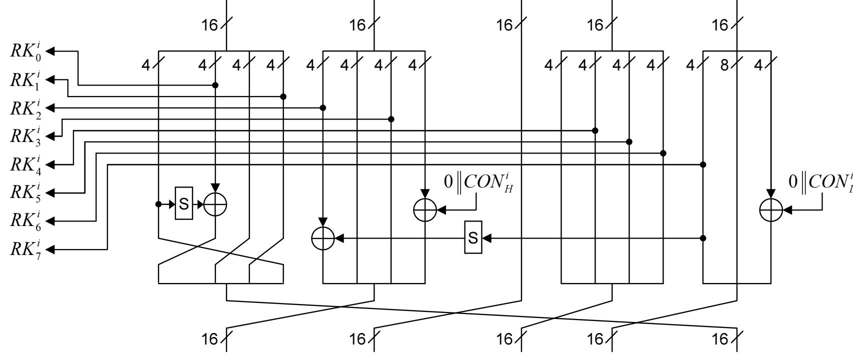


Fig. 5. 80-bit Key Schedule.

D Supplementary Information for Impossible Differential Cryptanalysis

The following is the subkey relationships used for the impossible differential cryptanalysis.

$$\mathrm{RK}_3^3 = \mathrm{RK}_5^1$$
$$\mathrm{RK}_5^3 = \mathrm{RK}_1^1$$
$$\mathrm{RK}_1^4 = \mathrm{RK}_6^1$$
$$\mathrm{RK}_0^{21} = S[S[S[\mathrm{RK}_4^{19}] \oplus \mathrm{RK}_3^1] \oplus \mathrm{RK}_4^{21}] \oplus \mathrm{RK}_7^2$$
$$\mathrm{RK}_5^{21} = S^{-1}[S^{-1}[\mathrm{RK}_1^{21} \oplus S[\mathrm{RK}_3^{20}] \oplus \mathrm{RK}_1^1] \oplus \mathrm{RK}_6^1]$$
$$\mathrm{RK}_0^{22} = \mathrm{RK}_4^{19}$$
$$\mathrm{RK}_2^{22} = S[\mathrm{RK}_6^{20}] \oplus S[\mathrm{RK}_7^2] \oplus \mathrm{RK}_2^2$$
$$\mathrm{RK}_4^{22} = S[S[S[\mathrm{RK}_3^{20}] \oplus \mathrm{RK}_1^1] \oplus \mathrm{RK}_3^{22}] \oplus S[S[S[S[\mathrm{RK}_6^{20}] \oplus S[\mathrm{RK}_7^2] \oplus \mathrm{RK}_2^2] \oplus S^{-1}[\mathrm{RK}_1^{22}$$
$$\oplus S^{-1}[S^{-1}[S^{-1}[\mathrm{RK}_1^{21} \oplus S[\mathrm{RK}_3^{20}] \oplus \mathrm{RK}_1^1] \oplus \mathrm{RK}_6^1] \oplus S[\mathrm{RK}_4^{21}] \oplus \mathrm{RK}_5^1]]] \oplus \mathrm{RK}_7^{21}] \oplus \mathrm{RK}_4^2$$
$$\mathrm{RK}_5^{22} = S^{-1}[S^{-1}[\mathrm{RK}_1^{22} \oplus S^{-1}[S^{-1}[S^{-1}[\mathrm{RK}_1^{21} \oplus S[\mathrm{RK}_3^{20}] \oplus \mathrm{RK}_1^1] \oplus \mathrm{RK}_6^1] \oplus S[\mathrm{RK}_4^{21}] \oplus \mathrm{RK}_5^1]]$$
$$\oplus \mathrm{RK}_6^2]$$
$$\mathrm{RK}_6^{22} = S[S[\mathrm{RK}_6^{20}] \oplus S[\mathrm{RK}_7^2] \oplus \mathrm{RK}_2^2] \oplus S^{-1}[\mathrm{RK}_1^{22} \oplus S^{-1}[S^{-1}[S^{-1}[\mathrm{RK}_1^{21} \oplus S[\mathrm{RK}_3^{20}] \oplus \mathrm{RK}_1^1]$$
$$\oplus \mathrm{RK}_6^1] \oplus S[\mathrm{RK}_4^{21}] \oplus \mathrm{RK}_5^1]]$$
$$\mathrm{RK}_0^{23} = S[S[S[\mathrm{RK}_4^{21}] \oplus \mathrm{RK}_5^1] \oplus \mathrm{RK}_3^{20}] \oplus S^{-1}[S[S[S[\mathrm{RK}_6^{20}] \oplus S[\mathrm{RK}_7^2] \oplus \mathrm{RK}_2^2] \oplus S^{-1}[\mathrm{RK}_1^{22}$$
$$\oplus S^{-1}[S^{-1}[S^{-1}[\mathrm{RK}_1^{21} \oplus S[\mathrm{RK}_3^{20}] \oplus \mathrm{RK}_1^1] \oplus \mathrm{RK}_6^1] \oplus S[\mathrm{RK}_4^{21}] \oplus \mathrm{RK}_5^1]]] \oplus \mathrm{RK}_7^{21} \oplus \mathrm{RK}_7^1]$$
$$\mathrm{RK}_1^{23} = \mathrm{RK}_6^{20}$$
$$\mathrm{RK}_3^{23} = S^{-1}[S^{-1}[\mathrm{RK}_1^{21} \oplus S[\mathrm{RK}_3^{20}] \oplus \mathrm{RK}_1^1] \oplus \mathrm{RK}_6^1]$$
$$\mathrm{RK}_4^{23} = \mathrm{RK}_3^{20}$$
$$\mathrm{RK}_5^{23} = \mathrm{RK}_1^{21}$$
$$\mathrm{RK}_6^{23} = S[\mathrm{RK}_2^{23}] \oplus S[\mathrm{RK}_1^{21}] \oplus S^{-1}[\mathrm{RK}_7^2 \oplus \mathrm{RK}_0^1]$$
$$\mathrm{RK}_7^{23} = S[S[\mathrm{RK}_7^{21}] \oplus S[\mathrm{RK}_1^{22}] \oplus S[\mathrm{RK}_7^1] \oplus \mathrm{RK}_2^1] \oplus \mathrm{RK}_4^{19}$$

Table 12. Procedure of Round Key Determination.

Step	target key	method	
1	$\mathrm{RK}_0^1, \mathrm{RK}_5^1, \mathrm{RK}_6^1$	diff-table	
2	$\mathrm{RK}_4^3, \mathrm{RK}_1^4$	key rel.	Input of F_3^3 and output of F_4^2 are decided from diff-table. Input of F_1^4 is decided from diff-table.
3	RK_3^1	guess	
4	RK_4^2	IO value	
5	RK_7^1	guess	Input of F_7^2 is decided.
6	RK_7^2	diff-table	
7	RK_1^1	guess	Input of F_2^2 is decided.
8	RK_5^3	key rel.	Output of F_6^2 is decided.
9	RK_2^2	diff-table	
10	RK_2^1	guess	
11	RK_6^2	IO value	
12	$\mathrm{RK}_2^{23}, \mathrm{RK}_4^{23}, \mathrm{RK}_5^{23}$	diff-table	IO value of F_0^{22} is decided.
13	$\mathrm{RK}_3^{20}, \mathrm{RK}_1^{21}$	key rel.	Input of F_3^{20} is decided from diff-table.
14	$\mathrm{RK}_5^{21}, \mathrm{RK}_3^{23}, \mathrm{RK}_6^{23}$	key rel.	Input of F_2^{22} is decided. IO value of F_0^{22} is decided. Input of F_5^{21} is decided, then output of F_4^{22} is decided.
15	RK_2^{22}	diff-table	
16	RK_6^{20}	key rel.	
17	RK_1^{23}	key rel.	IO value of F_3^{22} is decided.
18	RK_3^{22}	diff-table	
19	RK_0^{22}	IO value	
20	RK_4^{19}	key rel.	
21	RK_4^{22}	IO value	
22	RK_0^{23}	guess	IO value of F_1^{22} is decided.
23	RK_1^{22}	diff-table	
24	$\mathrm{RK}_4^{21}, \mathrm{RK}_7^{21}$	key rel.	
25	$\mathrm{RK}_5^{22}, \mathrm{RK}_6^{22}, \mathrm{RK}_7^{23}$	key rel.	

E Subkey Relationships used for Saturation Attack

The following is the subkey relationships used for the saturation attack.

$$\begin{aligned}
\mathrm{RK}_4^{21} &= \mathrm{RK}_3^{18} \\
\mathrm{RK}_5^{21} &= \mathrm{RK}_1^{19} \\
\mathrm{RK}_6^{21} &= S[\mathrm{RK}_2^{21}] \oplus \mathrm{RK}_2^{18} \\
\mathrm{RK}_2^{22} &= \mathrm{RK}_7^{19} \\
\mathrm{RK}_3^{22} &= \mathrm{RK}_5^{20} \\
\mathrm{RK}_4^{22} &= \mathrm{RK}_3^{19} \\
\mathrm{RK}_5^{22} &= \mathrm{RK}_1^{20} \\
\mathrm{RK}_6^{22} &= S^{-1}[\mathrm{RK}_7^{21} \oplus \mathrm{RK}_0^{20}] \\
\mathrm{RK}_7^{22} &= S[S[\mathrm{RK}_7^{20}] \oplus S^{-1}[\mathrm{RK}_6^{20} \oplus \mathrm{RK}_2^{17}]] \oplus \mathrm{RK}_0^{21}
\end{aligned}$$

F Related-key Truncated Differential and Its Active S-box Numbers

Table 13 shows the number of active S-boxes using related-key differential, where ΔKS, ΔRK, ΔX and AS denote key state difference, subkey difference, data difference, and the number of active S-boxes.

Table 13. Truncated Differential and Its Active S-box Numbers.

Round	ΔKS	ΔRK	ΔX	AS	Round	ΔKS	ΔRK	ΔX	AS
1	4D010	A2	A255	0	12	60402	80	A0E2	22
2	D8108	E1	6931	6	13	0402C	05	A630	27
3	010C3	08	9896	8	14	C02C0	88	8D39	30
4	10C30	46	4462	9	15	02C01	10	5A2E	33
5	0C302	20	2288	10	16	2C010	22	62C3	35
6	C3020	94	9411	11	17	C0104	80	8191	38
7	30201	40	0968	14	18	01041	08	0824	38
8	02016	12	1306	15	19	10410	42	4202	39
9	20160	0C	4545	19	20	04102	00	0081	39
10	01604	00	108C	20	21	41020	84	8100	41
11	16040	58	D840	21	22	10208	41	4124	41

SPONGENT: The Design Space of Lightweight Cryptographic Hashing

Andrey Bogdanov[1], Miroslav Knežević[1,2], Gregor Leander[3], Deniz Toz[1], Kerem Varıcı[1], and Ingrid Verbauwhede[1]

[1] Katholieke Universiteit Leuven, ESAT/COSIC and IBBT, Belgium
{andrey.bogdanov, deniz.toz, kerem.varici, ingrid.verbauwhede}@esat.kuleuven.be
[2] NXP Semiconductors, Leuven, Belgium
miroslav.knezevic@nxp.com
[3] DTU Mathematics, Technical University of Denmark
g.leander@mat.dtu.dk

Abstract. The design of secure yet efficiently implementable cryptographic algorithms is a fundamental problem of cryptography. Lately, lightweight cryptography – optimizing the algorithms to fit the most constrained environments – has received a great deal of attention, the recent research being mainly focused on building block ciphers. As opposed to that, the design of lightweight hash functions is still far from being well-investigated with only few proposals in the public domain.
In this article, we aim to address this gap by exploring the design space of lightweight hash functions based on the sponge construction instantiated with PRESENT-type permutations. The resulting family of hash functions is called SPONGENT. We propose 13 SPONGENT variants – for different levels of collision and (second) preimage resistance as well as for various implementation constraints. For each of them, on various technologies, we provide several ASIC hardware implementations - ranging from the lowest area to the highest throughput. We also prove essential differential properties of SPONGENT permutations, give a security analysis in terms of collision and preimage resistance, as well as study in detail dedicated linear distinguishers.

1 Introduction

1.1 Motivation

As crucial applications go pervasive, the need for security in RFID and sensor networks is dramatically increasing, which requires secure yet efficiently implementable cryptographic primitives including secret-key ciphers and hash functions. In such constrained environments, the area and power consumption of a primitive usually comes to the fore and standard algorithms are often prohibitively expensive to implement.

Once this research problem was identified, the cryptographic community designed a number of tailored lightweight cryptographic algorithms to specifically address this challenge: stream ciphers like Trivium [18,16], Grain [23,24], and Mickey [3] as well as block ciphers like SEA [43], DESL, DESXL [35], HIGHT [27], mCrypton [36], KATAN/KTANTAN [17], and PRESENT [10] — to mention only a small selection of the lightweight designs.

Rather recently, some significant work on lightweight hash functions has been also performed: [11] describes ways of using the PRESENT block cipher in hashing modes of operation and [1] and [21] take the approach of designing a dedicated lightweight hash function based on a sponge construction [15,7] resulting in two hash functions QUARK and PHOTON.

Among the most prominent security applications targeted by a lightweight hash function are (including the ones requiring preimage security only and collision security only):

- **Lightweight signature schemes:** ECC over $\mathbb{F}_{2^{163}}$ is implementable with just 11.904 GE without key storage after synthesis and around 15.000 GE on a chip [22]. For comparison, the smallest published SHA-256 implementation [32] requires 8.588 GE and the reportedly most compact SHA-3 finalists BLAKE and Grøstl need 13.560 GE [25] and 14.620 GE [44], respectively, to our best knowledge. Hence, adding a hashing engine based on one of these functions to a lightweight ECC implementation nearly doubles the footprint.
- **RFID security protocols** often rely on hash functions [2,40,46]. Some of the applications require collision resistance and some of them do not, just needing preimage security. An interesting case is constituted by keyed message authentication codes (MAC) often used in this context. Here, a lightweight hash function can require less area than a lightweight block cipher in a MAC mode at a fixed level of offline and online security. MACs can be also designed using sponge primitives [8].
- **Random number generation** in hardware is used for ephemeral key generation in public-key schemes, producing random input for cryptographic protocols, and for masking schemes in implementations with protection against side-channel attacks. This frequently needs a preimage-resistant hash function. Using a hash function for pseudorandom number generator (PRNG), given a seed, provides backward security which a block cipher based PRNG (e.g. in OFB mode) does not: Once the key is leaked e.g. through a side-channel attack, the adversary can compute the previous outputs of the block cipher based PRNG. Moreover, the postprocessing of a physical random number generator sometimes includes a preimage-resistant hash function.
- **Post-quantum signature schemes** can be built upon a hash function using Merkle trees [38], [12]. There have been several attempts to efficiently implement it [42,41]. Having a lightweight hash function allows to derive a more compact implementation of the Merkle signature scheme.

However, while for multiple block ciphers, designs have already closely approached the minimum ASIC hardware footprint theoretically attainable, it does not seem the case for some recent lightweight hash functions so far. This article proposes the family of sponge-based lightweight hash functions SPONGENT with a smaller footprint than most existing dedicated lightweight hash functions: PRESENT in hashing modes and QUARK. Its area is comparable to that of PHOTON, though sometimes being slightly more compact. However, a fair comparison in terms of area requirements is a challenging task, since the area occupation is highly dependent on the implementation, technology and tools used. To address this challenge, we provide implementation figures for SPONGENT on four different technologies.

For some SPONGENT variants, similarly to QUARK and PHOTON, a part of this advantage comes from a reduced level of second preimage security, while maintaining the standard level collision resistance. The other SPONGENT variants attain the standard preimage, second preimage and collision security, while having area requirements much lower than those of SHA-1, SHA-2, and SHA-3 finalists. This design subspace has not been specifically addressed by any previous concrete lightweight hash function proposal. Whereas we note that the design ideas of PRESENT in hashing modes, QUARK and PHOTON might be extended to any set of security parameters.

1.2 Design considerations for lightweight hashing

The footprint of a hash function is mainly determined by

1. the number of state bits (incl. the key schedule for block cipher based designs) as well as
2. the size of functional and control logic used in a round function.

For highly serialized implementations (usually used to attain low area and power), the logic size is normally rather small and the state size dominates the total area requirements of the design. Among the recent hash functions, QUARK, while using novel ideas of reducing the state size to minimize (1), does not appear to provide the smallest possible logic size, which is mainly due to the Boolean functions with many inputs used in its round transform. In contrast to that, SPONGENT keeps the round function very simple which reduces the logic size close to the smallest theoretically possible, thus, minimizing (2) and resulting in a significantly more compact design.

As shown in [11], using a lightweight block cipher in a hashing mode (single block length such as Davies-Meyer or double block length such as Hirose) is not necessarily an optimal choice for reducing the footprint, the major restriction being the doubling of the datapath storage requirement due to the feed-forward operation.

At the same time, no feed-forward is necessary for the sponge construction, which is the design approach of choice in this work. In a permutation-based sponge construction, let r be the *rate* (the number of bits input or output per one permutation call), c be the *capacity* (internal state bits not used for input or output), and n be the hash length in bits.

To explore the design space of lightweight hashing, we propose to instantiate the sponge construction with a PRESENT-type permutation. The resulting construction is called SPONGENT and we refer to its various parameterizations as SPONGENT-$n/c/r$ for different hash sizes n, capacities c, and rates r. SPONGENT is a hermetic sponge, i.e., we do not allow the underlying permutation to have any structural distinguishers. More precisely, for five different hash sizes of $n \in \{88, 128, 160, 224, 256\}$, covering most security applications in the field, we consider (up to) three types of preimage and second-preimage security levels:

- **Full preimage and second-preimage security.** The standard security requirements for a hash function with an n-bit output size are collision resistance of $2^{n/2}$ as well as preimage and second-preimage resistance of 2^n. For this, in SPONGENT, we set $r = n$ and $c = 2n$ to obtain SPONGENT-88/176/88, SPONGENT-128/256/128, SPONGENT-160/320/160, SPONGENT-224/448/224, and SPONGENT-256/512/256.
- **Reduced second-preimage security.** The design of [1] as well as the works [7,8,15] convincingly demonstrate that a permutation-based sponge construction can allow to almost halve the state size for $n \geq c$ and reasonably small r. In this case, the preimage and second-preimage resistances are reduced to 2^{n-r} and $2^{c/2}$, correspondingly, while the collision resistance remains at the level of $2^{c/2}$. In most embedded scenarios, where a lightweight hash function is likely to be used, the full second-preimage security is not a necessary requirement. For relatively small rate r, the loss of preimage security is limited. So we take this parametrization in the design of the smallest SPONGENT variants with $n \approx c$ for small r and obtain SPONGENT-88/80/8, SPONGENT-128/128/8, SPONGENT-160/160/16, SPONGENT-224/224/16, and SPONGENT-256/256/16. These five SPONGENT-variants were published in a shortened conference version [9] of this article.
- **Reduced preimage and second-preimage security.** In some applications, the collision security is of concern only and one can abandon the requirement of preimage security to be close to 2^n. In a permutation-based sponge, going for $c = n$ and $r = n/2$, results in the reduction of both the preimage security and second-preimage security to $2^{n/2}$, while maintaining the full collision security of $2^{n/2}$. On the implementation side, this parametrization can yield a favorable ratio between the rate and the permutation size which reduces the time-area product. We use this approach in the design of SPONGENT-160/160/80, SPONGENT-224/224/112, and SPONGENT-256/256/128.

The group of all SPONGENT variants with the same output size of n bits is referred to as SPONGENT-n. The SPONGENT-88 functions are designed for extremely restricted scenarios and low preimage security requirements. They can be used e.g. in some RFID protocols and for PRNGs. SPONGENT-128 and SPONGENT-160 might be used in highly constrained applications with low and middle requirements for collision security. The latter also provides compatibility to the SHA-1 interfaces. The parameters of SPONGENT-224 and SPONGENT-256 correspond to those of a subset of SHA-2 and SHA-3 to make SPONGENT compatible to the standard interfaces in usual lightweight embedded scenarios.

1.3 Organization of the article

The remainder of the article is organized as follows. Section 2 describes the design of SPONGENT and gives a design rationale. Section 3 presents some results of security analysis, including proven lower bounds on the number of differentially active S-boxes, best differential characteristics found, rebound attacks, and linear attacks. In Section 4, the implementation results are given for a range of trade-offs. We conclude in Section 5.

2 The design of SPONGENT

SPONGENT is a sponge construction based on a wide PRESENT-type permutation. Given a finite number of input bits, it produces an n-bit hash value. A design goal for SPONGENT is to follow the hermetic sponge strategy (no structural distinguishers for the underlying permutation are allowed).

2.1 Permutation-based sponge construction

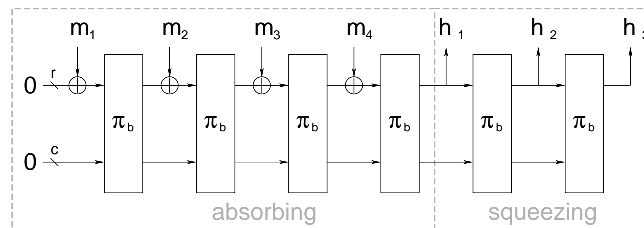


Fig. 1. Sponge construction based on a b-bit permutation π_b with capacity c bits and rate r bits. m_i are r-bit message blocks. h_i are parts of the hash value.

SPONGENT relies on a sponge construction – a simple iterated design that takes a variable-length input and can produce an output of an arbitrary length based on a permutation π_b operating on a state of a fixed number b of bits. The size of the internal state $b = r + c \geq n$ is called *width*, where r is the *rate* and c the *capacity*.

The sponge construction proceeds in three phases (see also Figure 1):

- **Initialization phase:** the message is padded by a single bit 1 followed by a necessary number of 0 bits up to a multiple of r bits (e.g., if $r = 8$, then the 1-bit message '0' is transformed to '01000000'). Then it is cut into blocks of r bits.

- **Absorbing phase:** the r-bit input message blocks are xored into the first r bits of the state, interleaved with applications of the permutation π_b.
- **Squeezing phase:** the first r bits of the state are returned as output, interleaved with applications of the permutation π_b, until n bits are returned.

In SPONGENT, the b-bit 0 is taken as the initial value before the absorbing phase. In all SPONGENT variants, except SPONGENT-88/80/8, the hash size n equals either capacity c or $2c$. The message chunks are xored into the r rightmost bit positions of the state. The same r bit positions form parts of the hash output.

Let a permutation-based sponge construction have $n \geq c$ and $c/2 > r$ which is fulfilled for the parameter choices of most of the SPONGENT variants. Then the works [7,8,15] imply the preimage security of 2^{n-r} as well as the second preimage and collision securities of $2^{c/2}$ if this construction is hermetic (that is, if the underlying permutation does not have any structural distinguishers). The best preimage attack we are aware of in this case has a computational complexity of $2^{n-r} + 2^{c/2}$. Later, this work is extended in [21] and preimage security is defined more generalized form: $min(2^{min(n,\ c+r)}, max(2^{min(n-r,\ c)}, 2^{c/2}))$.

For permutation-based sponge constructions with $n < c$ and $c/2 \leq r$ such as the remaining SPONGENT variants, it follows from the same works that the second preimage security is 2^n and collision security is $2^{c/2}$. The previous preimage attack also works for this case hence we claim that the preimage security is $min(2^n, max(2^{n-r}, 2^{c/2}))$ since $n - r < c$.

2.2 Parameters

We propose 13 variants of SPONGENT with five different hash output lengths at multiple security levels, see Table 1.

Table 1. 13 SPONGENT variants

	n (bit)	b (bit)	c (bit)	r (bit)	R number of rounds	security(bit) pre.	2nd pre.	col.
SPONGENT-88/80/8	88	88	80	8	45	80	40	40
SPONGENT-88/176/88	88	264	176	88	135	88	88	44
SPONGENT-128/128/8	128	136	128	8	70	120	64	64
SPONGENT-128/256/128	128	384	256	128	195	128	128	64
SPONGENT-160/160/16	160	176	160	16	90	144	80	80
SPONGENT-160/160/80	160	240	160	80	120	80	80	80
SPONGENT-160/320/160	160	480	320	160	240	160	160	80
SPONGENT-224/224/16	224	240	224	16	120	208	112	112
SPONGENT-224/224/112	224	336	224	112	170	112	112	112
SPONGENT-224/448/224	224	672	448	224	340	224	224	112
SPONGENT-256/256/16	256	272	256	16	140	240	128	128
SPONGENT-256/256/128	256	384	256	128	195	128	128	128
SPONGENT-256/512/256	256	768	512	256	385	256	256	128

2.3 PRESENT-type permutation

The permutation $\pi_b : \mathbb{F}_2^b \rightarrow \mathbb{F}_2^b$ is an R-round transform of the input STATE of b bits that can be outlined at a top-level as:

for $i = 1$ to R **do**
  STATE ← ɹǝʇunoƆl$_b(i)$ ⊕ STATE ⊕ lCounter$_b(i)$
  STATE ← sBoxLayer$_b$(STATE)
  STATE ← pLayer$_b$(STATE)
end for

where sBoxLayer$_b$ and pLayer$_b$ describe how the STATE evolves. For ease of design, only widths b with $4|b$ are allowed. The number R of rounds depends on block size b and can be found in Subsection 2.2 (see also Table 1). lCounter$_b(i)$ is the state of an LFSR dependent on b at time i which yields the round constant in round i and is added to the rightmost bits of STATE. ɹǝʇunoƆl$_b(i)$ is the value of lCounter$_b(i)$ with its bits in reversed order and is added to the leftmost bits of STATE.

The following building blocks are generalizations of the PRESENT structure to larger b-bit widths:

1. sBoxLayer$_b$: This denotes the use of a 4-bit to 4-bit S-box $S : \mathbb{F}_2^4 \rightarrow \mathbb{F}_2^4$ which is applied $b/4$ times in parallel. The action of the S-box in hexadecimal notation is given by the following table:

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	E	D	B	0	2	1	4	F	7	A	8	5	9	C	3	6

2. pLayer$_b$: This is an extension of the (inverse) PRESENT bit-permutation and moves bit j of STATE to bit position $P_b(j)$, where

$$P_b(j) = \begin{cases} j \cdot b/4 \mod b-1, & \text{if } j \in \{0, \ldots, b-2\} \\ b-1, & \text{if } j = b-1. \end{cases}$$

and can be seen in Figure 2.

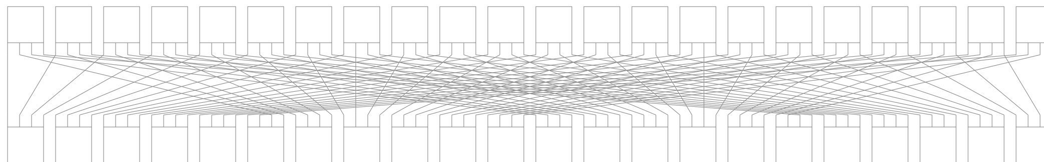


Fig. 2. The bit permutation layer of SPONGENT-88 at the example of pLayer$_{88}$

3. lCounter$_b$: This is one of the four $\lceil \log_2 R \rceil$-bit LFSRs. The LFSR is clocked once every time its state has been used and its final value is all ones. If ζ is the root of unity in the corresponding binary finite field, the n-bit LFRSs defined by the polynomials given below are used for the SPONGENT variants.

LFSR size (bit)	Primitive Polynomial
6	$\zeta^6 + \zeta^5 + 1$
7	$\zeta^7 + \zeta + 1$
8	$\zeta^8 + \zeta^4 + \zeta^3 + \zeta^2 + 1$
9	$\zeta^9 + \zeta^4 + 1$

Table 2 provides sizes and initial values of all the LFSRs.

Table 2. Initial values of lCounter_b for all SPONGENT variants

	LFSR size (bit)	Initial Value (hex)
SPONGENT-88/80/8	6	05
SPONGENT-88/176/88	8	D2
SPONGENT-128/128/8	7	7A
SPONGENT-128/256/128	8	FB
SPONGENT-160/160/16	7	45
SPONGENT-160/160/80	7	01
SPONGENT-160/320/160	8	A7
SPONGENT-224/224/16	7	01
SPONGENT-224/224/112	8	52
SPONGENT-224/448/224	9	105
SPONGENT-256/256/16	8	9E
SPONGENT-256/256/128	8	FB
SPONGENT-256/512/256	9	015

2.4 Design rationale

The overall design approach for SPONGENT is to target low area while favoring simplicity.

The 4-bit S-box is the major block of functional logic in a serial low-area implementation of SPONGENT. It fulfills the PRESENT design criteria in terms of differential and linear properties [10]. Moreover, any linear approximation over the S-box involving only single bits both in the input and output masks is unbiased. This aims to restrict the linear hull effect discovered in round-reduced PRESENT.

The function of the bit permutation pLayer is to provide good diffusion, by acting together with the S-box, while having a limited impact on the area requirements. This is its main design goal, while a bit permutation may occupy additional space in silicon. The counters lCounter and $\reflectbox{lCounter}$ are mainly aimed to prevent sliding properties and make prospective cryptanalysis approaches using properties like invariant subspaces [34] more involving.

The structures of the bit permutation and the S-box in SPONGENT make it possible to prove the following differential property (see Subsection 3.1 for the proof):

Theorem 1. *Any 5-round differential characteristic of the underlying permutation of* SPONGENT *with* $b \geq 64$ *has a minimum of* 10 *active S-boxes. Moreover, any 6-round differential characteristic of the underlying permutation of* SPONGENT *with* $b \geq 256$ *has a minimum of* 14 *active S-boxes.*

The concept of counting active S-boxes is central to the differential cryptanalysis. The minimum number of active S-boxes relates to the maximum differential characteristic probability of the construction. Since in the hash setting there are no random and independent key values added between the rounds, this relation is not exact (in fact that it is even not exact for most practical keyed block ciphers). However, differentially active S-boxes are still the major technique used to evaluate the security of SPN-based hash functions.

An important property of the SPONGENT S-box is that its maximum differential probability is 2^{-2}. This fact and the assumption of the independency of difference propagation in different rounds yield an upper bound on the differential characteristic probability of 2^{-20} over 5 rounds and of 2^{-28} over 6 rounds for $b \geq 256$ which follows from the claims of Theorem 1.

Theorem 1 is used to determine the number R of rounds in permutation π_b: R is chosen in a way that π_b provides at least b active S-boxes. Other types of analysis are performed in the next section.

3 Security Analysis

In this section, we discuss the security of SPONGENT against known cryptanalytic attacks by applying the most important state-of-the-art methods of cryptanalysis and investigating their complexity.

Table 3. Differential characteristics with lowest numbers of differentially active S-boxes (ASN). The probabilities are calculated assuming the independency of round computations.

# of rounds	SPONGENT-88/80/8		SPONGENT-128/128/8		SPONGENT-160/160/16		SPONGENT-224/224/16		SPONGENT-256/256/16	
	ASN	Prob	ASN	Prob	ASN	Prob	ASN	Prob	ASN	Prob
5	10	2^{-21}	10	2^{-22}	10	2^{-21}	10	2^{-21}	10	2^{-20}
10	20	2^{-47}	29	2^{-68}	20	2^{-50}	20	2^{-43}	-	–
15	30	2^{-74}	-	-	30	2^{-79}	30	2^{-66}	-	–

# of rounds	SPONGENT-160/160/80	
	ASN	Prob
5	10	2^{-21}
10	20	2^{-43}
15	30	2^{-66}

# of rounds	SPONGENT-224/224/112		SPONGENT-256/256/128	
	ASN	Prob	ASN	Prob
6	14	2^{-28}	14	2^{-28}

# of rounds	SPONGENT-88/176/88		SPONGENT-128/256/128		SPONGENT-160/320/160		SPONGENT-224/448/224		SPONGENT-256/512/256	
	ASN	Prob	ASN	Prob	ASN	Prob	ASN	Prob	ASN	Prob
6	14	2^{-28}	14	2^{-28}	14	2^{-28}	14	2^{-28}	14	2^{-28}

3.1 Resistance against differential cryptanalysis

Here we analyze the resistance of SPONGENT against differential attacks where Theorem 1 plays a key role providing a lower bound on the number of active S-boxes in a differential characteristic. The similarities of the SPONGENT permutations and the basic PRESENT cipher allow to reuse some of the results obtained for PRESENT in [10]. More precisely, the results on the number of differentially active S-boxes over 5 and 6 rounds will hold for all respective SPONGENT variants which is reflected in Theorem 1. The proof of the Theorem 1 is as follows:

Proof. [*Theorem 1*] The statements for SPONGENT variants with $64 \leq b \leq 255$ can directly be proven by applying the same technique used in [10, Appendix III]. The proof of the 6-round bounds for SPONGENT variants with $b \geq 256$ in Theorem 1 is based on some extended observations. Here, we will only give the proof for when the width, b, is a multiple of 64 bits, i.e., $b = 64n$. The proof for other b values can also be obtained by making use of the observations given below. Since the proof is specific to each b and hence more tedious, we do not present them here.

We obtain n groups and $4n$ subgroups by calling each four consecutive S-boxes as a *subgroup* and each sixteen consecutive S-boxes as a *group*. To be more specific: subgroup i is comprised of the S-boxes $[4(i-1)\dots 4i-1]$ and similarly group j has the subgroups $[4(j-1)\dots 4j-1]$. (see Figure 3). By examining the substitution and linear layers, one can make the following observations:

1. The S-box of SPONGENT is such that a difference in single input bit causes a difference in at least two output bits or vice versa.
2. The input bits to an S-box come from four distinct S-boxes of the same subgroup.
3. The input bits to a subgroup of four S-boxes come from 16 distinct S-boxes of the same group.
4. The input bits to a group of 16 S-boxes come from 64 different S-boxes.
5. The four output bits from a particular S-box enter four distinct S-boxes, each of which belongs to a distinct group of S-boxes in the subsequent round.
6. The output bits of S-boxes in distinct groups go to distinct S-boxes in distinct subgroups.
7. The output bits of S-boxes in distinct subgroups go to distinct S-boxes.

For the latter statement (SPONGENT-256), one has to deal with more cases. Consider six consecutive rounds of SPONGENT ranging from i to $i+5$ for $i \in [1\dots 155]$. Let D_j be the number of active S-boxes in round j. If $D_j \geq 3$, for $i \leq j \leq i+5$, then the theorem trivially holds. So let us suppose that one of D_j is equal to one first and to two then. We have the following cases:

Case $D_{i+2} = 1$. By using observation 1, we can deduce that $D_{i+1} + D_{i+3} \geq 3$ and all active S-boxes of round $i+1$ belong to the same subgroup from observation 2. Each of these active S-boxes have only a single bit difference in their output. So, according to observation 3 we have that $D_i \geq 2D_{i+1}$. Conversely, according to observation 5, all active S-boxes in round $i+3$ belong to distinct groups and have only a single bit difference in their input. So, according to observation 6, we have that $D_{i+4} \geq 2D_{i+3}$. Moreover, all active S-boxes in round $i+4$ belong to distinct subgroups and have only a single bit difference in their input. Thus, by using observation 7, we obtain that $D_{i+5} \geq 2D_{i+4}$ and can conclude that $\sum_{j=i}^{i+5} D_j \geq 1 + 3 + 2 \times 3 + 4D_{i+3} \geq 14$.

Case $D_{i+3} = 1$ If $D_{i+2} = 1$ we can refer to the first case. So, suppose that $D_{i+2} \geq 2$. According to the observation 2, all active S-boxes of round $i+2$ belong to the same subgroup and each of these active S-boxes has only a single bit difference in their output. Thus, according to observation 3, $D_{i+1} \geq 2D_{i+2} \geq 4$. Since all active S-boxes in round $i+1$ belong to distinct S-boxes of the same group and have only a single bit difference in their input, according to observation 4, we have that $D_i \geq 2D_{i+1}$. On the opposite, D_{i+4} and D_{i+5} can get one and two as a minimum value, respectively. Together this gives $\sum_{j=i}^{i+5} D_j \geq 8 + 4 + 2 + 1 + 1 + 2 \geq 18$.

Case $D_{i+1} = 1$ If $D_{i+2} = 1$, then we can refer to the first case. Thus, suppose that $D_{i+2} \geq 2$. According to observation 5, all active S-boxes in round $i+2$ belong to distinct groups and have only a single bit difference in their input. Thus, according to observation 6, we have that $D_{i+3} \geq 2D_{i+2}$. Since all active S-boxes in round $i+3$ belong to distinct subgroups and have only a single bit

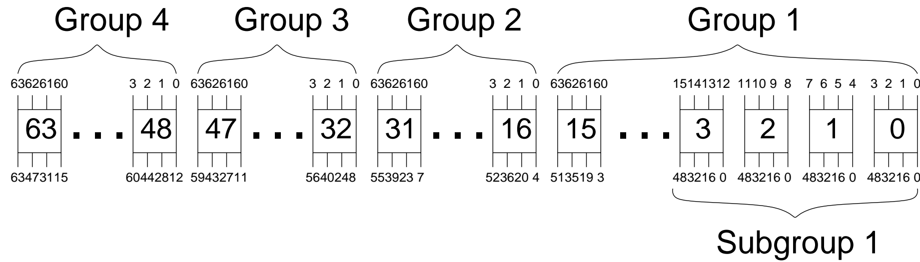


Fig. 3. The grouping and subgrouping of S-boxes for $b = 256$. The input numbers indicate the S-box origin from the previous round and the output numbers indicate the destination S-box in the following round.

difference in their input. Therefore, according to observation 7, we have that $D_{i+4} \geq 2D_{i+3}$. To sum up, $\sum_{j=i}^{i+5} D_j \geq 1+1+2+4+8+D_{i+5} \geq 16+D_{i+5} \geq 17$, since $D_{i+4} > 0$ implies that $D_{i+5} \geq 1$.
Case $D_{i+4} = 1$ If $D_{i+3} = 1$, then we can refer to the second case. So, suppose that $D_{i+3} \geq 2$. According to the observation 2, all active S-boxes of round $i+3$ belong to the same subgroup and each of those active S-boxes has only a single bit difference in their output. Therefore, according to observation 3, we have that $D_{i+2} \geq D_{i+3}$. Since, all active S-boxes in round $i+2$ belong to distinct S-boxes of the same group and have only a single bit difference in their input, according to observation 4, we have that $D_{i+1} \geq 2D_{i+2}$. Since $D_{i+1} > 0$, $D_i \geq 1$. Thus, we can conclude that $\sum_{j=i}^{i+5} D_j \geq D_i + 8 + 4 + 2 + 1 + 1 \geq D_i + 16 \geq 17$.
Cases $D_i = 1$ and $D_{i+5} = 1$ are similar to the those for the third and fourth cases.

So far we have considered all paths including one active S-box in one of the rounds and obtained 14 as the minimum number of active S-boxes. But if there exists a path that has two active S-boxes in each round, then the lower bound would be 12. For this purpose, without loss of generality, assume:

$D_{i+1} = D_{i+2} = D_{i+3} = 2$ The two active S-boxes in $i+2$ are either in the same subgroup or in different subgroups. For the former, from observations 3 and 7, we know that they have single bit of differences coming from two different subgroups of the same group in round $i+1$. From observation 1, these two S-boxes have at least two bits of input difference, hence we obtain $D_i = 4$ by observation 2 and 3. Furthermore the two S-boxes in round $i+2$ have two bits of output difference by observation 1. Hence, in round $i+3$, the active S-boxes have two bits of input and they are in distinct groups by observation 5. Therefore, it is possible to have $D_{i+4} = 2$ in distinct subgroups. Hence by using observation 7, we obtain $D_{i+5} = 4$. Thus, we can conclude that $\sum_{j=i}^{i+5} D_j \geq 4 + 2 + 2 + 2 + 2 + 4 \geq 16$.

For the latter, the two active S-boxes in round $i+1$ must have two bits of input and by observation 2 their input bits should be coming from distinct S-boxes in the same subgroup. So, the problem is reduced to the former case with one round of shift, and we can immediately say that $D_i = 2$ and $D_{i+4} = 4$. Hence by using observation 7, we obtain $D_{i+5} = 4$. Thus, we can conclude that $\sum_{j=i}^{i+5} D_j \geq 2 + 2 + 2 + 2 + 4 + 4 \geq 16$.

Based on these results, we conclude that the longest run with two active S-boxes in each round is four rounds, and the number of active S-boxes cannot be less than 14.

For all SPONGENT variants, we found that those 5- and 6-round bounds are actually tight. We present the characteristics attaining them in Table 3. Additionally, we identified iterative charac-

teristics with up to three active S-boxes in each round. For SPONGENT-88/80/8 the probability of the 2-round iterative characteristic is 2^{-14}. Whereas, for SPONGENT-160/160/80 and SPONGENT-224/224/16 (since they have the same state size) the probability for two rounds is 2^{-9}.

3.2 Collision attacks

A natural approach to obtain a collision for a sponge construction is to inject a difference in a message block and then cancel the propagated difference by a difference in the next message block, i.e., $(0\ldots0||\Delta m_i) \xrightarrow{\pi} (0\ldots0||\Delta m_{i+1})$. For this purpose, we follow a narrow trail strategy using truncated differential characteristics. We start from a given input difference (some difference restricted to S-boxes that the message block is xored into) and look for all paths that go to a fixed output difference (also located in the bitrate part of the state). Based on our experiments, even by using truncated differential characteristics, the probability of such a path is quite low and it is not possible to attack the full number of rounds.

Rebound attack: The rebound attack [37], a recent technique for cryptanalysis of hash functions, is applicable to both block cipher based and permutation based hash constructions. It consists of two main steps: the inbound phase where the freedom is used to connect the middle rounds by using the match-in-the-middle technique and the outbound phase where the connected truncated differentials are calculated in both forward and backward directions. It has been mostly used to improve the results on AES-based algorithms (ECHO [6], Grøstl [20], LANE [28], Whirlpool [5]), but it has also been successfully applied to similar permutations (Luffa [30], Keccak [19]).

Compared to the other algorithms the rebound attack has been successfully applied to, the design of SPONGENT imposes some limitations. First of all, since the permutation is bit oriented, and not byte oriented, it might be non-trivial to find the path followed by a given input difference and to determine the number of active S-boxes after several rounds. This is mainly because the difference propagation strictly depends on the values of the passive part of the state. Moreover, the probability that two inbound phases match requires more detailed analysis.

For SPONGENT-88/80/8, we tried to implement the rebound attack. We found that for up to 4 rounds it is impossible to obtain a characteristic that matches in the middle with the available degrees of freedom coming from the message bits. Starting from 5 rounds such a characteristic is possible, but we cannot generate enough pairs by using only a difference in the message bits when the whole state is active in the matching phase. Since the expected probability of matching the inbound phases is $2^{-b/4}$ (where $b/4$ is the number of S-boxes) and the available degree of freedom is only 2^{2r}, this argument is also valid for SPONGENT-128/128/8, SPONGENT-160/160/16, SPONGENT-224/224/16, and SPONGENT-256/256/16. For other SPONGENT variants there exist enough degrees of freedom but we are not aware of any dedicated attack.

Bound considerations for the rebound attack: The adversary might try to find a way to attack by using multiple inbounds with a sparse differential. Therefore, to explore the security against multiple inbound phases, we put the adversary into a best-case scenario as follows.

We know that there exists no differential characteristic over five rounds with the number of active S-boxes less than 10 for all SPONGENT variants. We can also deduce lower bounds on the number of active S-boxes for $1, 2, 3,$ and 4 rounds as $1, 2, 4$ and 6, respectively. Then a bound on

the minimum number of active S-boxes, hence the probability of a differential characteristic, for any number of rounds can be approximated by combining these bounds.[4]

The desired bit security level for a sponge construction with respect to collision attacks is $c/2$. From now on we assume that the complexity of each inbound phase is equal to $c/2$ and at least one active S-box matches between two inbound phases (with probability 2^{-8}). Let n_{in} be the number of inbound phases then we have to generate $n_{elm} = 2^{8 \cdot (n_{in}-1)/n_{in}}$ elements for each inbound phase. Let p denote the probability of each inbound phase, then p can be at least $2^{-(c/2 - \lceil log_2(n_{elm}) \rceil)}$ and we can compute the number of rounds in each inbound phase by using the given bounds above.

Under these assumptions, the maximum number of rounds per inbound phase and the percentage of the total number of rounds attacked is given in Table 4.

Table 4. Bounds for rebound attack

	2 Inbounds		3 Inbounds	
	rounds /inbound	attacked rounds(%)	rounds /inbound	attacked rounds(%)
SPONGENT-88/80/8	9	40.00	9	60.00
SPONGENT-88/176/88	10	14.81	9	20.00
SPONGENT-128/128/8	15	42.86	14	60.00
SPONGENT-128/256/128	14	14.36	13	20.00
SPONGENT-160/160/16	19	42.22	19	63.33
SPONGENT-160/160/80	19	31.67	19	47.50
SPONGENT-160/320/160	17	14.17	16	20.00
SPONGENT-224/224/16	28	46.67	27	67.50
SPONGENT-224/224/112	23	27.06	23	40.59
SPONGENT-224/448/224	23	13.53	23	20.29
SPONGENT-256/256/16	28	40.00	27	57.86
SPONGENT-256/256/128	28	28.72	27	41.54
SPONGENT-256/512/256	28	14.55	27	21.04

3.3 Preimage resistance

Here we apply a meet-in-the-middle approach to obtain preimages on SPONGENT. The attack has two main steps: Pre-computation and matching phase. Complexity of the attack is dominated by pre-computation phase.

Since the hash size is n bits, and the data is extracted in r bit chunks, there exists n/r rounds in the squeezing phase. To be able to compute the data backwards in the absorbing phase, we need to know not only h_i's but also d_i values to obtain the input value of the permutation π, where h_i denotes the part of the hash value and d_i is the concatenated part to h_i. The algorithm is as follows:

1. **Pre-computation:** We know that $\pi^{-1}(h_{i+1}, d_{i+1}) = (h_i, d_i)$ for each i in the squeezing phase. Since h_i (r-bits) is already fixed, the probability of finding such d_i is 2^{-r}. Therefore, we start with $2^{((n/r)-1) \cdot r} = 2^{n-r}$ different $d_{n/r}$ values to have a solution for d_1.

[4] Note that, Table 3 shows that these bounds might be optimistic.

2. **Match-in-the-middle:** Choose k such that $k \cdot r \geq c/2$. Then
   - Generate $2^{c/2}$ elements in the backward direction by using (h_1, d_1) and possible values for $m_{k+2}, \ldots, m_{2k+1}$ and store them in a table.
   - Generate $2^{c/2}$ elements in the forward direction by using possible values for $m_1, \ldots, m_k$ and compare with list in the previous step to find a match of c bits (corresponding to capacity) in the middle.
   - Obtain m_{k+1} by xor-ing the r bits (corresponding to bitrate) for the matching elements.

In the pre-computation part, we obtain the required value d_1 to compute the data backwards in the absorbing phase by 2^{n-r} computations. We need $2^{c/2}$ memory to store the elements generated in the second step and $2^{c/2}$ computations are needed to find a full match. These results are exactly those obtained in ("Errata for Keccak presentation") and mentioned in (extended Photon paper) which extends the bounds given in [15] for $c > n$. The results, together with the parameter k, are given in Table 5.

Table 5. Meet-in-the-middle attack results for SPONGENT

	k	Time Complexity $max(2^{n-r}, 2^{c/2})$	Memory Complexity $(2^{c/2})$
SPONGENT-88/80/8	5	2^{80}	2^{40}
SPONGENT-88/176/88	1	2^{88}	2^{88}
SPONGENT-128/128/8	8	2^{120}	2^{64}
SPONGENT-128/256/128	1	2^{128}	2^{128}
SPONGENT-160/160/16	5	2^{144}	2^{80}
SPONGENT-160/160/80	1	2^{80}	2^{80}
SPONGENT-160/320/160	1	2^{160}	2^{160}
SPONGENT-224/224/16	7	2^{208}	2^{112}
SPONGENT-224/224/112	1	2^{112}	2^{112}
SPONGENT-224/448/224	1	2^{224}	2^{224}
SPONGENT-256/256/16	8	2^{240}	2^{128}
SPONGENT-256/256/128	1	2^{128}	2^{128}
SPONGENT-256/512/256	1	2^{256}	2^{256}

Note that, if $c \leq n - r$, it is sufficient to try all possible 2^c values to construct the whole state in order to obtain a preimage, hence it provides an upper bound for the preimage resistance. If we combine the results we obtain $max(2^{min(n-r,c)}, 2^{c/2})$ and it can be generalized into the form: $min(2^{min(n,\ c+r)}, max(2^{min(n-r,\ c)}, 2^{c/2}))$. Here, $2^{min(n,\ c+r)}$ computations will be necessary depending on the permutation size when the generic attack, defined above, fails.

3.4 Linear attacks

The most successful attacks, the attacks that can break the highest number of rounds, for the block cipher PRESENT are those based on linear approximations. In particular the multi-dimensional linear attack [13] and the statistical saturation attack [14] claim to break up to 26 rounds. It was shown in [33] that both attacks are closely related. Moreover, the main reason why these attacks are the most successful attacks on PRESENT so far, is the existence of many linear trails with only one active S-box

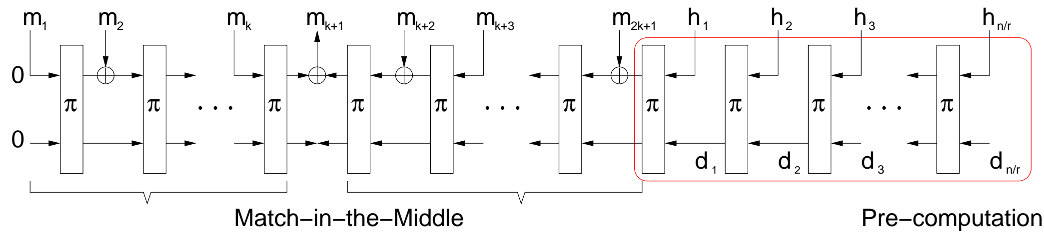


Fig. 4. Meet-in-the-middle attack against sponge construction

in each round. It is not immediately clear how linear distinguishers on the SPONGENT permutation π_b could be transferred into collision or (second) pre-image attacks on the hash function. However, as we claim that SPONGENT is a hermetic sponge construction, the existence of such distinguishers has to be excluded. So the SPONGENT S-box was chosen in a way that allows for at most one trail with this property given a linear approximation.

Unlike for the block cipher PRESENT, where the key determines the actual linear correlation between an input and an output mask, for the permutation π_b we can compute the actual linear trail contribution for all trails with only one active S-box in every round. Each such trail over w rounds has a correlation of $\pm 2^{-2w}$ and for each trail determining the sign is easy. More concretely, one can easily compute a $b \times b$ matrix M_t over the rationals such that the entry at position i, j is the correlation coefficient for round t for the linear trail with input mask e_i and output mask e_j. Here e_i (resp. e_j) is the unit vector with a single 1 at position i (resp. j). Note that the matrices M_t are sparse and all very similar, the only difference is caused by the round constant, which induces sign changes at a few positions only.

Given those matrices, it is now possible to compute the maximal linear correlation contribution for those one bit intermediate masks for all one bit input and output masks. For w rounds we simply compute $M^{(w)} = \prod_{i=1}^{w} M_i$ and the maximal correlation is given by $c_w := \max_{i,j} |M_{ij}^{(w)}|$. We compute this value for all SPONGENT variants. Table 6 summarizes those results. Most importantly, this table shows the maximal number of rounds w where the trail contributions is still larger than or equal to $2^{-b/2}$. Beyond this number of rounds, it seems unlikely that distinguishers based on linear approximations exist. For most SPONGENT variants, the best linear hull based on single-bit masks has exactly one linear trail.

4 Hardware Implementations

In this section we provide a wide range of hardware figures by evaluating all of the 13 SPONGENT variants in detail. Not only a comprehensive hardware evaluation is of our primary interest, we also further elaborate on the importance of having the unified benchmarking platform for comparing different lightweight designs. To further stress on the latter issue, we provide the results using four different CMOS technologies. For a thorough evaluation of area, throughput, maximum frequency, and power consumption, we use the UMC 130 nm CMOS generic process (UMC130) provided by the Faraday corporation[5]. Moreover, we provide the estimates of the circuit area using three

[5] The choice of the UMC130 library for our hardware implementation is driven by the size of a single scan flip-flop. One scan flip-flop in our UMC180 is 6.67 GE large, while in UMC130 it consists of 6.25 GE. In [21], for example, a scan flip-flop of only 6 GE has been reported.

Table 6. Results of linear trail correlation based on one bit masks

	b	max w with $c_w \geq 2^{-b/2}$	R	$\log_2 c_R$
SPONGENT-88/80/8	88	22	45	−90
SPONGENT-88/176/88	264	66	135	−270
SPONGENT-128/128/8	136	34	70	−140
SPONGENT-128/256/128	384	96	195	−388.4
SPONGENT-160/160/16	176	44	90	−180
SPONGENT-160/160/80	240	60	120	−240
SPONGENT-160/320/160	480	122	240	−473.7
SPONGENT-224/224/16	240	60	120	−240
SPONGENT-224/224/112	336	84	170	−340
SPONGENT-224/448/224	673	169	340	−675.3
SPONGENT-256/256/16	272	68	140	−280
SPONGENT-256/256/128	384	96	195	−388.4
SPONGENT-256/512/256	768	192	385	−770

other libraries: UMC 180 nm CMOS generic process (UMC180), an open source NANGATE 45 nm CMOS technology (NANGATE45) [39] as well as the advanced 90 nm CMOS standard cell library provided by NXP Semiconductors (NXP90).

In order to provide very compact implementations, we first focus on serialized designs. We explore different datapath sizes (d) for each of the SPONGENT variants and we focus on $d \in \{4, 8, \frac{b}{2}, b\}$. An architecture representing our serialized datapath is depicted in Fig. 5(a). The control logic consists of a single counter for the cycle count and some extra combinational logic to drive the select signals of the multiplexers. In order to further reduce the area we use so-called scan flip-flops, which act as a combination of two input multiplexer and an ordinary D flip-flop[6]. Instead of providing a reset signal to each flip-flop separately, we use two zero inputs at the multiplexers M_1 and M_2 to correctly initialize all the flip-flops. This additionally reduces hardware resources, as the scan flip-flops with a reset input approximately require an additional GE per bit of storage. With g_i we denote the value of $\text{lCounter}_b(i)$ in round i. $\text{lCounter}_b(i)$ is implemented as an LFSR as explained in Subsection 2.3. The input of the message block m, denoted with dashed line, is omitted in some cases, i.e. $d \geq r$. The pLayer module requires no additional logic except some extra wiring.

Using the most serialized implementation, the smallest variant of the SPONGENT family, SPONGENT-88/80/8, can be implemented using only 738 GE. Even the largest member of the family, SPONGENT-256/512/256, consumes only 5.1 kGE, while providing 256 bits of preimage and second preimage security, and 128 bits of collision resistance. Though some of this advantage is at the expense of a performance reduction, also less serialized (and, thus, faster) implementations result in area requirements significantly lower than 10 kGE. To demonstrate this, we implement all the SPONGENT variants as depicted in Fig. 5(b). Every round now requires a single clock cycle, therefore resulting in faster, yet rather compact designs.

[6] Scan flip-flops are typically used to provide scan-chain based testability of the circuit. Due to the security issues of scan-chain based testing [47], other methods such as Built-In-Self-Test (BIST) are recommended for testing the cryptographic hardware.

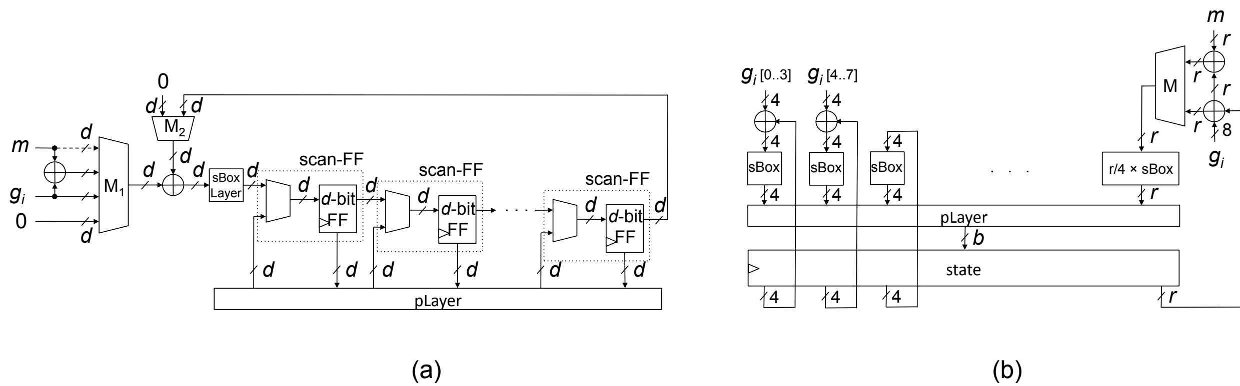


Fig. 5. Hardware architecture representing (a) serial datapath (b) parallel datapath of the SPONGENT variants

Another courtesy of our proposal is the result of 5 times unrolled design of SPONGENT variants which, all running at the maximum frequency of about 600 MHz, provide a throughput between 360 Mbps and 2 Gbps (depending on the variant) and consume between 5 kGE and 48 kGE.

Next, we present the obtained hardware figures for all of the SPONGENT variants. For the purpose of extensive hardware evaluation we use Synopsys Design Compiler version D-2010.03-SP4 and target the High-Speed UMC 130 nm CMOS generic process provided by Faraday Technology Corporation (fsc0h_d_tc).

The power is estimated by observing the internal switching activity of the complete design. Using Mentor Graphics ModelSim version 10.0 SE, we simulate the circuits' behavior for very long messages and generate the VCD (Value Change Dump) files. The VCD files are then converted to the backward SAIF (Switching Activity Interchange Format) files and used within Synopsys Design Compiler for the accurate estimation of the mean power consumption. A typical frequency of 100 kHz is used for all measurements.

Table 7 reports hardware figures obtained using the aforementioned methodology. For the sake of comparison, we include figures for several state-of-the-art lightweight hash functions. We also include two out of five SHA-3 finalists for which the data of compact hardware implementations is publicly available. We do not compare our design with software-like solutions that benefit from using an external memory for storing the intermediate data. Figure 6 illustrates the wide spectrum of our explored design space, where a typical trade-off between speed and area is scrutinized.

4.1 A Fair Comparison – Mission (Im)possible

A fair comparison of hardware performance between different designs has already been discussed in the literature [17,4]. It is rather obvious that such comparison is only possible once the highly optimized designs are implemented on the same hardware platform, using the same standard cell library and the same synthesis tools (including the design flow scripts). However, mainly due to the licensing issues and the designer's preference to use a certain software package, this becomes a very difficult task in practice.

To partially address this issue and in order to avoid any ambiguity we provide Table 8 with area requirements of the basic building cells from our UMC130 library. The library contains many

Table 7. Hardware performance of the SPONGENT family and comparison with state-of-the-art lightweight hash designs. The nominal frequency of 100 kHz is assumed in all cases and the power consumption is therefore adjusted accordingly.

Hash function	Security (bit)			Hash	Cycles	Datapath	Process	**Area**	**Throughput**	**Power***
	Pre.	Coll.	2nd Pre.	(bit)		(bit)	(μm)	(GE)	(kbps)	(μW)
SPONGENT-88/80/8	80	40	40	88	990	4	0.13	738	0.81	1.57
					45	88	0.13	1127	17.78	2.31
SPONGENT-88/176/88	88	44	88	88	8910	4	0.13	1912	0.99	3.4
					135	264	0.13	3450	65.19	7.5
SPONGENT-128/128/8	120	64	64	128	2380	4	0.13	1060	0.34	2.20
					70	136	0.13	1687	11.43	3.58
SPONGENT-128/256/128	128	64	128	128	18720	4	0.13	2641	0.68	6.1
					195	384	0.13	5011	65.64	10.9
SPONGENT-160/160/16	144	80	80	160	3960	4	0.13	1329	0.40	2.85
					90	176	0.13	2190	17.78	4.47
SPONGENT-160/160/80	80	80	80	160	7200	4	0.13	1730	1.11	3.4
					120	240	0.13	3139	66.67	6.8
SPONGENT-160/320/160	160	80	160	160	28800	4	0.13	3264	0.56	8.2
					240	480	0.13	6237	66.67	13.6
SPONGENT-224/224/16	208	112	112	224	7200	4	0.13	1728	0.22	3.73
					120	240	0.13	2903	13.33	5.97
SPONGENT-224/224/112	112	112	112	224	14280	4	0.13	2371	0.78	5.0
					170	336	0.13	4406	65.88	9.6
SPONGENT-224/448/224	224	112	224	224	57120	4	0.13	4519	0.39	11.5
					340	672	0.13	8726	65.88	19.2
SPONGENT-256/256/16	240	128	128	256	9520	4	0.13	1950	0.17	4.21
					140	272	0.13	3281	11.43	6.62
SPONGENT-256/256/128	128	128	128	256	18720	4	0.13	2641	0.68	6.1
					195	384	0.13	5011	65.64	10.9
SPONGENT-256/512/256	256	128	256	256	73920	4	0.13	5110	0.35	12.8
					385	768	0.13	9944	66.49	21.9
PHOTON-80/20/16 [21]	64	40	40	80	708	4	0.18	865	2.82	1.59
					132	20	0.18	1168	12.15	2.70
PHOTON-128/16/16 [21]	112	64	64	128	996	4	0.18	1122	1.61	2.29
					156	24	0.18	1708	10.26	3.45
PHOTON-160/36/36 [21]	124	80	80	160	1332	4	0.18	1396	2.70	2.74
					180	28	0.18	2117	20.00	4.35
PHOTON-224/32/32 [21]	192	112	112	224	1716	4	0.18	1735	1.86	4.01
					204	32	0.18	2786	15.69	6.50
PHOTON-256/32/32 [21]	224	128	128	256	996	8	0.18	2177	3.21	4.55
					156	48	0.18	4362	20.51	8.38
U-QUARK [1]	120	64	64	128	544	1	0.18	1379	1.47	2.44
					68	8	0.18	2392	11.76	4.07
D-QUARK [1]	144	80	80	160	704	1	0.18	1702	2.27	3.10
					88	8	0.18	2819	18.18	4.76
S-QUARK [1]	192	112	112	224	1024	1	0.18	2296	3.13	4.35
					64	16	0.18	4640	50.00	8.39
DM-PRESENT-80 [11]	64	32	64	64	547	4	0.18	1600	14.63	1.83
					33	64	0.18	2213	242.42	6.28
DM-PRESENT-128 [11]	64	32	64	64	559	4	0.18	1886	22.90	2.94
					33	128	0.18	2530	387.88	7.49
H-PRESENT-128 [11]	128	64	64	128	559	8	0.18	2330	11.45	6.44
					32	128	0.18	4256	200.00	8.09
C-PRESENT-192 [11]	192	96	192	192	3338	12	0.18	4600	1.90	-
					108	192	0.18	8048	59.26	9.31
KECCAK-f[400] [29]	160	80	160	160	1000	16	0.13	5090	14.40	11.50
					20	16	0.13	10560	720.00	78.10
KECCAK-f[200] [29]	128	64	128	128	900	8	0.13	2520	8.00	5.60
					18	8	0.13	4900	400.00	27.60
SHA-1 [31]	160	80	160	160	450	32	0.25	6812	113.78	11.00
SHA-256 [32]	256	128	256	256	490	32	0.25	8588	104.48	11.20
BLAKE [26]	256	128	256	256	816	32	0.18	13575	62.79	11.16
Grøstl [45]	256	128	256	256	196	64	0.18	14622	261.14	221.00

* The power figures rather serve an illustration purpose. A comparison between different technologies is difficult.

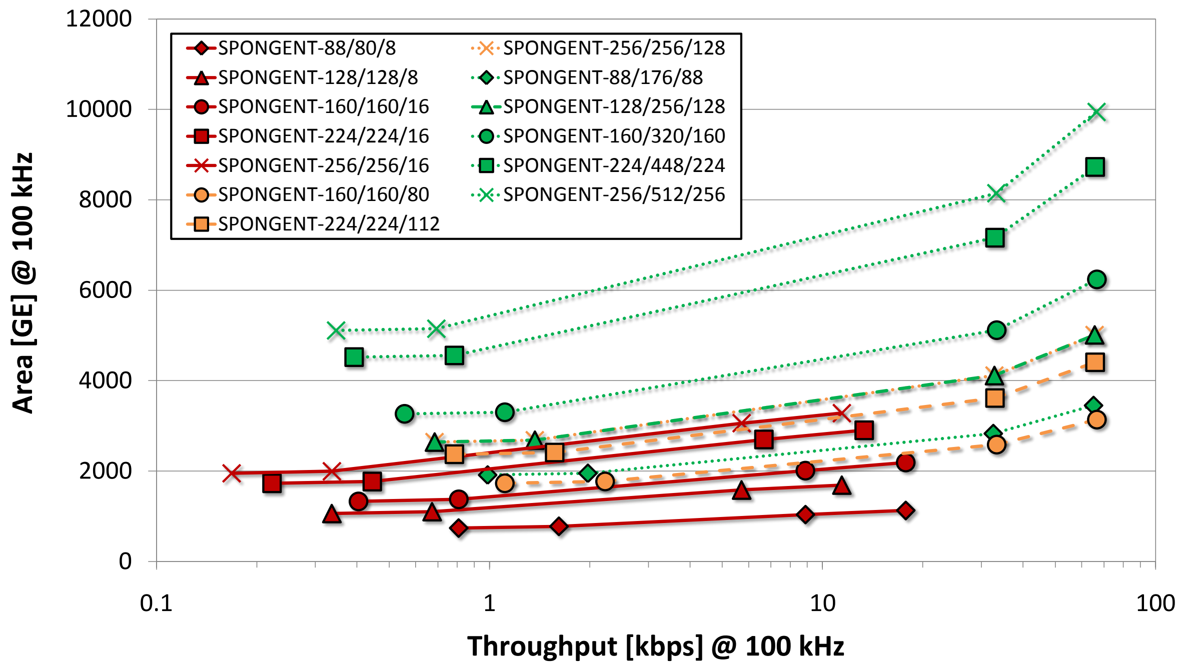


Fig. 6. Area versus throughput trade-off of the SPONGENT hash family

other cells and we only outline ones that are of particular interest to us. Several special cells acting as a combination of two or more basic gates (e.g. AO is a combination of AND and OR) are also used very often and are appropriate for reducing the physical size of the design. The size of these cells varies, mainly depending on the driving strength of the cell. The final design provided by the synthesis tool will therefore be driven by many internal factors, e.g. speed constraints, physical area constraints, fan-in, fan-out, length of the wires, and many others.

Moreover, we provide Table 9 where the same SPONGENT RTL designs were synthesized using four different libraries. Compared to our UMC130 library, the overhead of UMC180 and NANGATE45 libraries ranges up to 13 % and 20 %, respectively, while the NXP90 library results in smaller area up to 32 %, which represents a significant margin (the size is compared using gate equivalences).

The main cause of the above described variance is a different cells' size, which is directly related to the library type. A single scan flip-flop consumes at least 6.25 GE and 6.67 GE in UMC130 and UMC180, respectively. The NXP90 library has significantly smaller flip-flops which are the main area consumers in the case of SPONGENT family. NANGATE45 (with a scan flip-flop of 7.67 GE), on the other hand, is an open core library and seems to be a good candidate for accurate comparison between different lightweight designs.

5 Conclusion

In this work, we have explored the design space of lightweight cryptographic hashing by proposing the family of new hash functions SPONGENT tailored for resource-constrained applications. We

Table 8. Area requirements of selected standard cells in our UMC 130 nm library.

Standard cell	Number of inputs	Area [μm^2]	Area [GE]
D Flip Flop	1	20 – 40	5 – 10
Scan Flip Flop	1	25 – 47	6.25 – 11.75
NOT	1	3 – 28	0.75 – 7
NAND	2	4 – 23	1 – 5.75
	3	6 – 14	1.5 – 3.5
	4	12 – 18	3 – 4.5
NOR	2	4 – 40	1 – 10
	3	6 – 13	1.5 – 3.25
	4	11 – 19	2.75 – 4.75
AND	2	5 – 19	1.25 – 4.75
	3	7 – 16	1.75 – 4
	4	10 – 33	2.5 – 8.25
OR	2	5 – 25	1.25 – 6.25
	3	7 – 26	1.75 – 6.5

Standard cell	Number of inputs	Area [μm^2]	Area [GE]
XOR	2	11 – 16	2.75 – 4
	3	22 – 26	5.5 – 6.5
	4	30 – 31	7.5 – 7.75
AO, AN	6	6 – 17	1.5 – 4.25
	4	6 – 21	1.5 – 5.25
	6	10 – 25	2.5 – 6.25
	8	15 – 18	3.75 – 4.5
OA, NA	6	5 – 21	1.25 – 5.25
	4	6 – 21	1.5 – 5.25
	6	9 – 18	2.25 – 4.5
	8	15 – 18	3.75 – 4.5
MUX	2	9 – 28	2.25 – 7
	3	16 – 27	4 – 6.75
	4	25 – 35	6.25 – 8.75

AO = AND and OR, AN = AND and NOR,
OA = OR and AND, NA = NOR and AND.

consider 5 hash sizes for SPONGENT – ranging from the ones offering mainly preimage resistance only to those complying to (a subset of) SHA-2 and SHA-3 parameters. For each parameter set, we instantiate SPONGENT using up to three competing security paradigms (all of them offering full collision security): reduced second-preimage security, reduced preimage and second-preimage security, as well as full preimage and second-preimage security. Each parametrization accounts for its unique implementation properties in terms of ASIC hardware footprint, performance and time-area product, which are analyzed in the article. We also perform security analysis in terms of differential properties, linear distinguishers, and rebound attacks.

6 Acknowledgment

Andrey Bogdanov is a postdoctoral fellow of the Fund for Scientific Research - Flanders (FWO). This work is supported in part by the IAP Programme P6/26 BCRYPT of the Belgian State, by the European Commission under contract numbers ICT-2007-216676 ECRYPT NoE phase II and ICT-2007-238811 UNIQUE, and by the Research Council K.U.Leuven: GOA 11/007 TENSE.

Table 9. Area of the SPONGENT family compared using four different standard cell libraries

	Datapath (bit)	Area (GE) UMC 130 nm	UMC 180 nm	NANGATE 45 nm	NXP 90 nm
SPONGENT-88/80/8	4	738	759	868	521
	88	1127	1232	1236	883
SPONGENT-88/176/88	4	1912	1965	2264	1308
	264	3450	3847	3633	2553
SPONGENT-128/128/8	4	1060	1103	1256	737
	136	1687	1855	1831	1279
SPONGENT-128/256/128	4	2641	2724	3182	1813
	384	5011	5581	5715	4167
SPONGENT-160/160/16	4	1329	1367	1571	918
	176	2190	2241	2406	1752
SPONGENT-160/160/80	4	1730	1769	2066	1192
	240	3139	3434	3612	2650
SPONGENT-160/320/160	4	3264	3340	3930	2232
	480	6237	6949	7163	5262
SPONGENT-224/224/16	4	1728	1768	2071	1192
	240	2903	3203	3220	2334
SPONGENT-224/224/112	4	2371	2422	2826	1621
	336	4406	4900	4611	3197
SPONGENT-224/448/224	4	4519	4625	5430	3069
	672	8726	9696	9751	6932
SPONGENT-256/256/16	4	1950	2012	2323	1340
	272	3281	3721	3639	2612
SPONGENT-256/256/128	4	2641	2724	3182	1813
	384	5011	5581	5713	4213
SPONGENT-256/512/256	4	5110	5232	6163	3471
	768	9944	11054	10777	7426

References

1. Aumasson, J.P., Henzen, L., Meier, W., Naya-Plasencia, M.: Quark: A lightweight hash. In: Mangard, S., Standaert, F.X. (eds.) CHES. Lecture Notes in Computer Science, vol. 6225, pp. 1–15. Springer (2010)
2. Avoine, G., Oechslin, P.: A Scalable and Provably Secure Hash-Based RFID Protocol. In: PerCom Workshops. pp. 110–114. IEEE Computer Society (2005)
3. Babbage, S., Dodd, M.: The MICKEY Stream Ciphers. In: Robshaw, M.J.B., Billet, O. (eds.) The eSTREAM Finalists, Lecture Notes in Computer Science, vol. 4986, pp. 191–209. Springer (2008)
4. Badel, S., Dagtekin, N., Nakahara, J., Ouafi, K., Reffé, N., Sepehrdad, P., Susil, P., Vaudenay, S.: ARMADILLO: A Multi-purpose Cryptographic Primitive Dedicated to Hardware. In: Mangard, S., Standaert, F.X. (eds.) CHES. Lecture Notes in Computer Science, vol. 6225, pp. 398–412. Springer (2010)
5. Barreto, P.S.L.M., Rijmen, V.: The Whirlpool hashing function. In: Proceedings of the 1st NESSIE Workshop. p. 15. Leuven,B (2000)
6. Benadjila, R., Billet, O., Gilbert, H., Macario-Rat, G., Peyrin, T., Robshaw, M., Seurin, Y.: SHA-3 Proposal: ECHO. Submission to NIST (updated) (2009), `http://crypto.rd.francetelecom.com/echo/doc/echo_description_1-5.pdf`
7. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: On the Indifferentiability of the Sponge Construction. In: Smart, N.P. (ed.) EUROCRYPT. Lecture Notes in Computer Science, vol. 4965, pp. 181–197. Springer (2008)
8. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Sponge-Based Pseudo-Random Number Generators. In: Mangard, S., Standaert, F.X. (eds.) CHES. Lecture Notes in Computer Science, vol. 6225, pp. 33–47. Springer (2010)
9. Bogdanov, A., Knezevic, M., Leander, G., Toz, D., Varici, K., Verbauwhede, I.: spongent: A lightweight hash function. In: Preneel, B., Takagi, T. (eds.) CHES. Lecture Notes in Computer Science, vol. 6917, pp. 312–325. Springer (2011)
10. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: An Ultra-Lightweight Block Cipher. In: Paillier, P., Verbauwhede, I. (eds.) CHES. Lecture Notes in Computer Science, vol. 4727, pp. 450–466. Springer (2007)

11. Bogdanov, A., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y.: Hash Functions and RFID Tags: Mind the Gap. In: Oswald, E., Rohatgi, P. (eds.) CHES. Lecture Notes in Computer Science, vol. 5154, pp. 283–299. Springer (2008)
12. Buchmann, J., García, L.C.C., Dahmen, E., Döring, M., Klintsevich, E.: CMSS - An Improved Merkle Signature Scheme. In: Barua, R., Lange, T. (eds.) INDOCRYPT. Lecture Notes in Computer Science, vol. 4329, pp. 349–363. Springer (2006)
13. Cho, J.Y.: Linear Cryptanalysis of Reduced-Round PRESENT. In: Pieprzyk, J. (ed.) CT-RSA. Lecture Notes in Computer Science, vol. 5985, pp. 302–317. Springer (2010)
14. Collard, B., Standaert, F.X.: A Statistical Saturation Attack against the Block Cipher PRESENT. In: Fischlin, M. (ed.) CT-RSA. Lecture Notes in Computer Science, vol. 5473, pp. 195–210. Springer (2009)
15. Daemen, J., Peeters, M., Assche, G.V.: Sponge Functions. Ecrypt Hash Workshop 2007 (2007), http://www.csrc.nist.gov/pki/HashWorkshop/PublicComments/2007May.html
16. De Cannière, C.: Trivium: A Stream Cipher Construction Inspired by Block Cipher Design Principles. In: Katsikas, S.K., Lopez, J., Backes, M., Gritzalis, S., Preneel, B. (eds.) ISC. Lecture Notes in Computer Science, vol. 4176, pp. 171–186. Springer (2006)
17. De Cannière, C., Dunkelman, O., Knežević, M.: KATAN and KTANTAN - A Family of Small and Efficient Hardware-Oriented Block Ciphers. In: Clavier, C., Gaj, K. (eds.) CHES. Lecture Notes in Computer Science, vol. 5747, pp. 272–288. Springer (2009)
18. De Cannière, C., Preneel, B.: Trivium. In: Robshaw, M.J.B., Billet, O. (eds.) The eSTREAM Finalists, Lecture Notes in Computer Science, vol. 4986, pp. 244–266. Springer (2008)
19. Duc, A., Guo, J., Peyrin, T., Wei, L.: Unaligned Rebound Attack - Application to Keccak. Cryptology ePrint Archive, Report 2011/420 (2011), http://eprint.iacr.org/2011/420
20. Gauravaram, P., Knudsen, L.R., Matusiewicz, K., Mendel, F., Rechberger, C., Schlffer, M., Thomsen, S.S.: Grøstl – a SHA-3 candidate. Submission to NIST (Round 3) (2011), http://www.groestl.info/Groestl.pdf
21. Guo, J., Peyrin, T., Poschmann, A.: The photon family of lightweight hash functions. In: Rogaway, P. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 6841, pp. 222–239. Springer (2011)
22. Hein, D.M., Wolkerstorfer, J., Felber, N.: ECC Is Ready for RFID - A Proof in Silicon. In: Avanzi, R.M., Keliher, L., Sica, F. (eds.) Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 5381, pp. 401–413. Springer (2008)
23. Hell, M., Johansson, T., Maximov, A., Meier, W.: The Grain Family of Stream Ciphers. In: Robshaw, M.J.B., Billet, O. (eds.) The eSTREAM Finalists, Lecture Notes in Computer Science, vol. 4986, pp. 179–190. Springer (2008)
24. Hell, M., Johansson, T., Meier, W.: Grain: a stream cipher for constrained environments. IJWMC 2(1), 86–93 (2007)
25. Henzen, L., Aumasson, J.P., Meier, W., Phan, R.C.W.: VLSI Characterization of the Cryptographic Hash Function BLAKE. http://131002.net/data/papers/HAMP10.pdf (2010)
26. Henzen, L., Aumasson, J.P., Meier, W., Phan., R.C.W.: VLSI Characterization of the Cryptographic Hash Function BLAKE (2010), available at http://131002.net/data/papers/HAMP10.pdf
27. Hong, D., Sung, J., Hong, S., Lim, J., Lee, S., Koo, B., Lee, C., Chang, D., Lee, J., Jeong, K., Kim, H., Kim, J., Chee, S.: HIGHT: A New Block Cipher Suitable for Low-Resource Device. In: Goubin, L., Matsui, M. (eds.) CHES. Lecture Notes in Computer Science, vol. 4249, pp. 46–59. Springer (2006)
28. Indesteege, S.: The LANE hash function. Submission to NIST (2008), http://www.cosic.esat.kuleuven.be/publications/article-1181.pdf
29. Kavun, E., Yalcin, T.: A Lightweight Implementation of Keccak Hash Function for Radio-Frequency Identification Applications. In: Ors Yalcin, S. (ed.) Radio Frequency Identification: Security and Privacy Issues, Lecture Notes in Computer Science, vol. 6370, pp. 258–269. Springer Berlin / Heidelberg (2010)
30. Khovratovich, D., Naya-Plasencia, M., Röck, A., Schläffer, M.: Cryptanalysis of *Luffa* v2 Components. In: Biryukov, A., Gong, G., Stinson, D.R. (eds.) Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 6544, pp. 388–409. Springer (2010)
31. Kim, M., Ryou, J.: Power Efficient Hardware Architecture of SHA-1 Algorithm for Trusted Mobile Computing. In: Proceedings of the 9th international conference on Information and communications security. pp. 375–385. ICICS'07, Springer (2007)
32. Kim, M., Ryou, J., Jun, S.: Efficient Hardware Architecture of SHA-256 Algorithm for Trusted Mobile Computing. In: Yung, M., Liu, P., Lin, D. (eds.) Inscrypt. Lecture Notes in Computer Science, vol. 5487, pp. 240–252. Springer (2008)
33. Leander, G.: On Linear Hulls, Statistical Saturation Attacks, PRESENT and a Cryptanalysis of PUFFIN. to appear (2011)

34. Leander, G., Abdelraheem, M.A., AlKhzaimi, H., Zenner, E.: A Cryptanalysis of PRINTcipher: The Invariant Subspace Attack. In: Rogaway, P. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 6841, pp. 206–221. Springer (2011)
35. Leander, G., Paar, C., Poschmann, A., Schramm, K.: New Lightweight DES Variants. In: Biryukov, A. (ed.) FSE. Lecture Notes in Computer Science, vol. 4593, pp. 196–210. Springer (2007)
36. Lim, C.H., Korkishko, T.: mCrypton - A Lightweight Block Cipher for Security of Low-Cost RFID Tags and Sensors. In: Song, J., Kwon, T., Yung, M. (eds.) WISA. Lecture Notes in Computer Science, vol. 3786, pp. 243–258. Springer (2005)
37. Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl. In: Dunkelman, O. (ed.) FSE. Lecture Notes in Computer Science, vol. 5665, pp. 260–276. Springer (2009)
38. Merkle, R.: Secrecy, authentication and public key systems / A certified digital signature. Ph.D. thesis, Dept. of Electrical Engineering, Stanford University (1979)
39. NANGATE: The NanGate 45nm Open Cell Library, available at `http://www.nangate.com`
40. Osaka, K., Takagi, T., Yamazaki, K., Takahashi, O.: An Efficient and Secure RFID Security Method with Ownership Transfer. In: Wang, Y., ming Cheung, Y., Liu, H. (eds.) CIS. Lecture Notes in Computer Science, vol. 4456, pp. 778–787. Springer (2006)
41. Rohde, S., Eisenbarth, T., Dahmen, E., Buchmann, J., Paar, C.: Fast Hash-Based Signatures on Constrained Devices. In: Grimaud, G., Standaert, F.X. (eds.) CARDIS. Lecture Notes in Computer Science, vol. 5189, pp. 104–117. Springer (2008)
42. Shoufan, A.: An FPGA Accelerator for Hash Tree Generation in the Merkle Signature Scheme. In: Sirisuk, P., Morgan, F., El-Ghazawi, T.A., Amano, H. (eds.) ARC. Lecture Notes in Computer Science, vol. 5992, pp. 145–156. Springer (2010)
43. Standaert, F.X., Piret, G., Gershenfeld, N., Quisquater, J.J.: SEA: A Scalable Encryption Algorithm for Small Embedded Applications. Presented at the Workshop on RFID and Light-Weight Crypto in Graz, Austria (2005)
44. Tillich, S., Feldhofer, M., Issovits, W., Kern, T., Kureck, H., Muehlberghuber, M., Neubauer, G., Reiter, A., Koefler, A., Mayrhofer, M.: Compact Hardware Implementations of the SHA-3 Candidates ARIRANG, BLAKE, Grøstl, and Skein. Cryptology ePrint Archive, Report 2009/349 (2009)
45. Tillich, S., Feldhofer, M., Issovits, W., Kern, T., Kureck, H., Mühlberghuber, M., Neubauer, G., Reiter, A., Köfler, A., Mayrhofer, M.: Compact Hardware Implementations of the SHA-3 Candidates ARIRANG, BLAKE, Grøstl, and Skein. Cryptology ePrint Archive, Report 2009/349 (2009), available at `http://eprint.iacr.org/2009/349`
46. Tsudik, G.: YA-TRAP: Yet Another Trivial RFID Authentication Protocol. In: PerCom Workshops. pp. 640–643. IEEE Computer Society (2006)
47. Yang, B., Wu, K., Karri, R.: Scan Based Side Channel Attack on Dedicated Hardware Implementations of Data Encryption Standard. International Test Conference pp. 339–344 (2004)

A new generic protocol for authentication and key agreement in lightweight systems

Naïm Qachri[1], Frédéric Lafitte[2], and Olivier Markowitch[1]

[1] Département d'Informatique, Université Libre de Bruxelles,
CP212, boulevard du Triomphe, 1050 Brussels, Belgium
nqachri@ulb.ac.be,olivier.markowitch@ulb.ac.be
[2] Royal Military Academy, Department of Mathematics
Renaissancelaan 30, 1000 Brussels, Belgium
frederic.lafitte@rma.ac.be

Abstract. In this paper, we propose a new generic authenticated key agreement protocol where the master secret is automatically renewed based on a sequence of hash values, thus providing the system with an extended cryptoperiod. The focus of this work is to formally assess the security offered by the protocol's key renewing in the case of a long term use of the system. The formal analysis is carried using the automated tools ProVerif and AVISPA. The protocol is designed to be implemented on devices with limited computing and storage resources.

Key words: Mutual authentication, Key agreement protocol, Wireless communication security, Cryptoperiod

1 Introduction

Since a decade, we have seen many developments to strengthen the security of wireless embedded communication systems: these improvements intended to correct problems related, for example, to defective cryptographic primitives or protocols [17, 7, 10] (e.g. the 4-way handshake protocol defined for the Wimedia MAC layer standard [16] used to authenticate and generate session keys). Nowadays, the last versions of the Wifi and Wimax standards include the use of EAP [1] declined in different versions (LEAP – EAP using a Radius Server –, EAP-TTLS, etc.). In practice, EAP is not well suited for constrained environments such as handheld devices, short-range communication systems or even domestic wireless LAN devices. The reason being that many versions of EAP use certificates, public key encryption or exhaustive exchanges of information, that are not always appropriate for lightweight wireless devices. For example, certificate revocation is not viable in point to point communications.

Moreover, wireless communication protocols are dedicated to the specific technology considered. This lack of genericity makes the security evaluation of such protocols more cumbersome and also less re-usable. Furthermore, those protocols do not address the issue of the master secret cryptoperiod. This weakness has already been used in order to mount practical attacks in [17, 10]: instead of

renewing the key used by RC4, the WEP protocol uses a different IV (Inital-ization Vector). However, the number of IVs is not large enough to prevent key re-use, which is a serious weakness for stream ciphers.

Furthermore, the existing systems proposed in the literature do not meet all of the needed requirements; indeed, the existing solutions either do not combine authentication and key generation [14, 5], use public-key cryptography [20, 21, 12, 9], use a trusted entity [14, 15], or combine artificially an authentication scheme [13] with a key transport protocol [14] (involving more transfers without gaining more security). Similar authentication mechanisms used for [13] are used in [3, 4]. The ISO/IEC 9798-2 and ISO/IEC 11770-2 protocols (ISO standards that define some key agreement protocols) differ from our protocol since they do not authenticate the exchanged messages and they consider only single execution of the protocols. Therefore, because of the lack of messages authentications, the information exchanged during these protocols (that are used to produce the session keys) are not protected against modifications, and may be exposed to denial of services attacks. Moreover, most of the protocols does not include a session key confirmation procedure as well as a synchronization mechanism (i.e. in case of repetitive establishments of sessions).

Contribution. According to the paragraphs above, we are after a key agreement protocol that relies only on symmetric key cryptography and satisfies all the requirements for contributive authenticated key exchange in wireless communication systems. In particular, we focus on the protocol lifecycle by introducing an automated key renewing mechanism. The security of our proposal is analyzed in a Dolev-Yao model [6] where the adversary has corruption capabilities. We use the tools AVISPA and ProVerif in order to automate the analysis. Our protocol is generic since we do not instantiate the underlying cryptographic primitives. It is considered lightweight since no asymmetric algorithms are used. Therefore, in this symmetric key setting, our protocol best fits point to point communications, or centralized networks, where a device interacts with a limitted number of participants.

Outline. The paper is divided in six sections. The next section exposes the assumptions, definitions, and notations that will be used along the paper. The third section presents a general description of the protocol and a formal definition of the protocol and the subprotocols is given. The fourth section presents a formal analysis of the security of the protocol. The fifth section describes an interesting security finding. In the sixth section, we summarize some future works for a larger exploitation of our protocol.

2 Preliminaries

In this section we define the security requirements for authenticated key exchange protocols as well as the adversarial capabilities. We also introduce general assumptions and notations.

2.1 Requirements

Common security requirements for key agreement include:

Contributiveness: the key exchange is said to be contributive if Alice and Bob contributed equally to the computation of the new session keys (and the renewing of the long-term key).

Backward security: future sessions remain secure even if secrets used in past sessions are corrupted. Different types of secrets can be leaked in a session. The type of secret that is leaked is used to caracterise the type of corruption (see types of corruption below).

Forward security: past sessions remain secure even if future sessions are corrupted. Again, different kinds of corruptions can be considered.

Mutual authentication: the mutual authentication is a mechanism that allows to authenticate two devices to each other. In our case, this property is obtained through a *challenge-response* mechanism.

Key confirmation: the purpose of key confirmation is to make sure that both Alice and Bob derived the same session keys. Many existing protocols do not satisfy this property. Obviously, this verification procedure must not reveal the underlying generated key and can be achived through a *challenge-response* mechanism.

In addition to the rather common properties defined above, our protocol has the following property.

Key renewing: another desirable property (see intro) consists in the automated renewing of the master secret. We consider this key renewing procedure secure as long as the new key remains secret.

The requirements defined above only make sense when considered together with a specific adversary. In our case, the (active) adversary has Dolev-Yao capabilities [6] over all communication channels, i.e., it is capable to read, delay, delete, insert messages from/to the channels used in the execution of the protocol.

We also assume that the adversary is able to corrupt participants, i.e., to learn their secret values. Secrets involved in key exchange protocols are often categorized as short term (ephemeral) secrets and long term (master) secrets. In our case, the ephemeral secrets correspond to the nonces and the master secret consists of the chain of hash values. This leads to the consideration of three kinds of corruptions.

Types of corruptions: Corruptions that leak an ephemeral secret are referred to as type I corruptions whereas those leaking the master secret are referred to as type II corruptions. Type III corruptions correspond to the case where both ephemeral and master secrets are leaked.

In section 4, we assess to what extent each type of corruption can be used to compromise the requirements defined above, in particular, forward and backward secrecy. This analysis is conducted in a symbolic framework.

2.2 Assumptions

We suppose that Alice and Bob (wireless devices that know each other) share a symmetric key that has been secretly exchanged during an association step that happens before any exchange that would occur between the two devices. It is assumed that this association step is secure and therefore does not leak any information about the shared secret. Furthermore, the association step is not considered in the protocol because this step is strongly dependent of the technology considered and will remove a part of the genericity of the protocol. Moreover, it is assumed that the secret, shared by Alice and Bob, is not already shared by them with any other devices.

It is also assumed that the devices are tamper resistant (i.e. an attacker cannot physically read or modify the secrets stored in the devices).

2.3 Notations

The notations used in our protocol are the following:

- h_i denotes that the hash function H is applied successively, i times, according to the following construction:

$$h_1 = H(s)$$
$$h_i = H(h_{i-1} \parallel s) \qquad \forall i > 1$$

- $E_k(m)$ denotes a symmetric bloc encryption of the message m with the secret key k;
- $MAC_k(m)$ denotes the result of a keyed hash function applied on a message m;
- s denotes the secret shared between Alice and Bob during the association step;
- r_A and r_B denotes the random nonces chosen and sent respectively by Alice and Bob during a session of the key agreement protocol;
- $LSB_i(m)$ is a function that truncates m to its i *least significant bits.*

3 The protocol

3.1 General description of the protocol

Based on the initial secret (exchanged at the association step), keys for authentication and encryption are generated and shared between Alice and Bob.

The protocol is designed in order to avoid that an attacker, who discovers the secrets of a session of the protocol, can deduce the secrets that will be computed during the following sessions. The secret values of the different sessions are computed on the basis of a chain of hash values. During an initialization step (that takes place after the association step), Alice and Bob realize the computation of n consecutive hashing on the initial shared secret. Those hashed

values will be used to authenticate Alice and Bob and to generate the session keys as described hereafter. After the initialization step, when a session must be set, Alice invokes a 3-step main subprotocol that ensures mutual authentication by the means of challenge-response techniques, session keys generated between Alice and Bob and desynchronization resistance.

Since the produced secret hashed values are in a limited number, when only three hash values remain, a new secret is computed using those values and this new secret key is used to create a new chain of hash values that will be used for the next sessions of the protocol. The chain of hash values implies a notion of lifecycle of secret keys.

The key renewing procedure and the generation of keys during the key agreement are based on a secure key generation mechanism similar to the HMAC-Key Derivation Function (`HDKF` [11]).

3.2 The protocol life cycle

The protocol begins with the two following steps:

- the association step (where a secret s is exchanged);
- the initialization step (where some or all of the n hashed values, from the secret s, are computed in order to speed up further hash computations in the protocol and i is initialized to 1 by Alice and Bob).

$$h_1, \dots, h_n \qquad \text{and} \qquad CC \leftarrow 1$$

Then, the life cycle of the protocol is described as follows :

- $\lfloor \frac{n}{3} - 1 \rfloor$ successive executions of the main protocol and/or *resynchronization protocol* (if needed)
- renewing protocol (where the initialization step is made again after the renewing)
- successive executions of the main protocol (a new cycle is launched)...

The *Cycle Counter* (CC) is a variable that counts the number of chains of n hashed values completely used in the life cycle of the protocol since the initialization step (where CC is set to 1). Within a cycle, a session of the main protocol is characterized by a number i. The concatenation $CC \parallel i$ is a unique identifier of a session of the protocol between Alice and Bob.

The initialization step, made after the association step, consists in computing the chain of the n hashed values that will be used during the executions of the main subprotocol during a lifecycle. This initialization step is made again after a renewing of the secret.

3.3 The main subprotocol

When Alice and Bob have to initiate a new session i, being in the cycle CC, they execute the following main protocol:

$$\textbf{1.}\quad \textbf{Alice} \rightarrow \textbf{Bob} : ID_{Alice}, i, E_{h_{n-((i-1)\times 3)}}(m_1, H(m_1))$$
$$\text{where} \quad m_1 = (1, CC, i, r_A, ID_{Alice})$$

The message contains the first challenge under the form of a message to decrypt and verify. If the two devices share the same secret, then Bob can decrypt and verify it. The nonce, r_A, sent by Alice, has to be well chosen and contributes to the keys generation in a fair way in regards to Bob. The number i of the session is sent unencrypted to synchronize the two devices on the keys to use (see the resynchronization subprotocol). The hashing, at the end of the encrypted message, is computed to ensure that the message cannot be easily manipulated by the attacker regardless of the encryption algorithm.

Once Bob has decrypted the message, he chooses a second random nonce, r_B, and generates three keys by computing the following MAC:

$$(\mathbf{k_{SE}} \parallel \mathbf{k_{SA}} \parallel \mathbf{k_{conf}}) = LSB_q(MAC_{h_{n-((i-1)\times 3)-2}}(CC \parallel i \parallel r_A \parallel r_B \parallel h_{n-((i-1)\times 3)-1}))$$

where $\parallel$ is the concatenation operator. r_A and r_B are the contributions of respectively Alice and Bob in the computation of these three keys. q denotes the sum of the sizes of the three keys generated during a session of the protocol. $CC \parallel i$ is used to avoid replay attacks. $\mathbf{k_{SE}}$ is the key generated to encrypt the communication of the session that will take place between Alice and Bob and $\mathbf{k_{SA}}$ is the key generated to authenticate the packets transmitted during this communication.

The MAC algorithm has to be well dimensioned to generate enough bits for the three keys. On the basis of $\mathbf{k_{conf}}$, Bob creates a new challenge and sends it to Alice.

$$\textbf{2.}\quad \textbf{Bob} \rightarrow \textbf{Alice} : ID_{Bob}, i, E_{h_{n-((i-1)\times 3)}}(m_2, MAC_{k_{conf}}(m_2))$$
$$\text{where} \quad m_2 = (2, CC, i, r_B, r_A, ID_{Bob})$$

The challenge has the purpose to ensure that Alice can derive the good key and decrypt the message of Bob. From these keys, Alice can verify that she has derived the same keys than Bob if she is able to verify the MAC on the message. Alice can also authentify Bob, since only Bob knows the secret hashed value used to encrypt the message and authenticate. Furthermore, Bob sends the nonce r_A to prove that he has made the correct decryption of the first message.

$$\textbf{3.}\quad \textbf{Alice} \rightarrow \textbf{Bob} : ID_{Alice}, i, MAC_{k_{conf}}(3, ID_{Alice}, CC, i, r_A, r_B)$$

In this third message, Alice answers that she has well derived the keys and that the authentication of Bob succeeds, she provides also r_B to prove that she has made the correct decryption of the second message. At the end of the main protocol, Alice and Bob increment i.

3.4 The renewing subprotocol

Within a cycle, on the basis of n hash values, we can realize $\lfloor \frac{n}{3} - 1 \rfloor$ sessions of the main protocol. We use the last session of the protocol to generate a new secret value that will overwrite the previous shared secret between Alice and Bob (initially s).

The renewing is made when only three hash values (h_3, h_2 and h_1 remain). Alice and Bob run again the main subprotocol (see Figure 1) during which the new secret is computed from $h_1 = H(s)$ and s. This execution allows the exchange of r_A and r_B.

$$\mathbf{s_{new}} = MAC_{s_{old}}(CC \parallel r_A \parallel r_B \parallel H(s_{old}))$$

The shared secret hash values and the CC are computed for the future sessions of the next cycle of the main protocol:

$$h_1 = H(s_{new}), \ldots, h_j = H(h_{j-1}||s_{new}) \;\; \forall j \in \{2, \ldots, n\}$$
$$\text{and} \quad CC \leftarrow CC + 1; \quad i \leftarrow 1$$

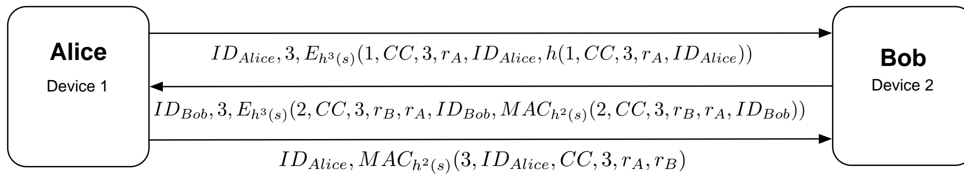


Fig. 1. The renewing protocol

3.5 The resynchronization protocol

If the two devices are desynchronized (i.e. if Alice and Bob consider a different value of i), they reveal their session values i and i'. In that case, the current session of the main protocol is aborted and a new session protocol is launched with a session value $max(i, i') + 1$.

We make the assumption that, in case of desynchronization, the devices cannot have different CC's, because it would mean that one of the two devices has done the renewing protocol without having incremented the variable CC (unless one of the devices were cloned).

4 Security Analysis

In this section, we use the automated protocol verification tools AVISPA and ProVerif in order to show that our protocol meets the requirements defined in section 2.1 when attacked by an active Dolev-Yao adversary with corruption capabilities.

The symbolic Dolev-Yao model abstracts away cryptographic operations on bits and considers only *symbols* in order to represent keys, nonces, messages, etc. Therefore, in this model, cryptographic primitives are considered ideal, since the adversary is able to recover the secret symbol (i.e. 100% of the secret bits) or not (i.e. 0% of the secret bits). In our case, since we do not instanciate the cryptographic algorithms used by our protocol, this abstraction is necessary for the analysis of the protocol.

AVISPA is used to establish the mutual authentication property whereas ProVerif is used to assess the impact of corruptions. In both cases, the three keys derived at the end of one session are considered as one unique symbol (being the concatenation of the three keys).

4.1 Analysis using AVISPA

AVISPA [19, 18] is an automated protocol verifier based on temporal logic. Its purpose is to check the security of protocols through a specification language named HLPSL. The language allows for the description of the protocol through roles and sessions of execution where the channel of communication meets the Dolev-Yao model [6]. We have chosen AVISPA, because it offers the opportunity to analyze our protocol in four *backends* (verification engines) and then to cover a large spectrum of possible attacks.

Since AVISPA does not define boolean or MAC functions, we have rather modeled the HMAC function (through its formal definition):

$$HMAC_k(\mathbf{m}) = H((k \oplus \mathbf{opad})||H((k \oplus \mathbf{ipad})||\mathbf{m}))$$

where $\oplus$ is the *exclusive-or* and $||$ the concatenation operation. The **opad** and **ipad** values are constant values, respectively composed of the repeated $0x5C$ and $0x36$ values, as long as the key k.

Because of the modeling of the HMAC with the use of *exclusive-or* operations, two of the four backends (i.e. SATMC and TA4SP) became not conclusive on the security of the protocol, because they cannot manage arithmetic or boolean operations used within the protocol specification. We have analyzed the protocol on parallel sessions where we have explicitly computed a chain of hashed values.

We have taken a particular care in defining the security goals within AVISPA and the complete formalization of the process of the mutual authentication. In the specification, we have authenticated both parties with the challenges provided by the exchange of their nonces r_A and r_B, but also on $\mathbf{k_{conf}}$ that confirms the good derivation of the key and the fact that they know the secret hashed values without revealing them.

The sessions were specified to consider various possible attacks such as *Man in the middle attacks*, or *replay attacks* (through parallel executions of sessions).

The protocol was also checked in order to find some way to attack the secrecy of the three keys generated or the mutual authentication of both parties. The tests have given safe (i.e. no attacks were found) results on two of the four

backends (the two other were non conclusive due to the use of boolean arithmetic). This validate that our protocol is secure, that both parties are mutually authenticated and that the lifecycle is secure (see the discussion section below). The complete specification of the protocol is given in append.

4.2 Analysis using ProVerif

ProVerif [2] is an automated cryptographic protocol verifier, mainly used to assess secrecy and authentication properties. The protocol may be specified in different formalisms, in particular an extension of the (typed) applied pi calculus. Next, the adversary's goals and capabilities are specified (see appendix B). Internally, ProVerif translates the protocol and the adversary's capabilities into first order logic formulas (i.e. Horn clauses). Finally, a dedicated resolution algorithm is used to output one of the three following outcomes:

– a (sound) confirmation that the adversary is unable to reach his goal
– the execution trace of a successful attack
– the inability of the tool to conclude

Fortunately, in many practical cases ProVerif is able to conclude in a few seconds. The main advantage of ProVerif lies in its ability to analyze an unbounded number of sessions.

We model the execution of three consecutive sessions in order to show whether leaking a secret during session i helps the adversary in the recovery of secrets from session $i-1$ (forward secrecy) or session $i+1$ (backward secrecy). At the end of each session, both participants use the exchanged key to encrypt a secret that is specific to the session. That is, the key established in session one is used to encrypt the symbol secret1, the one established during session two encrypts the symbol secret2, and so on. The adversary is then queried on those three secrets since recovering one of them is equivalent to recovering the corresponding session key. As mentionned in section 2.1, the types of secrets leaked are either ephemeral secrets (i.e. nonces), master secrets (i.e. values from the hash chain) or both master and ephemeral secrets, thus caracterising three kinds of corruptions, referred to respectively as “type I”, “type II”, and “type III” corruptions. We need not assess the impact of leaking the secret s, since this leakage would allow the adversary to recover the entire chain of hash values. However, s is *only* used in the computation of the hash values. Therefore, the hash function's one-wayness ensures the secrecy of s.

Primitives. Symmetric key encryption is modeled by two symbols senc and sdec, whose meaning is captured by the following equation

$$\forall x, \forall k \ \mathsf{sdec}(k, \mathsf{senc}(k, x)) = x$$

The hash function H and the message authentication code function MAC are simply function symbols for which the absence of equation that decomposes $H(x)$ or $MAC(k, x)$ ensure the function's onewayness. That is, the only way for an adversary to build the term $MAC(k, x)$ $(H(x))$ is by knowing the symbols k and x (resp. x).

Other symbols. As usual in ProVerif, we use the symbol c for "broadcast channel" to model an insecure network. We also use symbols of constants to represent the cycle counter (CC), the session number (i) and the identities of the participants $\mathsf{IDA}, \mathsf{IDB}$.

Processes. We now describe the processes corresponding to the roles of Alice and Bob (i.e. initiator and responder respectively). The complete input file given to ProVerif can be found in appendix B.

- Process for participant A :
  1. let $\mathsf{h1} = \mathsf{h}(s)$ in let $\mathsf{h2} = \mathsf{h}(\mathsf{h1})$ in let $\mathsf{h3} = \mathsf{h}(\mathsf{h2})$ in
  2. new $\mathsf{r1}$
  3. let $m = \langle 1, \mathsf{CC}, \mathsf{i}, \mathsf{r1}, \mathsf{A} \rangle$
  4. out(c, $\langle \mathsf{A}, \mathsf{i}, \mathsf{senc}(\mathsf{h3}, \langle m, \mathsf{h}(m) \rangle) \rangle$)
  5. in(c, $\mathsf{msg}(\mathsf{B}, \mathsf{i}, x)$)
  6. let $\langle m', y \rangle = \mathsf{sdec}(\mathsf{h3}, x)$ in
  7. let $\langle 2, \mathsf{CC}, \mathsf{i}, \mathsf{r2}, \mathsf{r1}, \mathsf{B} \rangle = m'$ in let $\langle \mathbf{k_{se}}, \mathbf{k_{sa}}, \mathbf{k_{conf}} \rangle = \mathsf{mac}(\mathsf{h2}, \langle \mathsf{CC}, \mathsf{i}, \mathsf{r1}, \mathsf{r2}, \mathsf{h2} \rangle)$
  8. if $y = \mathsf{mac}(\mathbf{k_{conf}}, m')$ then
  9. out(c, $\langle \mathsf{A}, \mathsf{i}, \mathsf{mac}(\mathsf{h1}, \langle 3, \mathsf{A}, \mathsf{CC}, \mathsf{i}, \mathsf{r1}, \mathsf{r2} \rangle) \rangle$)
  10. out(c, $\langle \mathsf{senc}(x, \mathbf{k_{se}}), \mathsf{senc}(s, \mathbf{k_{se}}), \mathsf{senc}(x, \mathbf{k_{sa}}), \mathsf{senc}(s, \mathbf{k_{sa}}) \rangle$)
- Process for participant B :
  1. let $\mathsf{h1} = \mathsf{h}(s)$ in let $\mathsf{h2} = \mathsf{h}(\mathsf{h1})$ in let $\mathsf{h3} = \mathsf{h}(\mathsf{h2})$ in
  2. new $\mathsf{r1}$
  3. in(c, $\langle \mathsf{A}, \mathsf{i}, x \rangle$)
  4. let $\langle m, y \rangle = \mathsf{sdec}(\mathsf{h3}, x)$
  5. if $y = \mathsf{h}(m)$ then
  6. let $\langle 1, \mathsf{CC}, \mathsf{i}, \mathsf{r1}, \mathsf{A} \rangle = m$ in let $\langle \mathbf{k_{se}}, \mathbf{k_{sa}}, \mathbf{k_{conf}} \rangle = \mathsf{mac}(\mathsf{h2}, \langle \mathsf{CC}, \mathsf{i}, \mathsf{r1}, \mathsf{r2}, \mathsf{h2} \rangle)$
  7. let $m' = \langle 2, \mathsf{CC}, \mathsf{i}, \mathsf{r2}, \mathsf{r1}, \mathsf{B} \rangle$ in
  8. out(c, $\langle \mathsf{B}, \mathsf{i}, \mathsf{senc}(\mathsf{h3}, \langle m', \mathsf{mac}(\mathbf{k_{conf}}, m') \rangle) \rangle$)
  9. in(c,$\langle \mathsf{A}, \mathsf{i}, \mathsf{mac}(\mathsf{h1}, \langle 3, \mathsf{A}, \mathsf{CC}, \mathsf{i}, \mathsf{r1}, \mathsf{r2} \rangle) \rangle$)
  10. out(c, $\langle \mathsf{senc}(x, \mathbf{k_{se}}), \mathsf{senc}(s, \mathbf{k_{se}}), \mathsf{senc}(x, \mathbf{k_{sa}}), \mathsf{senc}(s, \mathbf{k_{sa}}) \rangle$)

Results. Type I corruptions do not allow for breaking the corresponding session. In the case of type II corruptions, as long as one of the three hash values involved in one session remains secret, the adversary cannot deduce *any* of the session keys. That is, the keys exchanged in the current, next, and previous sessions remain secret. In the case that all three hash values are leaked, then the current session is broken (i.e. the adversary learned $\mathsf{secret2}$) while the other sessions remain safe. This result holds independently of whether the nonces are leaked or not. Thus, the only way to compromise a session would be to reveal all three hash values. However, this would not affect other sessions since future (resp. previous) values of the hash chain are made inaccessible by the secrecy of s (resp. the one-wayness of H).

4.3 Discussion

We will present the analysis of the protocol in its entirety. The main purpose is to demonstrate that the security of our protocol, which is more realistic and complete, can be formally analyzed. Our protocol can be represented like in the Figure 2. The arrows with 1 stands for the initialization step. The arrows marked with 2 represents the complete set of sessions during a cycle. The arrows marked with 3 illustrate the invocation of the renewing subprotocol. Finally, the arrow marked with 4 symbolizes the break point between two lifecycle and can stand for the renewing during the key derivation and the one-way property of the derivation.

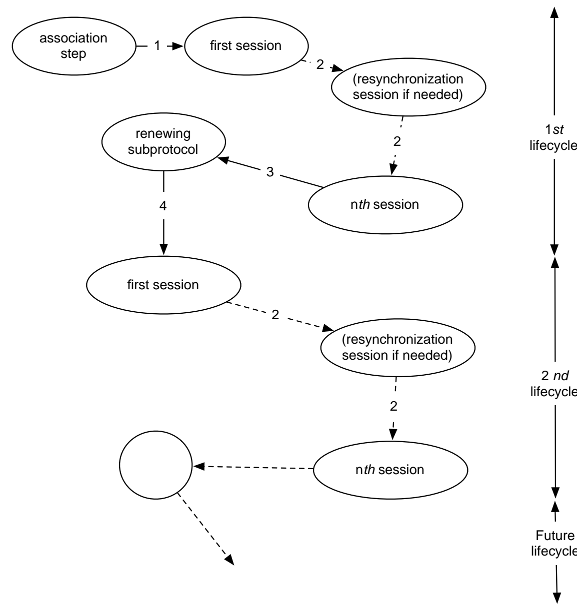


Fig. 2. The graphical representation of the protocol

We declare now that two lifecycle are independent thanks to its renewing subprotocol, because the one-way property and the use of random nonces produce a new secret that is strongly independent from the previous. This independence implies that our analysis can be restricted to one lifecycle.

If we examine a lifecycle, Alice and Bob will process $\lfloor \frac{n}{3} - 1 \rfloor$ sessions. The rest of the analyses are conducted with the use of formal methods, because they can scrutinize the security of single or multiple concurrent sessions of the main subprotocol.

Furthermore if the sessions are indeed independent, the security analysis can be reduced to single session execution. A session is independent from the others if it does not deliver information about past and future sessions. Proving the independence is done with the use of automated tools.

During a lifecycle, a session i has the most informations about the $(i-1)th$ and $(i+1)^{th}$ sessions. Proving this independence consists in proving that both consecutive sessions are independent. By induction, if the session $(i-1)th$ and i are independent and the session i and $(i+1)^{th}$ are independent, then by transitivity the sessions $(i+1)^{th}$ and $(i-1)^{th}$ are independent.

We have devised and made the analysis of parallel executions of three consecutive sessions with the tools AVISPA and ProVerif in sections 4.1 and 4.2 respectively. The analysis has not found any attack such as impersonation, replay, or man-in-the-middle attacks. We can consider then that the security of each session is equivalent to the security of one session. Furthermore, this independence proves that the protocol is backward and forward secure.

The construction of the chain (of hashed values) is an important aspect in the security of our subprotocols. In [8], the authors describe efficient and secure methods to compute hash chains for authentication. These methods are stronger than ours, but they make the assumption that a hashed value of the chain may be revealed after its use. In that case, the collection of the revealed values provide the ability to forge false chains of hashed values. We do not make such an assumption, because the elements of our hash chain are thrown after their use and therefore never revealed. Nonetheless, it remains possible to use the construction developed in [8] for our protocol.

5 Other security finding

If the underlying technology of implementation is based on very short range communication technologies (such as RFID or NFC technologies), it would be possible to detect cloned device thanks to the value of CC.

The cloned device will have access during a maximum of n sessions. After those n sessions, the key renewing will be made with one of the two devices (the legit or the cloned device). After the renewing, we have two possibilities. First, the legit user has made the renewing and then the cloned has not the new secret s and cannot process any session (the device should be cloned again). In the second case, the cloned device has made the renewing, but the user cannot process a session and then he detects the cloning.

This cloning detection is efficient only if eavesdropping is made really hard, because the cloned device could make the renewing by listening the renewing processed by the legit device. This security finding is limited to very short range communication technologies for that reason.

6 Future Works and Conclusion

We have developed a generic and efficient authenticated key agreement protocol for lightweight devices. This protocol provides automated key renewing, contributivity and security against nonce corruption. This protocol has been verified to be secure with AVISPA and ProVerif. Future studies will bring new

protocols with concrete cryptographic primitives for some specific applications. For instance, the usage of our authenticated key agreement protocol can be applied to create more robust authenticated distance bounding protocols for RFID technologies.

References

1. Aboba, B., Blunk, L., Vollbrecht, J., and Carlson, J. Extensible authentication protocol *(EAP)*. RFC 3748, June 2004.
2. Blanchet, B. Automatic verification of correspondences for security protocols. *Journal of Computer Security 17*, 4 (July 2009), 363–434.
3. Challal, Y., Bouabdallah, A., and Hinard, Y. Efficient multicast source authentication using layered hash-chaining scheme. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks* (Washington, DC, USA, 2004), LCN '04, IEEE Computer Society, pp. 411–412.
4. Choi, S. Denial-of-service resistant multicast authentication protocol with prediction hashing and one-way key chain. In *Proceedings of the Seventh IEEE International Symposium on Multimedia* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 701–706.
5. Diffie, W., and Hellman, M. E. New directions in cryptography. In *IEEE Transactions on Information Theory* (1976), vol. 22, pp. 644–654.
6. Dolev, D., and Yao, A. On the security of public key protocols. *Information Theory, IEEE Transactions on 29*, 2 (1983), 198–208.
7. Fluhrer, S., Mantin, I., and Shamir, A. Weaknesses in the key scheduling algorithm of rc4. In *Proceedings of the 4th Annual Workshop on Selected Areas of Cryptography* (2001), S. B. . Heidelberg, Ed., pp. 1–24.
8. Hu, Y.-C., Perrig, A., and Jakobsson, M. Efficient constructions for one-way hash chains. In *Applied Cryptography and Network Security* (New York, NY, June 2005).
9. Jeong, I., Katz, J., and Lee, D. One-round protocols for two-party authenticated key exchange. In *Applied Cryptography and Network Security*, M. Jakobsson, M. Yung, and J. Zhou, Eds., vol. 3089 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2004, pp. 220–232.
10. Klein, A. Attacks on the rc4 stream cipher. *Des. Codes Cryptography 48* (September 2008), 269–286.
11. Krawczyk, H. Cryptographic extraction and key derivation: The hkdf scheme. In *Advances in Cryptology, CRYPTO*, T. Rabin, Ed., vol. 6223 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2010, pp. 631–648.
12. LaMacchia, B., Lauter, K., and Mityagin, A. Stronger security of authenticated key exchange. In *Proceedings of the 1st international conference on Provable security* (Berlin, Heidelberg, 2007), ProvSec'07, Springer-Verlag, pp. 1–16.
13. Lamport, L. Password authentication with insecure communication. *Communications of the ACM 24*, 11 (November 1981), 770–772.
14. Needham, R., and Schroeder, M. Using encryption for authentication in large networks of computers. *Communications of the ACM 21*, 12 (December 1978), 993–999.
15. Neuman, B. C., and Ts'o, T. Kerberos: An authentication service for computer networks. *IEEE Communications 32*, 9 (September 1994), 33–38.

16. Qachri, N., and Roggeman, Y. The flaws and critics about the security layer for the wimedia mac standard. In *30-th symposium on Information Theory in the Benelux* (may 2009), pp. 89–96.
17. Stubblefield, A., Ioannidis, J., and Rubin, A. D. Using the fluhrer, mantin, and shamir attack to break wep. Tech. Rep. TD-4ZCPZZ, AT&T Labs, 2001.
18. team, T. A. Avispa v1.1 user manual, June 2006.
19. team, T. A. Hlpsl tutorial: A beginner's guide to modelling and analysis internet security protocols, June 2006.
20. Wang, F., and Zhang, Y. A new provably secure authentication and key agreement mechanism for sip using certificateless public-key cryptography. In *2007 International Conference on Computational Intelligence and Security* (December 2007), IEEE, Ed., pp. 809–814.
21. Zou, X., Thukral, A., and Ramamurthy, B. An authenticated key agreement protocol for mobile ad hoc networks. In *Mobile Ad-hoc and Sensor Networks. Second International Conference, MSN 2006. Proceedings (Lecture Notes in Computer Science Vol. 4325)*. Springer-Verlag, January 2006.

Appendix A. HLPSL specification of the main subprotocol

```
%%% This is the Alice Role of the authenticated key agreement protocol
%%% where the input parameters are the main knowledge of Alice and Bob

role alice (A,B: agent,
   S: symmetric_key,
   Hash: hash_func,
   CC: nat,
   Sess: nat,
   SND,RCV : channel (dy))

played_by A def=
 local
  State: nat,
  Kconf: message,
  R1,R2: text,
  Authb: message,
  K1: symmetric_key,
  K2: symmetric_key,
  K3: symmetric_key,
  X : hash(message.hash(message.nat.nat.nat.text.text.agent)) %%% expression of the
                                                              %%% knowledge on the
                                                              %%% format  used to
                                                              %%% compute  the message
                                                              %%% authentication code


 %%% creation of the 3 keys from the shared secret
 init
  State :=1 /\
  K1:=Hash(Hash(Hash(Hash(S).S).S).S) /\
  K2:=Hash(Hash(Hash(S).S).S) /\
  K3:=Hash(Hash(S).S)

 transition

 %%% First message that launchs the key agreement with a fresh Nonce R1

 1. State=0 /\ RCV(start) =|>
    State':= 2 /\ R1':=new() /\ SND(A.Sess.{1.CC.Sess.R1'.A.Hash(1.CC.Sess.R1'.A)}_K1)
```

```
 %%% Answer of Bob with R2 and the MAC that confirms that the key derived by Alice is
 %%% the same than Bob

 2. State=2 /\ RCV(B.Sess.{2.CC.Sess.R2'.R1.B.X'}_K1)
       /\ X' = Hash(xor(Hash(K2.Hash(K2.CC.Sess.R1.R2'.K3)) ,23644).Hash(xor(Hash(K2.
          Hash(K2.CC.Sess.R1.R2'.K3)),13878).2.CC.Sess.R2'.R1.B))
          =|> State':=4 /\ Kconf':=Hash(K2.Hash(K2.CC.Sess.R1.R2'.K3))
            /\ SND(Hash(xor(Kconf',23644).Hash(xor(Kconf',13878).3.A.CC.Sess.R1.R2)) )
            /\ witness(A,B,alice_bob_R2,R2')
            /\ witness(A,B,derive_ab_ks,Kconf')
            /\ request(A,B,bob_alice_R1,R1)

end role

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% This is the role played by Bob in the system
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

role bob (
 A,B: agent,
 S: symmetric_key,
 Hash: hash_func,
 CC: nat,
 Sess: nat,
 SND,RCV : channel (dy))

played_by B def=
 local
  State: nat,
  Kconf: message,
  R1,R2: text,
  Autha: message,
  K1: symmetric_key,
  K2: symmetric_key,
  K3: symmetric_key,
  X3: hash(nat.nat.nat.text.agent),
  X2: hash(message.hash(message.nat.agent.nat.nat.text.text))


 init
  State :=1 /\
  K1:=Hash(Hash(Hash(Hash(S).S).S).S) /\
  K2:=Hash(Hash(Hash(S).S).S) /\
  K3:=Hash(Hash(S).S)

 transition

 %%% Reception of the message of Alice that wants make a key agreement
 %%% based on the secret shared keys, on the fresh nonce of Alice and on
 %%% the nonce chosen by Bob

 1. State=1 /\ RCV(A.Sess.{1.CC.Sess.R1'.A.X3'}_K1) /\ Hash(1.CC.Sess.R1'.A) = X3'
     =|>  State':= 3 /\ R2':=new()  /\ Kconf':= Hash(xor(K2,23644).Hash(xor(K2,13878).
          CC.Sess.R1.R2'.K3)) /\ SND(B.Sess.{2.CC.Sess.R2'.R1'.B.Hash(xor(Kconf',
          23644).Hash(xor(Kconf',13878).2.CC.Sess.R2'.R1.B))}_K1) /\ secret(Kconf',
          kconf,{A,B}) /\ witness(A,B,bob_alice_R1,R1')

 %%% Last received message where Bob checks the good ending of the protocol and that
 %%% the authentication is well achieved

 2. State = 3 /\ RCV(X2') /\ Hash(xor(Kconf,23644).Hash(xor(Kconf,13878).3.A.CC.Sess.
      R1.R2)) = X2'
      =|>  State':=5  /\ request(A,B,alice_bob_R2,R2) /\ request(A,B,derive_ab_ks,
      Kconf)

end role

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%% This role makes the composition of the roles of Alice and Bob and describes a
%%% session of the  protocol
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

role session(A,B : agent,
    S: symmetric_key,
    Hash : hash_func,
    CC: nat,
    Sess: nat)

def=
 %%% Define of the channel of communications
 local SA, SB, RA, RB: channel (dy)

composition
 alice (A, B, S, Hash, CC, Sess, SA, RA)
 /\ bob (A, B, S, Hash, CC, Sess, SB, RB)

end role

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

role environment()
def=

const
  kconf, bob_alice_R1, alice_bob_R2, derive_ab_ks : protocol_id,
  kab, kai, kib     : symmetric_key,
  a, b : agent,
  h  : hash_func


intruder_knowledge = {a, b, h, kai, kib}  %%% it defines what knows an intruder i

%% The test involves 4 parallel sessions

composition

 %% Two legitimates sessions between Alice and Bob are executed with consecutive number
 %% and keys of session

 session(a, b, kab, h, 1, 1)  /\ session (a,b, h(h(h(h(kab).kab).kab.).kab), h, 1,2)

 %% Two sessions where the intruder plays one of the two role in a protocol session

 /\ session(a, i, kai, h, 1, 2)  /\ session(i, b, kib, h, 1, 3)

end role

 %%% The goal section describes the security goals that must achieve the protocol to
 %%% confirm the mutual authentication and to ensure the secrecy of the derived keys

goal

 secrecy_of kconf
 authentication_on bob_alice_R1
 authentication_on alice_bob_R2
 authentication_on derive_ab_ks

end goal

environment()
```

Appendix B. ProVerif specification of the main subprotocol

```
(* T Y P E S *)
type N. (*nonce*)
type M. (*message*)
type I. (*identity*)
type R. (*round*)
type O. (*other*)

(* S Y M B O L S *)
const RND1, RND2, RND3:R.
const IDA, IDB: I.
const CC, i1, i2, i3: O.


(* P R I M I T I V E S *)
fun h(bitstring): bitstring.
fun mac(bitstring, bitstring): bitstring.
fun senc(bitstring, bitstring): bitstring. (*/!\ this is authenticated encryption*)
reduc forall k: bitstring, x: bitstring; sdec(k, senc(k, x)) = x.


(* A D V E R S A R Y *)
free c: channel.
set attacker = active.
free s, secret1, secret2, secret3: bitstring [private].
query attacker (secret1).
query attacker (secret2).
query attacker (secret3).


(* P R O C E S S *)

let processA1 = new rA:N;
        let k2 = h(s) in
    let k1 = h((k2, s)) in
    let k0 = h((k1, s)) in
    let m = (RND1, CC, i1, rA, IDA) in
    let x = ( m, h(m) ) in
    out(c, (IDA, i1, senc(k0 , x)));
    in( c, (= IDB, = i1, y:bitstring) );
    let (mp:bitstring, macmp:bitstring) = sdec(k0 , y) in
        let (= RND2, = CC, = i1, rB :N, = rA, = IDB) = mp in
        let KEYS = mac(k2 , (CC, i1, rA, rB , k1 )) in
        if macmp = mac(KEYS , mp) then
          out(c, (IDA, i1, mac(KEYS , (RND3, IDA, CC, i1, rA, rB ))));
        out(c, senc(KEYS, secret1)).

let processB1 = new rB:N;
        let k2 = h(s) in
        let k1 = h((k2, s)) in
        let k0 = h((k1, s)) in
        in( c, (= IDA, = i1, y:bitstring));
        let (m:bitstring, = h(m)) = sdec(k0 , y) in
        let (= RND1, = CC, = i1, rA:N, = IDA) = m in
        let KEYS = mac(k2 , (CC, i1, rA, rB , k1 )) in
        let mp = (RND2, CC, i1, rB , rA, IDB) in
        let xp = (mp, mac(KEYS , mp)) in
        out(c, (IDB, i1, senc(k0 , xp)));
        in( c, (= IDA, = i1, (= RND3, = IDA, = CC, = i1, = rA, = rB )) );
        out(c, senc(KEYS, secret1)).

let processA2 = new rA:N;
        let tmp1 = h((h(s), s)) in
        let tmp2 = h((tmp1, s)) in
        let k2 = h((tmp2, s)) in
    let k1 = h((k2, s)) in
    let k0 = h((k1, s)) in
        let m = (RND1, CC, i2, rA, IDA) in
        let x = ( m, h(m) ) in
```

```
        out(c, (IDA, i2, senc(k0 , x )));
        in( c, (= IDB, = i2, y:bitstring) );
        let (mp:bitstring, macmp:bitstring) = sdec(k0 , y) in
        let (= RND2, = CC, = i2, rB :N, = rA, = IDB) = mp in
        let KEYS = mac(k2 , (CC, i2, rA, rB , k1 )) in
        if macmp = mac(KEYS , mp) then
        out(c, (IDA, i2, mac(KEYS , (RND3, IDA, CC, i2, rA, rB ))));
        out(c, senc(KEYS , secret2)).

let processB2 = new rB:N;
        let tmp1 = h((h(s), s)) in
        let tmp2 = h((tmp1 , s)) in
        let k2 = h((tmp2 , s)) in
    let k1 = h((k2 , s)) in
    let k0 = h((k1 , s)) in
        out(c, s);
        in( c, (= IDA, = i2, y:bitstring));
        let (m:bitstring, = h(m)) = sdec(k0 , y) in
        let (= RND1, = CC, = i2, rA:N, = IDA) = m in
        let KEYS = mac(k2 , (CC, i2, rA, rB , k1 )) in
        let mp = (RND2, CC, i2, rB , rA, IDB) in
        let xp = (mp, mac(KEYS , mp)) in
        out(c, (IDB, i2, senc(k0 , xp)));
        in( c, (= IDA, = i2, (= RND3, = IDA, = CC, = i2, = rA, = rB )) );
        out(c, senc(KEYS , secret2)).

let processA3 = new rA:N;
        let tmp1 = h((h(s), s)) in
        let tmp2 = h((tmp1 , s)) in
        let tmp3 = h((tmp2 , s)) in
        let tmp4 = h((tmp3 , s)) in
        let tmp5 = h((tmp4 , s)) in
        let k2 = h((tmp5 , s)) in
    let k1 = h((k2 , s)) in
    let k0 = h((k1 , s)) in
        let m = (RND1, CC, i3, rA, IDA) in
        let x = ( m, h(m) ) in
        out(c, (IDA, i3, senc(k0 , x )));
        in( c, (= IDB, = i3, y:bitstring) );
        let (mp:bitstring, macmp:bitstring) = sdec(k0 , y) in
        let (= RND2, = CC, = i3, rB :N, = rA, = IDB) = mp in
        let KEYS = mac(k2 , (CC, i3, rA, rB , k1 )) in
        if macmp = mac(KEYS , mp) then
        out(c, (IDA, i3, mac(KEYS , (RND3, IDA, CC, i3, rA, rB ))));
        out(c, senc(KEYS , secret3)).

let processB3 = new rB:N;
        let tmp1 = h((h(s), s)) in
        let tmp2 = h((tmp1 , s)) in
        let tmp3 = h((tmp2 , s)) in
        let tmp4 = h((tmp3 , s)) in
        let tmp5 = h((tmp4 , s)) in
        let k2 = h((tmp5 , s)) in
    let k1 = h((k2 , s)) in
    let k0 = h((k1 , s)) in
        in( c, (= IDA, = i3, y:bitstring));
        let (m:bitstring, = h(m)) = sdec(k0 , y) in
        let (= RND1, = CC, = i3, rA:N, = IDA) = m in
        let KEYS = mac(k2 , (CC, i3, rA, rB , k1 )) in
        let mp = (RND2, CC, i3, rB , rA, IDB) in
        let xp = (mp, mac(KEYS , mp)) in
        out(c, (IDB, i3, senc(k0 , xp)));
        in( c, (= IDA, = i3, (= RND3, = IDA, = CC, = i3, = rA, = rB )) );
        out(c, senc(KEYS , secret3)).


process (!processA1 |!processB1 )
        | (!processA2 |!processB2 )
```

| (!*processA3* |!*processB3*)

Relation among the Security Models for RFID Authentication Protocol

Daisuke Moriyama[1], Shin'ichiro Matsuo[1], and Miyako Ohkubo[1]

National Institute of Information and Communications Technology (NICT), Japan
{dmoriyam,smatsuo,m.ohkubo}@nict.go.jp

Abstract. In this paper, we present the relationship between the privacy definition for Radio Frequency Identification (RFID) authentication protocol. The security model is necessary to ensure the security or privacy, but many researchers describe different privacy notion for RFID authentication and the technical relationship among them is unclear. We reconsider the zero-knowledge privacy proposed by Deng et al. in ESORICS 2010 and show that this privacy is equivalent to the indistinguishability based privacy proposed by Juels and Weis. Furthermore, we present that these privacy definitions are technically weaker than the simulation based privacy proposed by Paise and Vaudenay in AsiaCCS 2008.

1 Introduction

Radio Frequency Identification (RFID) technology enables the reader to identify objects. Briefly speaking, RFID systems consist of a reader and many tags. The reader communicates with the tags over the wireless (insecure) channel and checks the identity. RFID is expected to replace barcode and it is now used in many industries (manufacturing, transportation, logistics, etc.). However, the existing low-cost tags only contain its identity without any protection and respond their identity directly when the reader provides the electric power. Therefore, many cryptographer have been studied the RFID authentication protocol to overcome the privacy problem.

In cryptography, the security/privacy of each schemes or protocols is evaluated by the security model. There are several security models for RFID authentication protocols [4, 5, 7, 8, 11–14, 16]. All these models define the three component: correctness, security and privacy. The correctness and security definitions are almost same in these models. The correctness ensures that the reader accepts the tag if the reader and tag correctly communicate each other. The

security requires that if a malicious adversary impersonates a valid tag and interferes the communication, then the reader rejects and aborts the session. However, the privacy notion is not commonly defined and the relationship between them is unclear. In this paper, we concentrate on the privacy definitions for the RFID authentication protocol and investigate the relationship.

Our Contributions. Our contributions are twofold:

1. We show that the indistinguishability based privacy definition proposed by Juels-Weis [10,11] and zero-knowledge based privacy definition proposed by Deng et al. [7] are equivalent. Though Deng et al. provided zero-knowledge based privacy is stronger than indistinguishability based privacy, we show that their argument is not adequate and these privacy definitions are proven to be equivalent.
2. There is a technical gap between indistinguishability based privacy and simulation based privacy proposed by Paise and Vaudenay [16]. There are many existing RFID authentication protocols which are proven to be secure in one of the two security models or its slight variants [9, 15], but no one investigates whether there exists a technical difference[1] between [11] and [16]. These privacy definitions are formalized in different style and it is hard to present the difference directly. Hence, we consider a variant of zero-knowledge based privacy proposed in [7] to reduce the gap between them (this variant is polynomially equivalent to the Juels-Weis security model). Then we compare the resulting privacy definition with [16] and show that Paise-Vaudenay privacy definition requires reader's privacy in addition to the tag's privacy.

Related Work. Ha et al. proposed an unpredictability based privacy model [8] and it was refined by several researchers [12,14]. This kind of privacy model requires that at least the tag's response to the reader is indistinguishable from random string. Ma et al. [14] showed that (1) the unpredictability based privacy model requires strictly stronger privacy than the indistinguishability based privacy

[1] We ignore the several minor differences including the timing of the corrupt query and registration of the new tags.

model [11], and (2) the existence of an RFID authentication protocol which satisfies the unpredictability based privacy model equals the existence of pseudo-random function. The pseudo-random function is used in many lightweight RFID authentication protocol, but we consider the unpredictability based privacy is too strong to satisfy. For example, if both the reader and tag can perform IND-CCA2 secure public key encryption and all communication is encrypted by each party's public key, then none of the secret information is revealed by the communication. However, the ciphertext usually consists of group elements and is easily distinguishable from random string.

The other privacy formalization is universal composability based privacy model [4, 5, 13]. This model requires that any actions of the malicious adversary are simulated by a simulator and any external environment cannot distinguish whether it interacts with the adversary or simulator. The authors did not describe the relationship between their model and the other privacy model, but Paise and Vaudenay demonstrated the RFID authentication protocol depicted in [5] does not hold narrow-forward privacy in the Paise-Vaudenay privacy model [16].

Throughout the paper we use the following notation. When A is a probabilistic machine or algorithm, $A(x)$ denotes the random variable of the output of A on input x. If $\mathcal{O}$ is an oracle, $A^{\mathcal{O}}$ denotes that A can issue $\mathcal{O}$ as oracle query. $y \stackrel{\mathsf{R}}{\leftarrow} A(x)$ denotes that y is randomly selected from $A(x)$ according to its distribution. Then, $A(x) \rightarrow a$ indicates the event that A outputs a on input x if a is a value. When A is a set, $y \stackrel{\mathsf{U}}{\leftarrow} A$ means that y is uniformly selected from A. When A is a value, $y := A$ denotes that y is set as A.

2 Existing RFID Security Models

We review security models proposed by Juels-Weis [11], Deng-Li-Yung-Zhao [7] and Paise-Vaudenay [16], respectively. Especially, we only concentrate on the privacy definition in the security model and call it as privacy model. We denote by $\mathcal{T}$ the total set of tags in the RFID authentication protocol that communicates with the reader $\mathcal{R}$. The reader runs Setup algorithm and obtains (pk, sk). The public pa-

rameter pk is published and secret key sk is kept as secret. If the RFID authentication protocol is based on symmetric key cryptography, each tag shares several secret keys with the reader (sk is the set of these secret keys). In the authentication phase, the reader and the tag communicate each other with wireless communication. We consider an active adversary $\mathcal{A}$ who can interfere/insert/delete/modify the communication message and its direction.

2.1 Juels-Weis Privacy Model

Juels and Weis proposed a privacy model for RFID authentication protocol based on indistinguishability [11]. We describe a slight variant of the privacy model modified by Deng et al. [7]. The privacy game between an adversary $\mathcal{A} := (\mathcal{A}_1, \mathcal{A}_2)$ and challenger is defined as follows:

Setup. The challenger runs Setup algorithm and obtains (sk, pk) to setup the reader $\mathcal{R}$ and set of tags $\mathcal{T}$. The adversary obtains public parameter pk and $(\mathcal{R}, \mathcal{T})$.

Phase 1. The adversary $\mathcal{A}_1$ can issue oracle queries $\mathcal{O} := \{\mathsf{Launch}, \mathsf{SendReader}, \mathsf{SendTag}, \mathsf{Result}, \mathsf{Corrupt}\}$ and interact with the reader and tags:

- $\mathsf{Launch}(1^k)$ — Launch the reader to initiate the session.
- $\mathsf{SendReader}(m)$ — Send arbitrary message m to the reader.
- $\mathsf{SendTag}(t, m)$ — Send arbitrary message m to the tag $t \in \mathcal{T}$.
- $\mathsf{Result}(\mathsf{sid})$ — Output whether the reader accepts or not for the session sid (sid is uniquely determined by the communication message).
- $\mathsf{Corrupt}(t)$ — Output the secret key of the tag t.

Challenge. The adversary sends two tags t_0^* and t_1^* ($t_0^* \neq t_1^*$) to the challenger and outputs state information st_1. Then the challenger flips a coin $b \overset{\mathsf{U}}{\leftarrow} \{0, 1\}$ and sets $\mathcal{T}' := \mathcal{T} \setminus \{t_0^*, t_1^*\}$.

Phase 2. The adversary $\mathcal{A}_2$ obtains st_1 and interacts with the reader $\mathcal{R}$ and tags $(t_b^*, \mathcal{T}')$ with the oracle queries. However, when the adversary interacts with the challenge tag t_b^*, we consider special algorithm $\mathcal{I}$. $\mathcal{I}$ relays the message between $\mathcal{A}$ and t_b^* so that the adversary communicates with t_b^* anonymously.

Guess. The adversary $\mathcal{A}_2$ outputs a guess b'.

We say that the adversary wins the game if $b' = b$ holds and (t_0^*, t_1^*) is not corrupted. The advantage of the adversary in the above game is defined as $\mathsf{Adv}_{\Pi,\mathcal{A}}^{\mathsf{IND}}(k) := |2 \cdot \Pr[b' = b] - 1|$. This advantage is also evaluated by the following experiment

$$\begin{array}{l} \underline{\mathsf{Exp}_{\Pi,\mathcal{A}}^{\mathsf{IND}\text{-}b}(k)} \\ (pk, sk) \stackrel{\mathsf{R}}{\leftarrow} \mathsf{Setup}(1^k); \\ (t_0^*, t_1^*, st_1) \stackrel{\mathsf{R}}{\leftarrow} \mathcal{A}_1^{\mathcal{O}}(pk, \mathcal{R}, \mathcal{T}); \\ b \stackrel{\mathsf{U}}{\leftarrow} \{0,1\}, \mathcal{T}' := \mathcal{T} \setminus \{t_0^*, t_1^*\}; \\ b' \stackrel{\mathsf{R}}{\leftarrow} \mathcal{A}_2^{\mathcal{O}}(\mathcal{R}, \mathcal{T}', \mathcal{I}(t_b^*), st_1): \\ \text{Output } b' \end{array}$$

and we have $\mathsf{Adv}_{\Pi,\mathcal{A}}^{\mathsf{IND}}(k) = |\Pr[\mathsf{Exp}_{\Pi,\mathcal{A}}^{\mathsf{IND}\text{-}0}(k) \to 1] - \Pr[\mathsf{Exp}_{\Pi,\mathcal{A}}^{\mathsf{IND}\text{-}1}(k) \to 1]|$.

Definition 1. *An RFID authentication protocol Π satisfies the privacy in the Juels-Wies security model if for any probabilistic polynomial time adversary $\mathcal{A}$, $\mathsf{Adv}_{\Pi,\mathcal{A}}^{\mathsf{IND}}(k)$ is negligible.*

In this paper, we call this privacy as IND-privacy.

2.2 Deng-Li-Yung-Zhao Privacy Model

The privacy model proposed by Deng et al. is based on a zero-knowledge formulation [7]. The intuition behind this privacy model is that when the communication message does not reveal any tag's identity nor secret key, then these messages should be simulated even if an algorithm cannot interact with the tag.

We consider two experiments $\mathsf{Exp}_{\mathcal{A},\mathcal{D}}^{\mathsf{ZK}\text{-}0}(k)$ and $\mathsf{Exp}_{\mathcal{S},\mathcal{D}}^{\mathsf{ZK}\text{-}1}(k)$. In the former experiment, the adversary $\mathcal{A}$ interacts with the reader and tags. $\mathcal{A}$ outputs an arbitrary subset of tags $\mathcal{C} \subseteq \mathcal{T}$ and the challenger uniformly chooses a challenge tag $t^* \stackrel{\mathsf{U}}{\leftarrow} \mathcal{C}$ at random. Then the adversary can interacts with $\mathcal{R}$, tags $\mathcal{T}' := \mathcal{T} \setminus \mathcal{C}$ and the challenge tag t^* anonymously. When the adversary sends message m to $\mathcal{I}$, this algorithm passes m to t^* and responds the output from t^*. Finally the adversary outputs its view and an distinguisher outputs a bit b with the view. The latter experiment is just same as the former experiment except that the simulator $\mathcal{S}$ cannot interact with the challenge tag. We note that the adversary and simulator cannot

issue any corrupt queries to the tags in $\mathcal{C}$ in the experiment. These experiments are depicted as follows:

$\mathsf{Exp}^{\mathsf{ZK\text{-}0}}_{\Pi,\mathcal{A},\mathcal{D}}(k)$	$\mathsf{Exp}^{\mathsf{ZK\text{-}1}}_{\Pi,\mathcal{S},\mathcal{D}}(k)$
$(pk, sk) \xleftarrow{\mathsf{R}} \mathsf{Setup}(1^k)$;	$(pk, sk) \xleftarrow{\mathsf{R}} \mathsf{Setup}(1^k)$;
$(\mathcal{C}, st_1) \xleftarrow{\mathsf{R}} \mathcal{A}_1^{\mathcal{O}}(pk, \mathcal{R}, \mathcal{T})$;	$(\mathcal{C}, st_1) \xleftarrow{\mathsf{R}} \mathcal{S}_1^{\mathcal{O}}(pk, \mathcal{R}, \mathcal{T})$;
$t^* \xleftarrow{\mathsf{U}} \mathcal{C}, \mathcal{T}' := \mathcal{T} \setminus \mathcal{C}$;	$t^* \xleftarrow{\mathsf{U}} \mathcal{C}, \mathcal{T}' := \mathcal{T} \setminus \mathcal{C}$;
$view_{\mathcal{A}} \xleftarrow{\mathsf{R}} \mathcal{A}_2^{\mathcal{O}}(\mathcal{R}, \mathcal{T}', \mathcal{I}(t^*), st_1)$;	$view_{\mathcal{S}} \xleftarrow{\mathsf{R}} \mathcal{S}_2^{\mathcal{O}}(\mathcal{R}, \mathcal{T}', st_1)$;
$b \xleftarrow{\mathsf{R}} \mathcal{D}(\mathcal{C}, t^*, view_{\mathcal{A}})$:	$b \xleftarrow{\mathsf{R}} \mathcal{D}(\mathcal{C}, t^*, view_{\mathcal{S}})$:
Output b	Output b

In this privacy model, the advantage of the adversary is defined by $\mathsf{Adv}^{\mathsf{ZK}}_{\Pi,\mathcal{A},\mathcal{S},\mathcal{D}}(k) = |\Pr[\mathsf{Exp}^{\mathsf{ZK\text{-}0}}_{\Pi,\mathcal{A},\mathcal{D}}(k) \to 1] - \Pr[\mathsf{Exp}^{\mathsf{ZK\text{-}1}}_{\Pi,\mathcal{S},\mathcal{D}}(k) \to 1]|$.

Definition 2. *An RFID authentication protocol Π satisfies the privacy in the Deng et al. security model if for any probabilistic polynomial time adversary $\mathcal{A}$, there exists a probabilistic polynomial time algorithm $\mathcal{S}$, for any probabilistic polynomial time distinguisher $\mathcal{D}$, $\mathsf{Adv}^{\mathsf{ZK}}_{\Pi,\mathcal{A},\mathcal{S},\mathcal{D}}(k)$ is negligible.*

In the following, we call this privacy as ZK-privacy.

2.3 Paise-Vaudenay Privacy Model

Vaudenay [18] proposed a simulation based privacy model for two-pass RFID authentication protocol. Paise and Vaudenay [16] extended it to satisfy reader authentication. The privacy game of their model is slightly similar to the Deng et al.'s privacy model, but the game flow is not explicitly defined. Instead, the adversary additionally can issue the following queries:

$\mathsf{CreateTag}$(ID) — Register a free tag to the reader. The reader assigns the secret key for this tag and updates the database.
$\mathsf{DrawTag}(\mathcal{C}, \mathsf{Dist})$ — According to the distribution Dist and the arbitrary sets of tags $\mathcal{C} \subseteq \mathcal{T}$, the oracle responds drawn tags $\mathcal{V} := \{\mathsf{vtag}_1, \ldots\}$. The oracle keeps a list list that maps the drawn tags to the real identity.
$\mathsf{Free}(\mathsf{vtag})$ — Change the drawn tag vtag to the free tag.

In their model, the challenger assigns temporal identity to each drawn tag. The adversary can issue $\mathsf{SendTag}$ query to the drawn

tags only and free tags do not execute the communication to the reader. These queries are useful when we consider the tag's recycle.

Paise and Vaudenay classifies the adversary's capacity into 2×4 categories.

1. Result query for the reader:
   (a) *Wide* — The adversary can issue result query.
   (b) *Narrow* — The adversary cannot issue result query.
2. Corrupt query for the tag:
   (a) *Strong* — No restriction for the corrupt query.
   (b) *Destructive* — If the adversary issues the corrupt query to a drawn tag, the tag is destroyed and unusable.
   (c) *Forward* — After the corrupt query, the adversary cannot issue any other queries in the experiment.
   (d) *Weak* — The adversary cannot issue corrupt query.

For example, wide-strong privacy is defined as follows. Consider the sets of the oracle queries $\mathcal{O}_1 := \{\mathsf{CreateTag}, \mathsf{DrawTag}, \mathsf{Free}, \mathsf{Corrupt}\}$ and $\mathcal{O}_2 := \{\mathsf{Launch}, \mathsf{SendReader}, \mathsf{SendTag}, \mathsf{Result}\}$. The wide-strong privacy game in this model is defined by the following experiments:

$\mathsf{Exp}^{\mathsf{SIM\text{-}0}}_{\Pi,\mathcal{A}}(k)$	$\mathsf{Exp}^{\mathsf{SIM\text{-}1}}_{\Pi,\mathcal{A},\mathcal{S}}(k)$
$(pk, sk) \stackrel{\mathsf{R}}{\leftarrow} \mathsf{Setup}(1^k)$;	$(pk, sk) \stackrel{\mathsf{R}}{\leftarrow} \mathsf{Setup}(1^k)$;
$b \stackrel{\mathsf{R}}{\leftarrow} \mathcal{A}^{\mathcal{O}_1,\mathcal{O}_2}(pk, \mathcal{R})$:	$b \stackrel{\mathsf{R}}{\leftarrow} \mathcal{A}^{\mathcal{S}^{\mathcal{O}_1}(pk)}(pk)$:
Output b	Output b

In the SIM-0 experiment, adversary $\mathcal{A}$ can create tags and interact with reader and tags through $\mathcal{O}_2$ query. On the contrary, the SIM-1 experiment requires that simulator $\mathcal{S}$ responds the adversary's oracle queries. Note that $\mathcal{S}$ cannot issue $\mathcal{O}_2$ query and all communication messages must be simulated by $\mathcal{S}$. The advantage of the adversary is defined by $\mathsf{Adv}^{\mathsf{SIM}}_{\Pi,\mathcal{A},\mathcal{S}}(k) := |\Pr[\mathsf{Exp}^{\mathsf{SIM\text{-}0}}_{\Pi,\mathcal{A}}(k) \to 1] - \Pr[\mathsf{Exp}^{\mathsf{SIM\text{-}1}}_{\Pi,\mathcal{A},\mathcal{S}}(k) \to 1]|$. Of course, we can formalize the other types of adversary in the same fashion.

Definition 3. *An RFID authentication protocol Π satisfies the (wide/ narrow)-(strong/descructive/forward/weak) privacy in the Paise-Vaudenay security model if for any probabilistic polynomial time adversary $\mathcal{A}$, there exists a probabilistic polynomial time algorithm $\mathcal{S}$, $\mathsf{Adv}^{\mathsf{SIM}}_{\Pi,\mathcal{A},\mathcal{S}}(k)$ is negligible.*

In the following, we call this privacy as SIM-privacy.

3 Equivalence between IND and ZK Privacy

In the previous section, we described the three privacy models. Deng et al. [7] showed that their ZK-privacy is stronger than IND-privacy. That is, there exists two examples of the RFID authentication protocols which are secure in the Juels-Weis privacy model but insecure in the zero-knowledge privacy model. However, we will show that these privacy models are proven to be equivalent. To justify our result, we first review their examples and point out the *flaw* of their argument.

The former example is constructed by digital signature scheme. In the setup phase, a reader generates signing/verification key pair (sk, vk) and sends the signature of the tag's identity $\sigma_i \stackrel{R}{\leftarrow} \mathsf{Sign}(sk, t_i)$ as a secret key. To authenticate the tag, the reader outputs a request message and the tag responds σ_i itself. Deng et al. argued that *"If the system has only one tag, it is clear to satisfy the IND-privacy but the simulator cannot simulate the signature at Phase 2 in the ZK-privacy"*. But we remark that this implication does not make sense. As we explicitly describe in Sect 2.1, IND-privacy assumes that the adversary must output two different tags (it is also implicitly assumed in the IND-CPA security for public key encryption). Thus their instantiation is not adequate to consider the IND-privacy. If we consider there are more than two tags in the system, it is clear that the adversary against IND-privacy can distinguish the message since the output of the tag's message is deterministically defined. Moreover, even in the case for ZK-privacy, the simulator can simulate the message properly since it can access to the (only one) tag in the Phase 1 and obtain the message (again, we note that this message is always reused when the tag receives a request message).

The building block of the latter example is public key encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ and an RFID authentication protocol Π which holds IND-privacy. Following [7], we assume that when the reader sends a to the tag, it responds b to the reader in Π. Now we consider the following RFID authentication protocol Π'. In the setup phase, a reader generates public/secret key pair $(pk_{\mathsf{PKE}}, sk_{\mathsf{PKE}}) \stackrel{R}{\leftarrow} \mathsf{Gen}(1^k)$ and sends sk_{PKE} to the tags (we remark that all tags in this protocol shares this unique secret key) as a secret key for Π'. When the reader authenticates the tag, it generates a and sends encrypted

message $c \stackrel{\mathsf{R}}{\leftarrow} \mathsf{Enc}(pk_{\mathsf{PKE}}, a)$. If the tag receives the message, it decrypts as $a := \mathsf{Dec}(sk_{\mathsf{PKE}}, c)$, generates b with Π and responds $a \| b$ to the reader. Deng et al. said that Π' satisfies IND-privacy and does not satisfy ZK-privacy since no simulator can output the decryption of the ciphertext. However, we found that this argument is also wrong and Π' still holds ZK-privacy. Since the communication message is indistinguishable, simulator $\mathcal{S}_1$ can internally run zero-knowledge adversary $(\mathcal{A}_1, \mathcal{A}_2)$. No problem happens for the simulation of $\mathcal{A}_1$ and $\mathcal{A}_1$ outputs $(\mathcal{C}, st_1)$. When $\mathcal{A}_1$ outputs $(\mathcal{C}, st_1)$, $\mathcal{S}_1$ uniformly chooses $t_1^* \stackrel{\mathsf{U}}{\leftarrow} \mathcal{C}$ and runs $\mathcal{A}_2$ with input $(pk, \mathcal{R}, \mathcal{T} \setminus \mathcal{C}, \mathcal{I}(t_1^*), st_1)$. Remark that t_1^* may not be identical to the challenge tag, but IND-privacy ensures that no adversary can distinguish whether he interacts with the challenge tag or t_1^*. If $\mathcal{A}_2$ sends a message to the challenge tag, $\mathcal{S}_1$ simply sends it to t_1^* and responds its message. When $\mathcal{A}_2$ outputs $view_{\mathcal{A}}$, then $\mathcal{S}_1$ sets $st_1' := view_{\mathcal{A}}$ and outputs $(\mathcal{C}, st_1')$. Finally, $\mathcal{S}_2$ outputs st_1' as its view regardless of the choice of the challenge tag. Since the simulator can continue Phase 1 until the adversary outputs the view (Phase 1 and 2 for the adversary), these outputs are indistinguishable for any distinguisher $\mathcal{D}$. Of course, if we try to simulate the response of the $\mathsf{SendTag}$ query issued by $\mathcal{A}_2$ with $\mathcal{S}_2$, it is difficult to construct such a simulator since $\mathcal{S}_2$ must break the security for public key encryption. The key point here is that IND-privacy allows $\mathcal{S}_1$ to simulate the whole behavior of the zero-knowledge adversary $(\mathcal{A}_1, \mathcal{A}_2)$.

Now, we show that IND-privacy is equivalent to ZK-privacy.

Theorem 1. *The indistinguishability based privacy model is equivalent to the zero-knowledge based privacy model.*

Lemma 1. *If an RFID authentication protocol Π holds IND-privacy, it implies ZK-privacy.*

Proof. We prove the above theorem by the following sequence of games. Especially, we show that if for any IND adversary $\mathcal{B}$, $\mathsf{Adv}^{\mathsf{IND}}_{\Pi,\mathcal{B}}(k)$ is negligible, then for any ZK adversary $\mathcal{A}$, there exists an simulator $\mathcal{S}$, for any distinguisher $\mathcal{D}$, $\mathsf{Adv}^{\mathsf{ZK}}_{\Pi,\mathcal{A},\mathcal{S},\mathcal{D}}(k)$ is negligible. For each game, $\Pr[T_j]$ denotes the probability that the distinguisher outputs 1 in Game j.

Game 0: Game 0 is same as the original ZK-privacy game between a challenger and $\mathcal{A}$. Without loss of generality, we assume that $t_0^* \overset{\mathsf{U}}{\leftarrow} \mathcal{C}$ is chosen as the challenge tag. It is clear that $\Pr[T_0] = \Pr[\mathsf{Exp}^{\mathsf{ZK\text{-}0}}_{\Pi,\mathcal{A},\mathcal{D}}(k) \to 1]$.

Game 1: We modify Game 1 by changing the challenge tag. In addition to t_0^*, we select $t_1^* \overset{\mathsf{U}}{\leftarrow} \mathcal{C}$ and the adversary (anonymously) interacts with t_1^* instead of t_0^*.

Game 2: Game 2 is the original ZK-privacy game between a challenger and $\mathcal{S}$ under the condition that $\mathcal{S}$ runs $\mathcal{A}$ as Fig. 2. Remark that the challenge tag is chosen as Game 0 and the inputs to the distinguisher is t_0^*.

Game 0 (ZK-0)	Game 1	Game 2 (ZK-1)
$(pk, sk) \overset{\mathsf{R}}{\leftarrow} \mathsf{Setup}(1^k)$;	$(pk, sk) \overset{\mathsf{R}}{\leftarrow} \mathsf{Setup}(1^k)$;	$(pk, sk) \overset{\mathsf{R}}{\leftarrow} \mathsf{Setup}(1^k)$;
$(\mathcal{C}, st_1) \overset{\mathsf{R}}{\leftarrow} \mathcal{A}_1^{\mathcal{O}}(pk, \mathcal{R}, \mathcal{T})$;	$(\mathcal{C}, st_1) \overset{\mathsf{R}}{\leftarrow} \mathcal{A}_1^{\mathcal{O}}(pk, \mathcal{R}, \mathcal{T})$;	$(\mathcal{C}, st_1) \overset{\mathsf{R}}{\leftarrow} \mathcal{S}_1^{\mathcal{O}}(pk, \mathcal{R}, \mathcal{T})$;
$t_0^* \overset{\mathsf{U}}{\leftarrow} \mathcal{C}, \mathcal{T}' := \mathcal{T} \setminus \mathcal{C}$;	$t_0^*, t_1^* \overset{\mathsf{U}}{\leftarrow} \mathcal{C}, \mathcal{T}' := \mathcal{T} \setminus \mathcal{C}$;	$t_0^* \overset{\mathsf{U}}{\leftarrow} \mathcal{C}, \mathcal{T}' := \mathcal{T} \setminus \mathcal{C}$;
$view_{\mathcal{A}} \overset{\mathsf{R}}{\leftarrow} \mathcal{A}_2^{\mathcal{O}}(\mathcal{R}, \mathcal{T}', \mathcal{I}(t_0^*), st_1)$;	$view_{\mathcal{A}} \overset{\mathsf{R}}{\leftarrow} \mathcal{A}_2^{\mathcal{O}}(\mathcal{R}, \mathcal{T}', \mathcal{I}(t_1^*), st_1)$;	$view_{\mathcal{S}} \overset{\mathsf{R}}{\leftarrow} \mathcal{S}_2^{\mathcal{O}}(\mathcal{R}, \mathcal{T}', st_1)$;
$b \overset{\mathsf{R}}{\leftarrow} \mathcal{D}(\mathcal{C}, t_0^*, view_{\mathcal{A}})$:	$b \overset{\mathsf{R}}{\leftarrow} \mathcal{D}(\mathcal{C}, t_0^*, view_{\mathcal{A}})$:	$b \overset{\mathsf{R}}{\leftarrow} \mathcal{D}(\mathcal{C}, t_0^*, view_{\mathcal{A}})$:
Output b	Output b	Output b

Fig. 1. Game Transformation for Theorem 1

$\mathcal{S}_1^{\mathcal{O}}(pk, \mathcal{R}, \mathcal{T})$	$\mathcal{S}_2(\mathcal{R}, \mathcal{T}', st_1')$
$(\mathcal{C}, st_1) \overset{\mathsf{R}}{\leftarrow} \mathcal{A}_1^{\mathcal{O}}(pk, \mathcal{R}, \mathcal{T})$;	$view_{\mathcal{S}} := view_{\mathcal{A}}$:
$t_1^* \overset{\mathsf{U}}{\leftarrow} \mathcal{C}, \mathcal{T}' := \mathcal{T} \setminus \mathcal{C}$;	Output $view_{\mathcal{S}}$
$view_{\mathcal{A}} \overset{\mathsf{R}}{\leftarrow} \mathcal{A}_2^{\mathcal{O}}(\mathcal{R}, \mathcal{T}', \mathcal{I}(t_1^*), st_1)$;	
$st_1' := view_{\mathcal{A}}$:	
Output $(\mathcal{C}, st_1')$	

Fig. 2. Simulation in Game 2

The flow of the game transformation is depicted in Fig. 1. We evaluate the relations between pairs of advantages with the following claims.

Claim. There exists a probabilistic algorithm $\mathcal{B}$ such that

$$|\Pr[T_1] - \Pr[T_0]| \leq \mathsf{Adv}^{\mathsf{IND}}_{\Pi,\mathcal{B}}(k).$$

Proof. If $(\mathcal{A}, \mathcal{D})$ distinguishes Game 0 and Game 1 with non-negligible probability, we construct an algorithm $\mathcal{B} := (\mathcal{B}_1, \mathcal{B}_2)$ which can break the IND-privacy. $\mathcal{B}$ internally runs $(\mathcal{A}, \mathcal{D})$ in the IND-privacy game as follows:

$\mathcal{B}_1^{\mathcal{O}}(pk, \mathcal{R}, \mathcal{T})$	$\mathcal{B}_2^{\mathcal{O}}(pk, \mathcal{I}(t_b^*), st_1')$
$(\mathcal{C}, st_1) \stackrel{\mathsf{R}}{\leftarrow} \mathcal{A}_1^{\mathcal{O}}(pk, \mathcal{R}, \mathcal{T})$;	$view_{\mathcal{A}} \stackrel{\mathsf{R}}{\leftarrow} \mathcal{A}_2^{\mathcal{O}}(\mathcal{R}, \mathcal{T}', \mathcal{I}(t_b^*), st_1)$;
$t_0^*, t_1^* \stackrel{\mathsf{U}}{\leftarrow} \mathcal{C}, \mathcal{T}' := \mathcal{T} \setminus \mathcal{C}$;	$b' \stackrel{\mathsf{R}}{\leftarrow} \mathcal{D}(\mathcal{C}, t_0^*, view_{\mathcal{A}})$:
$st_1' := (\mathcal{T}', t_0^*, st_1)$:	Output b'
Output (t_0^*, t_1^*, st_1')	

When the adversary $\mathcal{A}_1$ outputs $\mathcal{C}$, $\mathcal{B}_1$ chooses two tags (t_0^*, t_1^*) in $\mathcal{C}$ and sends it to the challenger. Since the challenger chooses a coin $b \stackrel{\mathsf{U}}{\leftarrow} \{0, 1\}$ and $\mathcal{B}_2$ can access to $\mathcal{I}(t_b^*)$, $\mathsf{SendTag}$ query which $\mathcal{A}_2$ issues to the challenge tag can be completely simulated. If the flipped coin is $b = 0$, the output distribution is same as Game 0. Otherwise, this simulation is equivalent to Game 1. Therefore, we obtain

$$|\Pr[T_1] - \Pr[T_0]| \leq \left|\mathsf{Adv}^{\mathsf{IND\text{-}1}}_{\Pi,\mathcal{B}}(k) - \mathsf{Adv}^{\mathsf{IND\text{-}0}}_{\Pi,\mathcal{B}}(k)\right| = \mathsf{Adv}^{\mathsf{IND}}_{\Pi,\mathcal{B}}(k).$$

□

Claim. We have $\Pr[T_2] = \Pr[T_1]$.

Proof. We show that the output distribution of $\mathcal{A}$ in Game 1 is equivalent to that of $\mathcal{S}$ in Game 2. Recall that $\mathcal{S}_2$ cannot interact with the challenge tag in the original ZK-privacy game. Nonetheless, the previous claim shows that the anonymous interaction between $\mathcal{A}_2$ and t_0^* can be changed by another tag. This means that even if $\mathcal{S}_1$ chooses another tag $t_1^* \in \mathcal{C}$ and replaces the anonymous interaction by $\mathcal{I}(t_1^*)$, $\mathcal{A}_2$ cannot distinguish the difference between the games. Therefore $\mathcal{S}_1$ can simulate $(\mathcal{A}_1, \mathcal{A}_2)$ as Fig.2 and obtains the view of the adversary $view_{\mathcal{A}}$. Any oracle queries made by $(\mathcal{A}_1, \mathcal{A}_2)$ can be simulated correctly since $\mathcal{S}_1$ can send the same query to $\mathcal{O}$. Thus $\mathcal{A}_2$'s output in Game 1 is equivalent to $\mathcal{S}_2$'s output in Game 2 and it is (information theoretically) indistinguishable for any distinguisher $\mathcal{D}$. Therefore we have $\Pr[T_2] = \Pr[T_1]$. □

It is clear that $\Pr[T_2] = \Pr[\mathsf{Exp}^{\mathsf{ZK\text{-}1}}_{\Pi,\mathcal{B},\mathcal{D}}(k) \to 1]$ and finally we have $\mathsf{Adv}^{\mathsf{ZK}}_{\Pi,\mathcal{A},\mathcal{S},\mathcal{D}}(k) \leq \mathsf{Adv}^{\mathsf{IND}}_{\Pi,\mathcal{B}}(k)$. □

Remark. If the zero-knowledge adversary sets $\mathcal{C}$ as only one tag, then we can directly transform Game 0 to Game 2. The strategy of the simulator is same as Fig. 2. The simulator issues SendTag query in Phase 1 until the zero-knowledge adversary finishes the experiment.

Lemma 2. *If an RFID authentication protocol Π holds ZK-privacy, it implies IND-privacy* [2].

Proof. Again, we prove the above theorem by the following sequence of games. We show that if for any ZK adversary $\mathcal{A}$, there exists an simulator $\mathcal{S}$, for any distinguisher $\mathcal{D}$, $\mathsf{Adv}^{\mathsf{ZK}}_{\Pi,\mathcal{B},\mathcal{S},\mathcal{D}}(k)$ is negligible, then for any IND adversary $\mathcal{A}$, $\mathsf{Adv}^{\mathsf{IND}}_{\Pi,\mathcal{A}}(k)$ is negligible.

For each game, $\Pr[T_j]$ denotes the probability that the experiment outputs 1 in Game j.

Game 0: Game 0 is same as the original IND-0 privacy game between a challenger and $\mathcal{A}$. We consider $\mathcal{A}_1$ outputs two tags (t_0^*, t_1^*) and t_0^* is chosen as the challenge tag in this game. It is clear that $\Pr[T_0] = \Pr[\mathsf{Exp}^{\mathsf{IND\text{-}0}}_{\Pi,\mathcal{A}}(k) \to 1]$.

Game 1: We modify Game 1 by changing the challenge tag from t_0^* to t_1^*. It is clear that $\Pr[T_1] = \Pr[\mathsf{Exp}^{\mathsf{IND\text{-}1}}_{\Pi,\mathcal{A}}(k) \to 1]$.

Using $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, we construct the following ZK-privacy adversary $\mathcal{B} := (\mathcal{B}_1, \mathcal{B}_2)$ and distinguisher $\mathcal{D}$.

$\underline{\mathcal{B}_1^{\mathcal{O}}(pk, \mathcal{R}, \mathcal{T})}$
$(t_0^*, t_1^*, st_1) \overset{\mathsf{R}}{\leftarrow} \mathcal{A}_1^{\mathcal{O}}(pk, \mathcal{R}, \mathcal{T})$;
$\mathcal{C} := \{t_0^*, t_1^*\}$;
$st_1' := (st_1, t_0^*, t_1^*)$:
Output $(\mathcal{C}, st_1')$

$\underline{\mathcal{B}_2^{\mathcal{O}}(\mathcal{R}, \mathcal{T}', \mathcal{I}(t^*), st_1')}$
$b' \overset{\mathsf{R}}{\leftarrow} \mathcal{A}_2^{\mathcal{O}}(\mathcal{R}, \mathcal{T}', \mathcal{I}(t^*), st_1)$;
$view_{\mathcal{B}} := t_{b'}$:
Output $view_{\mathcal{B}}$

$\underline{\mathcal{D}(\mathcal{C}, t^*, view_{\mathcal{B}})}$
$t^* = view_{\mathcal{B}} \iff b := 1$;
$t^* \neq view_{\mathcal{B}} \iff b := 0$:
Output b

[2] This lemma has been provided by Deng et al. [7], but their proof is informal. So we give the precise security proof based on the game transformation technique.

The adversary $\mathcal{B}_1$ sets two tags (t_0^*, t_1^*) as $\mathcal{C}$ and one of the two tags can be accessed by $\mathcal{B}_2$. If t_0^* is chosen from $\mathcal{C}$, it is equivalent to Game 0 with respect to $\mathcal{A}$ and we obtain

$$\Pr[\mathsf{Exp}^{\mathsf{IND\text{-}0}}_{\Pi,\mathcal{A}}(k) \to 0] = 1 - \Pr[T_0] = \Pr[\mathsf{Exp}^{\mathsf{ZK\text{-}0}}_{\Pi,\mathcal{B},\mathcal{D}}(k) \to 1 \mid \mathcal{C} \to t_0^*].$$

Otherwise, it can be viewed as Game 1 and

$$\Pr[\mathsf{Exp}^{\mathsf{IND\text{-}1}}_{\Pi,\mathcal{A}}(k) \to 1] = \Pr[T_1] = \Pr[\mathsf{Exp}^{\mathsf{ZK\text{-}0}}_{\Pi,\mathcal{B},\mathcal{D}}(k) \to 1 \mid \mathcal{C} \to t_1^*].$$

Of course, the challenger uniformly selects the challenge tag and $\Pr[\mathcal{C} \to t_0^*] = \Pr[\mathcal{C} \to t_1^*] = 1/2$. Thus we obtain

$$\Pr[\mathsf{Exp}^{\mathsf{ZK\text{-}0}}_{\Pi,\mathcal{B},\mathcal{D}}(k) \to 1] = \frac{1}{2} + \frac{1}{2} \cdot (\Pr[T_1] - \Pr[T_0]).$$

Recall that we have assumed that Π is ZK-privacy. Thus, for any adversary $\mathcal{B}$, there exists an algorithm $\mathcal{S}$ such that for any distinguisher $\mathcal{D}$, $|\Pr[\mathsf{Exp}^{\mathsf{ZK\text{-}0}}_{\Pi,\mathcal{B},\mathcal{D}}(k) \to 1] - \Pr[\mathsf{Exp}^{\mathsf{ZK\text{-}1}}_{\Pi,\mathcal{S},\mathcal{D}}(k) \to 1]|$ is negligible. However, $\mathcal{S}$ has no information about the flipped coin in the experiment and we have $\Pr[\mathsf{Exp}^{\mathsf{ZK\text{-}1}}_{\Pi,\mathcal{S},\mathcal{D}}(k) \to 1] = 1/2$. Finally, we obtain

$$\begin{aligned}\mathsf{Adv}^{\mathsf{IND}}_{\Pi,\mathcal{B}}(k) &= |\Pr[T_1] - \Pr[T_0]| \\ &= |2 \cdot \Pr[\mathsf{Exp}^{\mathsf{ZK\text{-}0}}_{\Pi,\mathcal{B},\mathcal{D}}(k) \to 1] - 1| \\ &= 2 \cdot \mathsf{Adv}^{\mathsf{ZK}}_{\Pi,\mathcal{B},\mathcal{S},\mathcal{D}}(k).\end{aligned}$$

□

4 Relation between SIM and IND Privacy

4.1 Constraint for Corrupt Query

We revisit the privacy relation between SIM-privacy and IND-privacy. These models have been informally analyzed by many researcher and several papers conclude that SIM-privacy is stronger than IND-privacy since a wide-strong adversary can corrupt all tags in the experiment (recall that in the IND-privacy, the adversary must output uncorrupted tags for the challenge phase). However, there are four wide adversaries for SIM-privacy and it is meaningful to consider the other privacy notions. Recently, Vaudenay depicted that IND-privacy game can be written by the wide-destructive SIM-privacy

game [19]. Of course, the condition for the corrupt query in IND-privacy game is different from that in SIM-privacy game and we can say that wide-forward SIM-privacy does not imply IND-privacy in the sense of the adaptive corruption[3]. However, it is still an open problem whether IND-privacy implies wide-weak SIM-privacy. Furthermore, we can consider two variants for IND-privacy:

1. Strong IND-privacy — the challenge tags can be corrupted in Phase 1, and
2. Weak IND-privacy — the adversary cannot issue corrupt query to any tag.

Then, strong/weak IND-privacy is arguable to compare against wide-strong/wide-weak SIM-privacy. Of course, the actual procedure of the IND experiment is different from that of SIM experiment, but the restriction for the corrupt query in strong (resp. weak) IND-privacy is same as the wide-strong (resp. wide-weak) SIM-privacy. One can also consider these variants for the ZK-privacy which are equivalent to the strong/weak IND-privacy, respectively.

Interestingly, we prove that ZK-privacy variants do not imply SIM-privacy variants for any cases in the next subsection. We note one can think that the adaptive registration of the tag is allowed in SIM-privacy through SetupTag oracle, but it is not a technical point and can be easily modified.

4.2 Another Aspect of the Privacy Definitions

To show the gap between ZK-privacy and SIM-privacy, we change ZK-privacy as a game transformation technique. For simplicity, we consider weak ZK-privacy and wide-weak SIM-privacy.

First, we consider a slight variant of weak ZK-privacy such that the adversary can anonymously access to any tags in $\mathcal{C}$ in Phase 2. This is done by a slight modification for the intermediate algorithm $\mathcal{I}$. When the adversary outputs $\mathcal{C}$, each tag in $\mathcal{C}$ is randomized and indexed by the challenger. The challenger keeps the list $\{(i, \mathrm{ID}_j)\}_{i,j}$

[3] If an RFID authentication protocol specifies that the secret key of each tag is initially correlated and always updated, the adversary can obtain the challenge tag's secret key in Phase 1 of the IND-privacy game. However, this protocol can hold wide-forward SIM-privacy due to the key update algorithm.

where $i \in \{1, \ldots, |\mathcal{C}|\}$ and $\mathrm{ID}_j \in \mathcal{C}$ which is initially empty. When the adversary issues $\mathsf{SendTag}$ query to $\mathcal{I}$ with input (i, m), the challenger checks the list. If the index i is not contained in the list, new identity ID in $\mathcal{C}$ is uniformly chosen and the tuple (i, ID) is inserted in the list. The message is sent to the corresponding identity and its response is returned to the adversary. This is quite natural extension for ZK-privacy but we remark that this modification partially interpolates $\mathsf{DrawTag}$ query in ZK privacy to allow anonymous access. We call the modified privacy as ZK′-privacy. Consider that $\mathcal{O}' := \{\mathsf{Launch}, \mathsf{SendTag}, \mathsf{SendReader}, \mathsf{Result}\}$ and it is described as follows:

$\underline{\mathsf{Exp}^{\mathsf{ZK'\text{-}0}}_{\Pi,\mathcal{A},\mathcal{D}}(k)}$	$\underline{\mathsf{Exp}^{\mathsf{ZK'\text{-}1}}_{\Pi,\mathcal{S},\mathcal{D}}(k)}$
$(pk, sk) \overset{\mathsf{R}}{\leftarrow} \mathsf{Setup}(1^k);$	$(pk, sk) \overset{\mathsf{R}}{\leftarrow} \mathsf{Setup}(1^k);$
$(\mathcal{C}, st_1) \overset{\mathsf{R}}{\leftarrow} \mathcal{A}_1^{\mathcal{O}'}(pk, \mathcal{R}, \mathcal{T});$	$(\mathcal{C}, st_1) \overset{\mathsf{R}}{\leftarrow} \mathcal{S}_1^{\mathcal{O}'}(pk, \mathcal{R}, \mathcal{T});$
$\mathcal{T}' := \mathcal{T} \setminus \mathcal{C};$	$\mathcal{T}' := \mathcal{T} \setminus \mathcal{C};$
$view_{\mathcal{A}} \overset{\mathsf{R}}{\leftarrow} \mathcal{A}_2^{\mathcal{O}'}(\mathcal{R}, \mathcal{T}', \mathcal{I}(\mathcal{C}), st_1);$	$view_{\mathcal{S}} \overset{\mathsf{R}}{\leftarrow} \mathcal{S}_2^{\mathcal{O}'}(\mathcal{R}, \mathcal{T}', st_1);$
$b \overset{\mathsf{R}}{\leftarrow} \mathcal{D}(\mathcal{C}, \{i, \mathrm{ID}_j\}_{i,j}, view_{\mathcal{A}})$:	$b \overset{\mathsf{R}}{\leftarrow} \mathcal{D}(\mathcal{C}, \{i, \mathrm{ID}_j\}_{i,j}, view_{\mathcal{S}})$:
Output b	Output b

In this privacy model, the advantage of the adversary is defined by $\mathsf{Adv}^{\mathsf{ZK'}}_{\Pi,\mathcal{A},\mathcal{S},\mathcal{D}}(k) = |\Pr[\mathsf{Exp}^{\mathsf{ZK'\text{-}0}}_{\Pi,\mathcal{A},\mathcal{D}}(k) \to 1] - \Pr[\mathsf{Exp}^{\mathsf{ZK'\text{-}1}}_{\Pi,\mathcal{S},\mathcal{D}}(k) \to 1]|$.

Definition 4. *An RFID authentication protocol Π satisfies the ZK′-privacy if for any probabilistic polynomial time adversary $\mathcal{A}$, there exists an probabilistic polynomial time algorithm $\mathcal{S}$, for any probabilistic polynomial time distinguisher $\mathcal{D}$, $\mathsf{Adv}^{ZK'}_{\Pi,\mathcal{A},\mathcal{S},\mathcal{D}}(k)$ is negligible.*

Theorem 2. *ZK′-privacy is equivalent privacy notion to ZK-privacy.*

Proof. It is clear that ZK′-privacy implies ZK-privacy. We prove that if an RFID authentication protocol Π satisfies ZK-privacy, Π is also ZK′-privacy. This proof follows from the standard hybrid argument. Assume that the adversary against ZK′-privacy issues $\mathsf{SendTag}$ query at most q_s. Based on the ZK′-0 experiment, we change the output from $\mathsf{SendTag}$ query in Phase 2. The response is simulated by $\mathcal{S}$ for ZK-privacy until j-th invocation and executed by real tag after j-th invocation. When the adversary issues j-th $\mathsf{SendTag}$ query, the challenger flips a coin $b \overset{\mathsf{U}}{\leftarrow} \{0, 1\}$. If $b = 1$, the challenger activates

the real tag, and otherwise it runs a simulator to output the response. The difference between $b = 1$ and $b = 0$ is clearly bounded by $\mathsf{Adv}^{\mathsf{ZK}}_{\Pi,\mathcal{A},\mathcal{S},\mathcal{D}}(k)$. For $0 \leq j \leq q_s$, we can apply the same argument and finally we obtain an experiment which is identical to the ZK′-1 experiment. Therefore we have $\mathsf{Adv}^{\mathsf{ZK}'}_{\Pi,\mathcal{A},\mathcal{S},\mathcal{D}}(k) \leq q_s \cdot \mathsf{Adv}^{\mathsf{ZK}}_{\Pi,\mathcal{A},\mathcal{S},\mathcal{D}}(k)$. □

Now, recall the simulation strategy in Lemma 1. The simulator $\mathcal{S}$ chooses an arbitrary tag to simulate the anonymous access for the adversary if the RFID authentication holds IND-privacy. ZK′-privacy implies that the simulator can simulate the message between the reader and all tags in $\mathcal{C}$ without any communication with these tags. Even when particular tags are chosen by a distribution (i.e. DrawTag query in SIM-privacy), the tag's behavior is indistinguishable from the another tag and simulated by the simulator. Therefore, if the RFID authentication protocol satisfies ZK′-privacy (= IND-privacy), any specific information that corresponds to tag's secret is not revealed.

However, when we compare this privacy with wide-weak SIM-privacy, we can prove the following theorem.

Theorem 3. *ZK′-privacy does not imply SIM-privacy.*

Proof. One of the main technical differences between these privacy definitions is whether the simulator can interact with the reader or not. We explicitly wrote that the simulator takes as input $\mathcal{R}$ and can issue SendReader query in ZK′-privacy. On the other hand, SIM-privacy requires that the simulator must simulate SendReader query along with SendTag query. This gap leads the fact that ZK′-privacy does not imply SIM-privacy.

Let Π be an RFID authentication protocol that satisfies ZK′-privacy. For simplicity, we assume that $(m_1, m_2, m_3, \ldots)$ is the communication message exchanged by the reader and a tag in this protocol. Let $f : \mathcal{X} \to \mathcal{Y}$ be a one-way function and consider the following RFID authentication protocol Π'.

Setup. The reader runs Π to obtain (pk, sk) and shares secret keys with each tag in some cases. Choose $x \stackrel{\mathsf{U}}{\leftarrow} \mathcal{X}$ and compute $y := f(x)$. The reader publishes $pk' := (pk, f, y)$ and holds x as special secret key of the reader in Π'.

Authentication. The authentication is executed as follows:

1. The reader obtains m_1 from Π and sends it to the tag.
2. When the tag receives the message, it generates m_2 with Π and responds $m_2' := 0\|m_2$ to the reader.
3. When the reader receives the message m_2', it is parsed as $b\|m_2$. If $b = 0$, the reader generates m_3 and sends it to the tag (this is same as the honest execution of Π). If $b = 1$, the reader outputs x as the third message.

It is clear that the above RFID authentication protocol Π' satisfies ZK′-privacy. The simulator can issue $\mathsf{SendReader}$ query to obtain x when the adversary sends $0\|m_2$ to the reader. The other messages are trivially simulated by the assumption that Π is ZK′-privacy.

On the other hand, we can show that Π' does not meet SIM-privacy. The SIM adversary $\mathcal{A}$ launches the reader and sends $0\|m_2$ to the reader to obtain x. $\mathcal{A}$ sets $b := 1$ if and only if $y = f(x)$ and terminates the experiment by outputting b. It is obvious that $\Pr[\mathsf{Exp}^{\mathsf{SIM\text{-}0}}_{\Pi,\mathcal{A}}(k) \to 1] = 1$. However, it is infeasible for any simulator to output x' such that $y = f(x')$ from the assumption that f is one-way function. Therefore we have $\Pr[\mathsf{Exp}^{\mathsf{SIM\text{-}1}}_{\Pi,\mathcal{A},\mathcal{S}}(k) \to 1] \leq \varepsilon$ for a negligible fraction ε. Thus we have $\mathsf{Adv}^{\mathsf{SIM}}_{\Pi,\mathcal{A},\mathcal{S}}(k)$ is not negligible. □

Remark that this kind of the separation example is originally described in Pass, Shelat and Vaikuntanathan to show the gap between their variants of non-malleability definition for public key encryption [17]. We think that it is interesting to show the gap between IND-privacy and SIM-privacy based on the same idea. As a result, SIM-privacy covers reader's information leakage though IND-privacy and ZK-privacy only consider the tag's privacy.

5 Conclusion

We analyzed the three privacy models for RFID authentication protocol. Contrary to the discussion described in Deng et al. [7], we showed that IND-privacy is equivalent to ZK-privacy. Furthermore, we provided a polynomially equivalent variant of the ZK-privacy to consider the relation between SIM-privacy and obtained the result that ZK-privacy does not imply SIM-privacy since SIM-privacy covers the reader's privacy in addition to the tag's privacy.

References

1. Avoine, G.: Adversarial model for radio frequency identification. ePrint Archive, 2006/049 (2005)
2. Bellare, M., Desai, A., Pointcheval, D., Rogaway, P.: Relations among notions of security for public-key encryption schemes. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 26–45. Springer Heidelberg (1998)
3. Bellare, M., Sahai, A.: Non-malleable encryption: Equivalence between two notions, and an indistinguishability-based characterization. CRYPTO 1999 1666 pp. 519–536 (1999)
4. Burmester, M., Le, T.V., Medeiros, B.D., Tsudik, G.: Universally composable rfid identification and authentication protocols. TISSEC 2009 12(4) pp. 21:1–33 (2009)
5. Burmester, M., van Le, T., de Medeiros, B.: Provably secure ubiquitous systems: Universally composable RFID authentication protocols. In: SecureComm 2006, pp. 1–9. IEEE (2006)
6. Canard, S., Coisel, I.: Data synchronization in privacy-preserving rfid authentication schemes. In: RFIDSec 2008. (2008)
7. Deng, R.H., Li, Y., Yung, M., Zhao, Y.: A new framework for RFID privacy. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) ESORICS 2010. LNCS, vol. 6345, pp. 1–18. Springer Heidelberg (2010)
8. Ha, J., Moon, S., Zhou, J., Ha, J.: A new formal proof model for RFID location privacy. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 267–281. Springer Heidelberg (2008)
9. Hermans, J., Pashalidis, A., Vercauteren, F., Preneel, B.: A new rfid privacy model. In: Atluri, V., Diaz, C. (eds.) ESORICS 2011. LNCS, vol. 6879, pp. 568–587. Springer Heidelberg (2011)
10. Juels, A., Weis, S.A.: Defining strong privacy for RFID. In: PerCom 2007, pp. 342–347. IEEE (2007)
11. Juels, A., Weis, S.A.: Defining strong privacy for RFID. ACM Transactions on Information and System Security 13(1) (2009)
12. Lai, J., Deng, R.H., Li, Y.: Revisiting unpredictability-based RFID privacy. In: Zhou, J., Yung, M. (eds.) ACNS 2010. LNCS, vol. 6123, pp. 475–492. Springer Heidelberg (2010)
13. Le, T.V., Burmester, M., de Medeiros, B.: Universally composable and forward-secure rfid authentication and authenticated key exchange. In: ASIACCS 2007. pp. 242–252. (2007)
14. Ma, C., Li, Y., Deng, R.H., Li, T.: RFID privacy: Relation between two notions, minimal condition, and efficient construction. In: ACMCCS 2009, pp. 54–65. ACM (2009)
15. Ouafi, K., Phan, R.C.W.: Traceable privacy of recent provably-secure RFID protocols. In: Bellovin, S.M., Gennaro, R., Keromytis, A.D., Yung, M. (eds.) ACNS 2008. LNCS, vol. 5037, pp. 479–489. Springer Heidelberg (2008)
16. Paise, R.I., Vaudenay, S.: Mutual authentication in RFID. In: ASIACCS 2008. pp. 292–299. (2008)
17. Pass, R., Shelat, A., Vaikuntanathan, V.: Relations among notions of non-malleability for encryption. In: Kurosawa, K. (ed.) ASIACRYPT 2007. LNCS, vol. 4833, pp. 519–535. Springer Heidelberg (2007)
18. Vaudenay, S.: On privacy models for RFID. In: Kurosawa, K. (ed.) ASIACRYPT 2007. LNCS, vol. 4833, pp. 68–87. Springer, Heidelberg (2007)
19. Vaudenay, S.: Privacy models for RFID schemes. In: Yalcin, S.O. (ed.) RFIDSec 2010. LNCS, vol. 6370 pp. 65. (2010)

CANAuth - A Simple, Backward Compatible Broadcast Authentication Protocol for CAN bus

Anthony Van Herrewege, Dave Singelee, Ingrid Verbauwhede
firstname.lastname@esat.kuleuven.be

***Abstract*— The Controller-Area Network (CAN) bus protocol [1] is a bus protocol invented in 1986 by Robert Bosch GmbH, originally intended for automotive use. By now, the bus can be found in devices ranging from cars and trucks, over lightning setups to industrial looms. Due to its nature, it is a system very much focused on safety, i.e., reliability. Unfortunately, there is no build-in way to enforce security, such as encryption or authentication.**

In this paper, we investigate the problems associated with implementing a backward compatible message authentication protocol on the CAN bus. We show which constraints such a protocol has to meet and why this eliminates, to the best of our knowledge, all the authentication protocols published so far.

Furthermore, we present a message authentication protocol, CANAuth, that meets all of the requirements set forth and does not violate any constraint of the CAN bus.

***Keywords*—CAN bus, embedded networks, broadcast authentication, symmetric cryptography**

I. Introduction

The Controller-Area Network (CAN) bus protocol [1] is a bus protocol invented in 1986 by Robert Bosch GmbH, originally intended for automotive use.

CAN bus nodes are all connected to the same, shared bus line. The CAN bus is wired such that a 0-signal is dominant over a recessive 1-signal. These dominant and recessive signals are used for a CSMA/CA scheme. An arbitration scheme using priority resolution is used to decide which node is allowed to transmit data over the bus. The lower a node its ID (and thus the more dominant 0s it sends during bus arbitration), the higher its priority. Using this scheme makes the CAN bus fit for use as a real-time communication bus.

During transmission, a node continuously checks the signal on the bus and compare this with the signal it is transmitting. If a mismatch occurs, an error is raised, except during arbitration, in which case the node will just stop transmitting. After each message, a CRC is transmitted and receiving nodes will raise an error in case a mismatch occurs. To prevent broken nodes from continuously disturbing and invalidating messages, internal error counters are kept which, depending on their value, disable the right of a node to raise errors. This guarantees that communication over the CAN bus is very reliable, since, even with broken nodes raising errors, messages will eventually be transmitted successfully.

One year after its introduction, Philips presented the first CAN bus controller and, in time, the protocol was used more and more in non-automotive machines. By now, the bus protocol can be found in a wide range of devices: from cars and trucks, over lightning setups to industrial looms, printers, freezers and trains.

Due to its nature, the design of CAN bus is very much focused on safety, i.e., reliability. Unfortunately, there is no build-in way to enforce security, such as encryption or authentication. This leads to many possible attacks, as demonstrated by Koscher et al. [2] and Checkoway et al. [3]. Examples of such attacks are controlling brakes, remotely starting the car and controlling the air conditioning.

In this paper, we investigate the possibility of implementing a backward compatible message authentication protocol on the CAN bus. We show which constraints such a protocol has to meet and why this eliminates, to the best of our knowledge, all the authentication protocols published so far.

In Section II, we very briefly show some of the work that is already been done on broadcast authentication protocols. Section III gives a detailed overview of the requirements such a protocol should meet and the constraints it should honor for it to be usable on the CAN bus. Then, in Section IV, we present our proposal for such an authentication protocol. In Section V, an adversarial model is defined and the security properties of the protocol are investigated. Finally, we present our conclusions in Section VI.

II. Previous Work

In the past, different protocols have been published on how to archieve authentication in broadcast networks, some of those even aimed at lightweight embedded networks. The next few paragraphs give a very concise overview of some of those protocols.

The TESLA protocols are a family of related lightweight authentication protocols, relying on delayed key disclosure to guarantee message authenticity. The original TESLA protocol was published by Perrig et al. in 2000 [4], [5]. This protocol is designed to provide authenticated broadcast capabilities. Subsequently, Perrig et al. presented μTESLA [6], a modification of the original TESLA protocol, aimed at sensor networks, with severe contraints placed on computation and storage capabilities.

Other protocols have been published for use in broadcast networks, such as a symmetric solution by Gennaro and

Rohatgi [7] and Rohatgi's improved protocol [8].

Unfortunately, all of these protocols have certain characteristics which violate the constraints set in Section III, such as requiring challenge-response communication, introducing delays (the main problem with TESLA and μTESLA) or needing to transmit large amounts of data (the main problem of the solutions by Gennaro and Rohatgi), all of which are not acceptable in a CAN bus network.

III. Problem Overview

In this section we explain the requirements and constraints for a backward compatible, lightweight message authentication protocol on CAN bus. This should make it clear why none of the published protocols so far are usable on the CAN bus. Furthermore, we show how we will we meet those requirements and constraints with our proposed CANAuth protocol.

A. Authentication Protocol Requirements

The basic requirements a message authentication protocol should provide are:

Message authentication
: The authenticity of messages should be provable. The exact origin of the message is not important as long as it is guaranteed that it was sent by a trusted node.

Replay attack resistance
: Replaying previously sent authenticated messages should lead to those messages being discarded by the verifying nodes.

Group keys
: It should be possible for a group of related messages to be authenticated with the same key[1]. This reduces the size of key storage needed.

Backward compatibility
: A node employing the authentication protocol has to be able to authenticate its messages without disturbing the workings of any incompatible nodes. It should be possible to connect a number of nodes supporting the authentication protocol to an existing bus without having to do any reconfiguration of the existing nodes.

B. Restrictions Due To CANBus

Although many authentication protocols exist, the CAN bus presents a few peculiar problems. Due to the fact that we want a backward compatible authentication protocol, following restrictions have to be taken into account:

Hard real-time
: CAN bus systems are often employed in environments where hard real-time constraints apply, e.g., cars. Therefore, message transmission and processing times should not be significantly influenced by the authentication protocol. All authentication data needs to be send along with its message, so as not to disturb the real-time response capabilities of the system.

Message length
: A single message on the CAN bus can contain between 1 and 8 bytes of data. Any extra authenticated data needed by the authentication protocol somehow has to be transmitted along with the message it belongs to.

Message IDs
: Each type of message on the CAN bus is associated with a certain ID, e.g. ID 1 = temperature of location L_1, ID 2 = humidity of location L_1, ... An ID consists of 11 bits[2] and due to the fact that an ID is coupled to a very specific message content, most of the IDs will already be taken in an existing network. This means one can not go around adding extra IDs with some new content type.

Unidirectional communication
: A CAN interface sending messages has no ID for itself and all communication is broadcast. Due to this there is no notion of bi-directional communication between nodes, since CAN interfaces have no ID. The only bi-directional "communication" possible between multiple nodes is though an error flag. Even then, a transmitting node can not find out which receiving node raised the error.

The first requirement, *Message authentication*, can be met by attaching a message authentication code (MAC) to a message. Due to the *Hard real-time* restriction, the employed MAC algorithm needs to be fast. Ideally, it should be possible to start processing for verification as soon as part of the authentication data is received. One algorithm satisfying these requirements is HMAC [9], [10], assuming the hash function employed therein is fast.

The *Replay attack resistance* requirement can then be met by incorporating some counter value in the MAC calculation. The same counter value should never be reused however, which could lead to problems if a counter of limited length is used and a large amount of authenticated messages are sent. Furthermore, the less state that has to be saved across subsequent system restarts, the better. A solution for this is the establishment of a session-key during system startup. A limited length counter can then be made to work in our benefit, since it allows the system to gauge when a new session-key should be established.

Due to the restriction on *Message IDs* and *Unidirectional communication*, nodes can not send a message back in response to some message they receive. That would require the creation of a large amount of new IDs, each with a specific new content type such as ID 5 = response of node $\mathcal{N}_i$ to messages with ID 1. Thus, if n nodes want to be

[1] Note that this is slightly different from the usual use of the term *Group key*, which normally implies a key shared between a group of users. In this case the key is shared between a group of messages.

[2] When using extended CAN, IDs are 29 bits long.

able to respond to m different message IDs, $n \cdot m$ IDs have to be created. Given the large number of different message content types, that is not possible as it would quickly lead to exhaustion of all the available 11 bit IDs. Furthermore, it would require reconfiguration of existing nodes on a bus each time a new node is added. In case bi-directional communication should prove necessary, a (backward compatible) workaround will need to be devised.

Supporting the *Group keys* requirement is straightforward using options made available by the CAN protocol. In general, one can program a CAN interface with a number of *acceptance codes.* These codes tell the interface to listen to messages with an ID matching any of the *acceptance codes.* It is also possible to specify *acceptance masks*, allowing a single *acceptance code* to match multiple IDs. By linking keys to these $\{acceptance\ code, acceptance\ mask\}$ sets, a *group key* setup can be achieved. We define a group of related messages $\mathcal{G}_i$ as the collection of all messages with IDs matching the pair $\{acceptance\ code, acceptance\ mask\}_i$.

The last requirement, *Backward compatibility* presents the biggest problem. For one, attaching authentication data by concatenating it with an existing CAN message is impossible, since that will violate the *Message length* constraint. Neither can the authentication data be put inside a CAN message, since this will decrease the already very limited maximum message size. A third option would be to sent one long data packet over multiple messages. However, this would reduce the real-time capabilities of the system.

C. Transmission with CAN+

One solution to all this is sending the authentication data through an out-of-band channel. This can be achieved with the CAN+ protocol [11]. Using this protocol, extra bits can be inserted in between the sampling points of a CAN bus interface. A graphical presentation of how this works, is shown in Fig. 1.

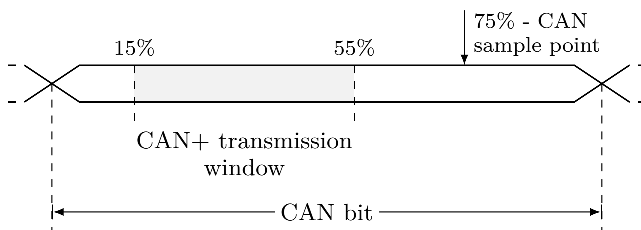


Fig. 1. Timeframe during which the CAN+ protocol can insert extra data into a CAN bit, without disrupting the working of regular CAN bus controllers.

The number of data bits that can be transmitted this way is limited by the maximum attainable clock speed of the CAN+ interface. Ziermann et al. report that on a 1 MHz CAN network, they can transmit 15 CAN+ bytes for each CAN byte, using an FPGA running at 300 MHz. At lower CAN bus speeds, this number increases linearly as:

$$\frac{\text{CAN+ data bits}}{\text{CAN data bit}} = \frac{1\text{ MHz}}{f_{\text{bus}}} \cdot 16 - 1.$$

The loss of one CAN+ data bit is due to the need to send a start bit at the beginning of every CAN+ transmission. On a 100 kHz CAN network, it is thus possible to transmit up to 159 CAN+ bits for each CAN bit.

Since it should be possible to authenticate a message irrespective of its length or the bus speed of the CAN bus network, authentication data should be limited to 15 bytes. This allows one to send all necessary authentication data in the worst case scenario, i.e. as part of a 1 byte CAN message on a 1 MHz CAN bus. Due to this restriction on authentication data length, the use of public-key (and identity-based) cryptography is not possible, due to its large key size requirement compared to symmetric cryptography.

The restrictions and requirements an authentication protocol has to adhere to on the CAN bus make any published protocols, to the best of our knowledge, unusable.

D. Problems with multi-node transmissions

One could devise a system whereby multiple nodes transmit during the CAN+ timeframe, which would then allow bi-directional communication. This would violate the hard real-time constraint as soon as a certain number of nodes needed to sent data, since one would still need multiple messages to do that. During a key establishment, such a breach of constraints could be disregarded though. However, in the next paragraph, we prove that any protocol that requires bi-directional communication between nodes, will impose a speed limit on the CAN bus.

The CAN bus protocol uses a Carrier Sense Multiple Access with Collision Detection (CSMA/CA) protocol during transmission, whereby bit collisions are to be detected within a bit its transmission window. Assume one wanted to sent a minimum of one data bit with the CAN+ protocol per CAN bit. Each CAN+ transmission begins with a start bit, so there would have to be two bit transmissions during the CAN+ timeframe. This timeframe occupies maximum 40% of a CAN bus bit transmission timeframe. Thus, in this case, the CAN+ protocol would need to work at a frequency at least $2 \cdot \frac{1}{0.40} = 5$ times faster than the CAN protocol.

The CAN bus standard guarantees that for a bus length of a maximum of 30 meter, CSMA/CA, and thus the CAN bus protocol, can work at a maximum bus speed of 1 MHz. However, since we want CSMA/CA to function during the CAN+ transmission, which needs to work at a frequency at least 5 times as fast as the CAN bus, the maximum CAN bus speed for the given bus length is only 200 kHz.

Furthermore, at this rate, of the 16 $\text{CAN}+$ bits sent during a CAN byte transmission, only 8 $\text{CAN}+$ bits are usable for data, since the other 8 are needed as start bits. The more data bits one wishes to sent during the CAN+ transmission window, the lower the CAN bus speed needs

to be if one wishes to adhere to the maximum speed of 1 MHz for CAN+.

Of course, it is possible to increase the speed of CAN+ in our example to 5 MHz, in which case CAN can run at 1 MHz. However, if so, the maximum bus length would be reduced as well, to allow for timely collision detection during the CAN+ transmissions.

Thus, if one requires bi-directional communication with the help of CAN+, either the maximum bus length or the CAN bus speed needs to decrease.

IV. CANAuth Protocol

In this section we propose a simple authentication protocol, CANAuth, that meets all of the requirements set forth in Section III and does not violate any of the constraints.

A. Data Transmission and Frame Format

As suggested in Section III, authentication data is transmitted out-of-band with the CAN+ [11] protocol, which gives us a maximum length of 15 bytes for the authentication message. The bytes are subdivided in fields as shown in Fig. 2.

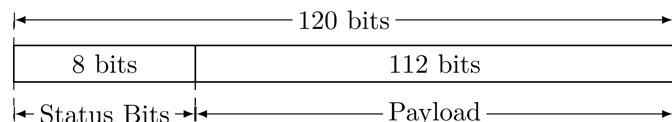


Fig. 2. The CANAuth data frame fields. The data frame consists of the first 15 CAN+ bytes (= 120 CAN+ bits) of a CAN message.

The first eight bits are used as status flags. Currently, only the first two of these bits are used, with the remaining six available to be used in future versions of the protocol. The remaining 112 bits are used to either transmit key establishment or signature data. In the following subsections, more information is given about these two possibilities.

B. Key Establishment

The key establishment protocol described here requires one or more pre-shared 128 bit keys to be available on each CANAuth node. Each group of related messages $\mathcal{G}i$ should get a pre-shared key $\mathcal{K}p_i$ assigned during development of the CAN bus design[3]. We assume that the keys are stored in some tamper-proof storage and are unable to be queried by anyone but the node itself.

The node responsible for transmitting messages $\mathcal{G}i$ is to set up the key for that group. In case multiple nodes transmit messages $\mathcal{G}i$, the key established by the node transmitting with the lowest ID for that group gains precedence.

[3]Even better would be if all 'devices' of the same type (i.e. all Brand X Model Y cars) all contained different keys, programmed during production. This to prevent one $\mathcal{K}p_i$ being found from leading to a security breach of all devices of the same type.

To prevent replay attacks, key establishment works in two phases. First, the appropriate node (as per the rule defined above) transmits a message on the CAN bus with an attached CAN+ message of the form shown in Fig. 3.

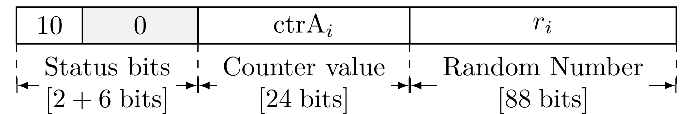


Fig. 3. The CANAuth data frame fields during the first part of key establishment.

The first bit of the frame is set to one to signal that key establishment is taking place. The second bit is set to zero, signalling that this message is the first of two needed for key establishment. The next six bits are unused and should be set to zero. The actual payload of the message is a 24 bit counter and an 88 bit random number.

With the counter value ctrA_i and the random number r_i all nodes possessing the pre-shared key $\mathcal{K}p_i$ generate the session key $\mathcal{K}s_i$ for messages $\mathcal{G}_i$ using HMAC [9]. Implementations of CANAuth are free to use whichever hashing algorithm for HMAC that is deemed strong enough. Depending on the hashing algorithm used, the HMAC output will be longer than 128 bits. The session key $\mathcal{K}s_i$ consists of the 128 LSB of the HMAC output and is generated as follows:

$$\mathcal{K}s_i = \mathrm{HMAC}(\mathcal{K}p_i, \mathrm{ctrA}_i \parallel r_i) \bmod 2^{128}. \tag{1}$$

The addition of a counter in the message prevents an adversary from isolating the trusted key establishment node from the bus and setting up a session key with the other nodes on the bus, using a previously transmitted random number. Even though the adversary would not know the session key $\mathcal{K}s_i$, he could then transmit recorded authenticated messages. Thus, to prevent this attack, before any node accepts the session key $\mathcal{K}s_i$, it should verify that the random number did in fact originate from a trusted node and that it has not been used before.

There are two checks to guarantee this. First, each node on the bus should store the last validated counter value ctrA_i for each $\mathcal{G}_i$ in non-volatile memory. New key establishment messages are only accepted when the transmitted counter value is higher than the stored value. Second, as the next step in the key establishment, the key establishment node transmits a second message, using the format shown in Fig. 4.

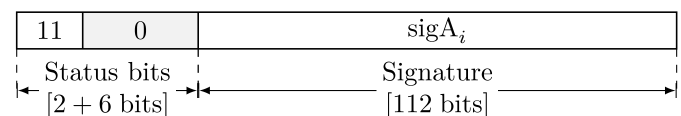


Fig. 4. The CANAuth data frame fields during the second part of key establishment.

In this case, the first and second bits are both set to one, to signal that this is a message to authenticate the key establishment. The next six bits are again unused and should be set to zero. Following that is a 112 bit signature generated as follows:

$$\mathrm{sigA}_i = \mathrm{HMAC}(\mathcal{K}s_i, \mathrm{ctrA}_i \parallel r_i) \bmod 2^{112}. \quad (2)$$

All nodes participating in the establishment protocol can now verify that the sending node knows the session key and is thus trustworthy and that ctrA_i and r_i where not tampered with. These nodes, including the transmitting node, set their stored value ctrA_i to whatever value was transmitted in the first key establishment message.

An adversary could still isolate the key establishment node from the bus and get valid pairs of key establishment messages, but to keep the machine in working condition, will have to forward those messages to the rest of the bus anyway, due to reasons we explain later.

The addition of the counter to the key establishment protocol does not only have benefits though. Once the counter is at its maximum value, listening nodes will not accept new session keys anymore, since the transmitted counter value can not be higher than the stored counter value. This limits the maximum lifetime of any machine utilizing the CANAuth protocol. That is why we propose the use of a 24 bit counter field, which allows $2^{24} = 16\,777\,216$ session key setups, enough to set up a new session key once every minute for ± 32 years.

In case an error is raised in response to any of the two key establishment messages, the key establishment node will restart the protocol from the first message.

C. Message Authentication

Once a session key has been established, messages $\mathcal{G}_i$ can be authenticated. The frame format for authentication data is shown in Fig. 5.

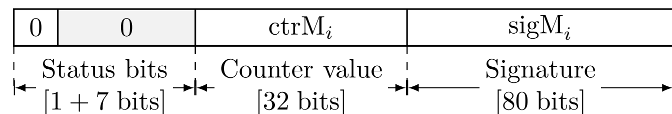


Fig. 5. The CANAuth data frame fields during message authentication.

The counter value ctrM_i is there to prevent replay attacks. Nodes should keep a local copy of the value of ctrM_i encountered in the last valid message. Authenticated message should only be accepted when ctrM_i is greater than the stored value. Due to this rule, the session key $\mathcal{K}s_i$ needs to be renewed each time ctrM_i is about to wrap around back to 0. For every message $\mathcal{G}_i$ a node transmits, ctrM_i should be increased by at least 1.

The second part of the authentication data frame is the signature, which is obtained by taking the 80 LSB of the HMAC of the counter value ctrM_i and the CAN message data msg_i under the session key $\mathcal{K}s_i$, i.e.:

$$\mathrm{sigM}_i = \mathrm{HMAC}(\mathcal{K}s_i, \mathrm{ctrM}_i \parallel \mathrm{msg}_i) \bmod 2^{80}. \quad (3)$$

D. Handling of Invalid Authentications

On a CAN bus system, every node can transmit an error frame at any time, which invalidates the message that is being transmitted. Nodes keep track of the number of errors on the bus. A node that is receiving data will increase its receiving error counter (REC) by 1 for each erroneous message, except if it signaled the error itself, in which case its REC is increased by 8. For every successfully received message, the REC is decreased by 1.

To prevent a faulty node from invalidating every message on the bus, nodes are allowed to signal errors as long as their REC is below 127. Once a node its REC value is higher, it has to abstain from signalling errors until enough messages have been successfully received to decrease its REC below 127.

CANAuth leverages this error handling capability by utilizing the counters in much the same way. Whenever a node can not successfully validate a message, CANAuth will make that node raise an error and increase its REC by 8. Whenever an error is raised, every CANAuth compatible node should discard the corresponding message, even if a node does not detect an error and can successfully verify the attached signature.

E. A Note on Hardware Implementation Speed

The latest a message can be rejected on the CAN bus is by signalling an error frame after the transmission of the ACK delimiter bit. The end of the ACK delimiter bit is 3 CAN bit lengths after the last CRC bit has been send. Thus, to be able to reject messages with invalid CANAuth data, the CANAuth controller should be done verifying the signature by then.

The transmission speed of CAN+, over which CANAuth data is transmitted, should be at least 40 times that of the CAN bus[4]. Since the CAN+ bits need to be sampled, the internal clock of a CAN+ controller will need to be at least twice as fast as that, so for every transmission of a CAN bit, there are 80 clock cycles on the CAN+ controller.

For a CAN message of 1 byte, there are 18 CAN bit lengths between the transmission of the last CAN data bit and the transmission of the ACK delimiter bit. Furthermore, the last CANAuth bit is transmitted after 55% of the duration of a CAN bit transmission [11]. Thus, there are at least $(18+0.45)\cdot 80 = 1\,476$ clock cycles available for the CANAuth implementation to calculate and verify the HMAC signature, assuming a CAN message with 1 byte of data and 16 CAN + bits per CAN bit. That should be

[4]CAN+ can only utilize a maximum of 40% of the duration of a CAN bit for transmission. During this time, for CANAuth to work, at least 16 bits need to be transmitted, thus $f_{\mathrm{CAN+}} = \frac{16}{0.4} \cdot f_{\mathrm{CAN}} = 40 \cdot f_{\mathrm{CAN}}$.

more than sufficient for any hardware implementation of a hash function to calculate its result.

V. Security Properties

The requirements and workings of the CANAuth authentication protocol have been explained. In this section, we define a simple model for an adversary $\mathcal{A}$ upon which we can base ourselves when discussing the security of our authentication protocol. Furthermore, we show how certain properties of the CAN bus make attacks a lot harder.

A. Adversarial Model

The adversary $\mathcal{A}$ has access to all the data that is transmitted over the CAN bus. Furthermore, $\mathcal{A}$ has physical access to the bus and the nodes. Thus, $\mathcal{A}$ is capable of setting up a Man-in-the-Middle (MitM) attack. Although $\mathcal{A}$ has access to the nodes, it reasonable to assume $\mathcal{A}$ does not possess the technology to set up invasive attacks on the nodes, thereby allowing access to e.g. tamper-proof storage.

We furthermore assume that all pre-shared and session keys are stored in tamper-proof to and from which only the node can write and read. Counter values should be stored in tamper-proof memory as well, but are not required to be shielded from reading by an outsider, since their values are public anyway.

B. Denial-of-Service Attacks

One important class of attacks are Denial-of-Service (DoS) attacks. It is trivial to invent methods to set up a DoS attack on the proposed protocol. Since such attacks will always somehow involve disturbing the signal on the CAN bus, it would be just as simple, if not simpler, for $\mathcal{A}$ to connect the bus to ground at strategic times to create an error condition. Since it is impossible to defend against such an attack, we will ignore any kind of attack that can be reduced to a simple DoS attack.

An example of one such attack, goes as follows. We have a CAN bus with two nodes, $\mathcal{N}_a$ and $\mathcal{N}_b$, which both know a pre-shared key $\mathcal{K}p_1$. During the first key establishment message, $\mathcal{A}$ executes a straightforward MitM attack:

$$\mathcal{N}_a \quad \xrightarrow{r_1} \quad \mathcal{A} \quad \xrightarrow{r_1'} \quad \mathcal{N}_b.$$

The result is that nodes $\mathcal{N}_a$ and $\mathcal{N}_b$ generate different session keys, and thus $\mathcal{N}_b$ will find the key establishment authentication message invalid. Thus, $\mathcal{N}_a$ will restart the key establishment. The attacker $\mathcal{A}$ learns nothing however and could have achieved exactly the same result by raising an error frame when the key establishment authentication message appeared on the bus, hence this attack reduces to a DoS attack.

C. Protocol Security = HMAC Security

We will now argue that the security of our protocol reduces to the security of the underlying HMAC algorithm. An adversary $\mathcal{A}$ is considered capable of breaking the protocol when he can forge authentication signatures for messages msg_i with probability $p > \epsilon$. Since signature content is entirely generated with HMAC, $\mathcal{A}$ can forge signatures sig_i' iif, given a message msg_i and a counter value ctr_i,

$$P\left(\text{sig}_i' = \text{sig}_i\right) > \epsilon.$$

Being able to forge signatures thus implies that $\mathcal{A}$ has knowledge of one of the following:

1) pre-shared key $\mathcal{K}p_i$
2) session key $\mathcal{K}s_i$
3) a key $\mathcal{K}s_i'$ for which

$$\text{HMAC}(\mathcal{K}s_i', i \parallel \text{ctr}_i \parallel \text{msg}_i) \bmod 2^{80} = \text{sig}_i.$$

Since invasive readings of node content is not possible in our adversarial model, option one is ruled out, since that would require invasive access to a node $\mathcal{N}$ storing $\mathcal{K}p_i$. The only way option two is possible then is if $\mathcal{A}$ manages to break HMAC. Finally, the third option requires $\mathcal{A}$ to find a collision on HMAC. Thus, the security of CANAuth depends entirely on the security of HMAC.

D. Tamper Resistance

The fact that a trusted node only increases its key establishment counter ctrA_i after a valid key establishment provides certain security benefits. If instead ctrA_i was increased after every unsuccessful key establishment attempt, $\mathcal{A}$ could easily mount a DoS attack by constantly raising errors, thereby rapidly increasing the counter to its maximum value and preventing any further key establishment. One could argue that this is not important, since our adversarial model ignores DoS attacks. However, a more important advantage of not increasing the counter ctrA_i is that it leaves the machine in which the CANAuth protocol is implemented in a non-functional state should $\mathcal{A}$ manage to establish a session key. Thus, this quality can work as a deterent against attacks: either $\mathcal{A}$ does not attempt any attacks and his machine stays functional or he has to completely break the protocol by finding $\mathcal{K}p_i$ to be able to keep his machine functional.

Assume $\mathcal{A}$ manages to find a valid tuple $\{\text{ctrA}_i, r_i, \mathcal{K}s_i\}$, but not $\mathcal{K}p_i$[5]. Due to the checks during the key establishment, ctrA_i needs to be higher than the last used authenticated counter value. Thus, if $\mathcal{A}$ uses the valid tuple, after the next machine reboot or when the message authentication counter reaches its maximum, the protocol will not accept any more valid key establishment messages from a trusted node, since the trusted node will transmit a key establishment counter value ctrA_i' equal to or lower than the ctrA_i value used by $\mathcal{A}$. Nodes will thus reject all (valid) messages from the trusted node. This means that if $\mathcal{A}$ ever uses such a valid tuple, the machine will be left in a non-functional state.

[5]Knowledge of $\mathcal{K}p_i$ would make this attack obsolete, since $\mathcal{A}$ can then gain full control over the messages $\mathcal{G}_i$ by either sending authenticated messages $\mathcal{G}_i$ himself or executing MitM attacks.

Note that this non-functionality can never happen during normal operation, since nodes only increase their key establishment counter value ctrA_i after a valid key establishment phase and only one node is allowed to set up session keys $\mathcal{K}s_i$ for $\mathcal{G}_i$, so counter values are always synchronized between verifying and signing nodes.

E. Limitations on On-line Attack Speed

Due to the nature of the CAN bus, message throughput on the bus is very modest at best. A table showing the time it takes to transmit messages is shown in Table I. These times are calculated assuming there are zero errors on the bus, an average number of stuffing bits[6] and include the necessary 3 bit interframe space after each message.

TABLE I
LIST OF THEORETICAL MINIMUM TRANSMISSION TIMES FOR n MESSAGES OF LENGTH l ON AN s BAUD CAN BUS. AN AVERAGE NUMBER OF STUFFING BITS WAS ACCOUNTED FOR WHEN CALCULATING THESE TIMES [1].

Baud rate s & message length l	Number of messages n: 2^{16}	2^{32}	2^{64}
100 Kbps			
1 byte	39 s	29 d	3.4×10^8 y
4 bytes	54 s	2 mo	4.8×10^8 y
8 bytes	2 m	2 mo	6.7×10^8 y
500 Kbps			
1 byte	8 s	6 d	6.8×10^7 y
4 bytes	11 s	9 d	9.6×10^7 y
8 bytes	15 s	12 d	1.3×10^8 y
1 Mbps			
1 byte	4 s	3 d	3.4×10^7 y
4 bytes	6 s	5 d	4.8×10^7 y
8 bytes	8 s	6 d	6.7×10^7 y

As can be deduced from Table I, on-line attacks on the system, using honest nodes as oracles to verify a key guess, would be extremely slow. Furthermore, every wrong key guess leads to an error message on the bus, which in turn increases the *REC* of each node. If these nodes their *REC* exceeds 127, they will not signal errors anymore and can thus not be used as verification oracles. Thus, for every wrong key guess a valid message needs to appear on the bus, so that the honest nodes their *REC* does not exceed 127. So in reality, the time it would take to transmit a certain number of guesses would take at least twice as long as the times given in Table I.

VI. CONCLUSION

In this paper, we gave an overview of the problems associated with message authentication protocols on the CAN bus. Furthermore, we presented CANAuth, a simple, lightweight message authentication protocol based on HMAC for use on the CAN bus. Due to its backward compatibility, CANAuth-compatible nodes can work on a CAN bus without any modification to existing nodes. The security of the proposed protocol depends entirely on the security of the employed HMAC variant. Furthermore, due to the nature of the CAN bus and the design of CANAuth, adversaries executing on-line attacks against the protocol can only do this at very slow speeds and are severely hindered should they be able to establish a session key, further enhancing security.

One of the major drawbacks of the proposed protocol is that all nodes that must be able to verify messages $\mathcal{G}_i$ need to know the pre-shared key $\mathcal{K}p_i$. Thus, as future work, it would be interesting to research if there is any possibility of using some kind of asymmetric primitives whilst maintaining a sufficiently high level of security.

[6] There can only be a maximum of five bits of equal value in sequence on the CAN bus. Whenever such a series of five equal bits is encountered, the CAN controller inserts a stuffing bit of inverse value [1].

ACKNOWLEDGMENTS

This work was supported in part the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), by K.U.Leuven-BOF (OT/06/40), by FWO grant G.0300.07, and by the European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II.

REFERENCES

[1] W. Voss, *A Comprehensible Guide to Controller Area Network*. Massachusetts, USA: Copperhill Media Corporation, 2005.
[2] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, “Experimental Security Analysis of a Modern Automobile,” in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2010, pp. 447–462.
[3] S. Checkoway, D. McCoy, D. Anderson, B. Kantor, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno, “Comprehensive Experimental Analysis of Automototive Attack Surfaces,” in *Proceedings of the USENIX Security Symposium*, San Francisco, CA, August 2011.
[4] A. Perrig, R. Canetti, J. D. Tygar, and D. X. Song, “Efficient Authentication and Signing of Multicast Streams over Lossy Channels,” in *IEEE Symposium on Security and Privacy*, 2000, pp. 56–73.
[5] A. Perrig, R. Canetti, D. X. Song, and J. D. Tygar, “Efficient and Secure Source Authentication for Multicast,” in *NDSS*. The Internet Society, 2001.
[6] A. Perrig, R. Szewczyk, V. Wen, D. E. Culler, and J. D. Tygar, “SPINS: security protocols for sensor networks,” in *MOBICOM*, 2001, pp. 189–199.
[7] R. Gennaro and P. Rohatgi, “How to sign digital streams,” in *CRYPTO*, ser. Lecture Notes in Computer Science, B. S. K. Jr., Ed., vol. 1294. Springer, 1997, pp. 180–197.
[8] P. Rohatgi, “A compact and fast hybrid signature scheme for multicast packet authentication,” in *ACM Conference on Computer and Communications Security*, 1999, pp. 93–100.
[9] H. Krawczyk, M. Bellare, and R. Canetti, “HMAC: Keyed-Hashing for Message Authentication,” RFC 2104 (Informational), Internet Engineering Task Force, February 1997. [Online]. Available: http://www.ietf.org/rfc/rfc2104.txt
[10] M. Bellare, R. Canetti, and H. Krawczyk, “Keying Hash Functions for Message Authentication,” in *CRYPTO*, ser. Lecture Notes in Computer Science, N. Koblitz, Ed., vol. 1109. Springer, 1996, pp. 1–15.
[11] T. Ziermann, S. Wildermann, and J. Teich, “CAN+: A new backward-compatible Controller Area Network (CAN) protocol with up to 16x higher data rates,” in *DATE*. IEEE, 2009, pp. 1088–1093.

The Technology Dependence of Lightweight Hash Implementation Cost

Xu Guo and Patrick Schaumont

Center for Embedded Systems for Critical Applications (CESCA)
Bradley Department of Electrical and Computer Engineering
Virginia Tech, Blacksburg, VA 24061, USA
{xuguo,schaum}@vt.edu

Abstract. The growing demand of security features in pervasive computing requires cryptographic implementations to meet tight cost constraints. Lightweight Cryptography is a generic term that captures new efforts in this area, covering lightweight cryptography proposals as well as lightweight implementation techniques. This paper demonstrates the influence of technology selection when comparing different lightweight hash designs and when using lightweight cryptography techniques to implement a hash design. First, we demonstrate the impact of technology selection to the cost analysis of existing lightweight hash designs through two case studies: the new lightweight proposal Quark [1] and a lightweight implementation of CubeHash [2]. Second, by observing the interaction of hash algorithm design, architecture design, and technology mapping, we propose a methodology for lightweight hash implementation and apply it to Cubehash optimizations. Finally, we introduce a cost model for analyzing the hardware cost of lightweight hash implementations.

1 Introduction

Lightweight Cryptography is a recent trend that combines several cryptographic proposals that implement a careful tradeoff between resource-cost and security-level. For Lightweight Hash implementations, lower security levels mean a reduced level of collision resistance as well as preimage and second-preimage resistance compared with the common SHA standard [3].

The growing emphasis on engineering aspects of cryptographic algorithms can be observed through recent advances in lightweight hash designs, which are strongly implementation-oriented. Fig. 1 demonstrates three important design fields in lightweight hash design: algorithm-specification, hardware architecture, and silicon implementation. Crypto-engineers are familiar with the relationship between algorithm- and architecture-level. However, the silicon implementation remains a significant challenge for the crypto-engineer, and the true impact of design decisions often remains unknown until the design is implemented.

This paper contributes to this significant challenge in three ways. First, by mapping an existing lightweight hash proposal to different technology nodes and

standard-cell libraries, we illustrate how important technology-oriented cost factors can be taken into account during analysis. Second, by studying the interaction between hardware architecture and silicon implementation, we demonstrate the importance of low-level memory structures in lightweight hash design. As a case study, we show how to optimize CubeHash [2], a SHA-3 Round 2 candidate with a high security level, to fit in 6,000 GEs (Gate Equivalent) by combining bit-slicing (at hardware architecture level) and proper memory structures (at silicon implementation level). Third, we built a cost model for lightweight hash designs and provide guidelines for both lightweight hash developers and hardware designers.

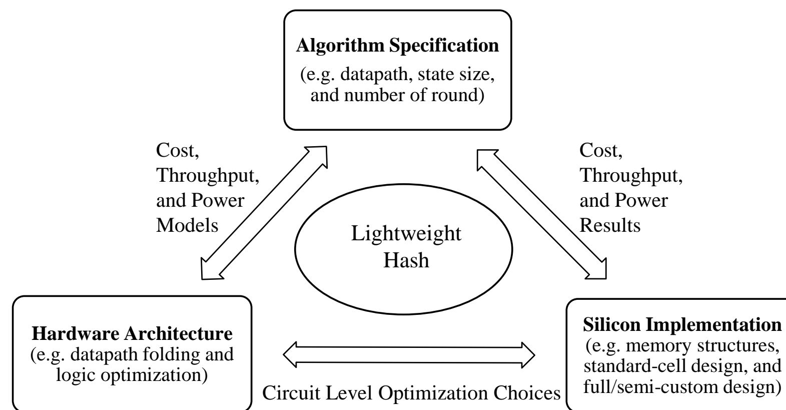


Fig. 1. The interactive process in the development of lightweight cryptography.

2 Lightweight Hash Comparison Issues

Due to the nature of lightweight hash designs they are very sensitive to small variations in comparison metrics. For example, several of them claim area around 1,000 GEs with power consumption around 2 μW. However, it appears that most lightweight hash designers only discuss algorithm implementations at logic level, and they make abstraction of important technological factors. This makes a comparison between different lightweight hash designs difficult or even impossible.

To understand the issues, one should first identify whether the selected metrics are dependent or independent of the technology. Below we discuss several metrics that are commonly used to compare different lightweight hash proposals.

2.1 Area

Most of the lightweight hash papers compare the hardware cost by using the post-synthesis circuit area in terms of GEs. However, the circuit area estimation based

on GEs is a coarse approach; it ignores the distinction between control, datapath, and storage, for example. Moreover, GEs are strongly technology dependent. In the following two case studies of Quark [1] and CubeHash [2], we will demonstrate different aspects of technology dependence of area cost.

- Standard-cell Library Impact. This is brought by different technology nodes and different standard-cell libraries. The ASIC library influence can be found by using the same synthesis scripts for the same RTL designs in a comprehensive exploration with different technology nodes and standard-cell libraries. As found in the Quark case study in Section 3, the cost variation range caused by changing standard-cell libraries can be from -17.7% to 21.4%, for technology nodes from 90nm to 180nm.
- Storage Structure Impact. Hash designs are state-intensive, and typically require a proportionally large amount of gates for storage. This makes the selection of the proper storage structure an important tuning knob for the implementation of a hash algorithm. For example, we implement several bit-sliced versions of CubeHash [4,5,6], and show that the use of a register file may imply an area reduction of 42.7% area reduction compared with a common flip-flop based memory design.

2.2 Power

The power consumption is another commonly used metric which is strongly correlated to the technology. For the standard-cell library impact, we can see from Table 1 the power efficiency in terms of $nW/MHz/GE$ differs by a factor of two to three times across different technology nodes. For the storage structure impact, as illustrated in our case study of CubeHash in Section 4, the power variation range can be from -31.4% to 14.5% at 130nm technology node. Therefore, it is important to provide proper context when comparing the power estimation results for different lightweight hash implementations.

Table 1. Compare the characteristics of different ASIC technology nodes from some commercial standard-cell libraries [7].

Technology Node	Gate Density $[kGEs/mm^2]$	Power $[nW/MHz/GE]$
180 nm	125	15.00
130 nm	206	10.00
90 nm	403	7.00
65 nm	800	5.68

However, in general we think that the power consumption is a less important metric in comparing different lightweight hash designs. Take the most popular and ultra-constrained RFID application as an example, which has the power

budget for security portion as 27 μW at 100KHz [8,9]. Just by looking at the average power number in Table 1, we can easily get a rough estimation of the area needed to consume 27 μW would be 18k GEs, 27k GEs, 39k GEs and 48k GEs at 180nm, 130nm, 90nm and 65nm, respectively. Therefore, all of the lightweight hash proposals which are much smaller should satisfy the power requirement.

2.3 Latency

The latency of hash operations is measured in clock cycles and is technology independent. However, one related metric, Throughput, may be technology dependent if the maximum throughput is required since the maximum frequency of a design is technology dependent. Since in most lightweight hash targeted applications, especially tag-based applications, hashing a large amount of data is unlikely to happen or the operating frequency is fixed at a very low value (e.g. 100 KHz for RFID), latency as a technology independent metric should be sufficient to characterize the lightweight hash performance.

2.4 Summary

As discussed above, only the latency metric is independent of technology. Power is technology dependent metric but it is a much less important metric than area. Therefore, the rest of the work will focus on the technology dependent analysis of the hardware cost of lightweight hash designs. The technology impacts of standard-cell libraries and storage structures are measured in a quantitative way through two concrete case studies of Quark and CubeHash designs.

3 ASIC Library Dependent Cost Analysis of Quark

In this section, we investigate the impact of technology library selection on the overall GE count of a design. We do this through the example of the Quark lightweight hash function.

3.1 Overview of Quark

The Quark hash family by Aumasson et al. was presented at CHES2010 [1], using sponge functions as domain extension algorithm, and an internal permutation inspired from the stream-cipher GRAIN and the block-cipher KATAN. There are three instances of Quark: U-Quark, D-Quark and S-Quark, providing at least 64, 80, 112 bit security against all attacks (collisions, second preimages, length extension, multi-collisions, etc.) [10]. The authors reported the hardware cost after layout of each variant as 1379, 1702 and 2296 GEs at 180nm.

The open-source RTL codes found at the Quark website [10] help us evaluate the standard-cell library impact on the hardware cost by reusing the same source codes. The evaluation is performed at two steps: first, we look at cost variations at three technology nodes (180nm/130nm/90nm); second, at the same 130nm

Table 2. Summary of technology nodes and standard-cell libraries used in technology dependent cost analysis.

Technology	Vendor	Standard-Cell Library	Notes
90 *nm*	UMC	*fsd0a_a_generic_core_tc*	Standard Performance Low-K
130 *nm*	UMC	*fsc0g_d_sc_tc*	Standard Performance High Density
180 *nm*	UMC	*fsa0a_c_sc_tc*	High Performance
130 *nm*	UMC	*fsc0g_d_sc_tc*	Standard Performance High Density
130 *nm*	IBM	*scx3_cmos8rf_rvt_tt_1p2v_25c*	Standard Performance
130 *nm*	IBM	*scx3_cmos8rf_lpvt_tt_1p2v_25c*	Low Power

technology node we have tried: a) two standard-cell libraries from different vendors; and b) two different standard-cell libraries from the same vendor. A list of all the technology nodes and libraries can be found at Table 2.

3.2 The Impact of Technology Nodes and Standard-Cell Libraries

To evaluate the standard-cell library impacts we used the Synopsys Design Compiler (C-2009.06-SP3) to map the Quark VHDL source codes [10] to all the different technologies and libraries listed in Table 2. The synthesis constraints are set to optimize for the minimum area.

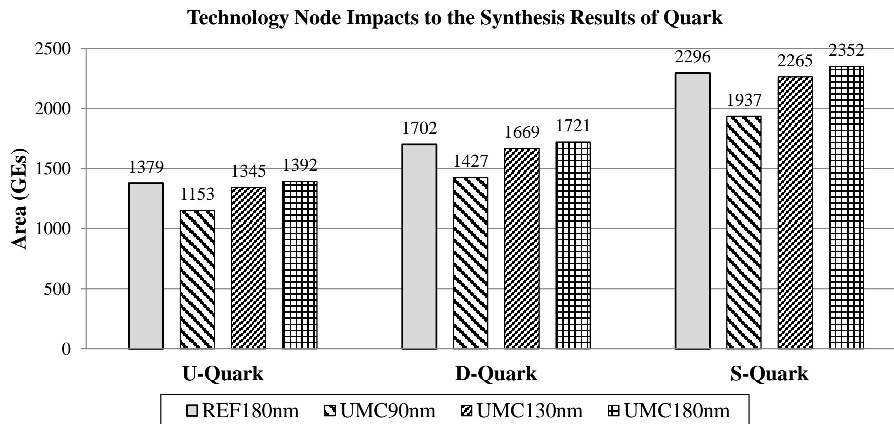


Fig. 2. Cross-comparison of Quark post-synthesis area at different technology nodes. (Note: numbers of REF180nm refer to the hardware cost presented in [1])

Take the smallest U-Quark as an example. As shown in Fig. 2, the synthesized circuit area varies from 1153 GEs to 1392 GEs in 90nm and 180nm, respectively.

The differential area, 239 GEs, implies a variation range from -17.2% to 20.7%. The similar trend can also be found for D-Quark and S-Quark. In average, the area variation is from -17.3% to 20.9% for all the three Quark variants.

The Quark RTL designs have very simple logic operations in their compression function and most of the circuit area is used for storing the state bits. For a further analysis of which part of the design contributes more to those variations, we show the portion of combinational and non-combinational logic of each synthesized design in Fig. 3. From this graph, we can clearly see that the combinational part, largely determined by the logic complexity of the compression functions, has much less variations under different technology nodes and contributes an average of 20.9% to the total area variation for all cases. The non-combinational part, mainly occupied by the storage of state bits, is more sensitive to different technology nodes.

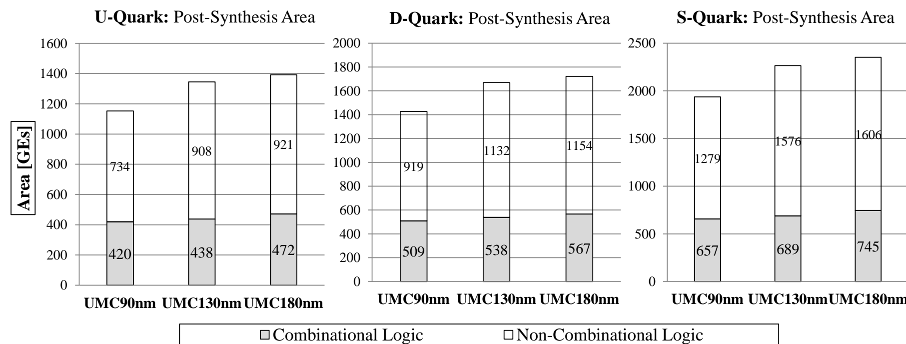


Fig. 3. Technology impact on the combinational logic *vs.* non-combinational logic after synthesis.

Through the above exploration we have demonstrated that a direct comparison of circuit area obtained at different technology nodes may cause significant inaccuracy. However, as shown in Fig. 4 comparing synthesis results using different libraries (UMC *vs.* IBM and Low Power Vt Library *vs.* Regular Vt Library) at the same $130nm$ technology node shows very small variations with an acceptable variation range from -1.1% to 1.1%. This means more accurate comparison in hardware cost could be achieved if the same standard-cell libraries are used or at least different standard-cell libraries are at the same technology node.

4 Storage Structure Dependent Cost Analysis of CubeHash

In this Section we discuss the impact of storage architecture on GE, through the example of a bitsliced Cubehash design proposed by Bernstein [4,5,6].

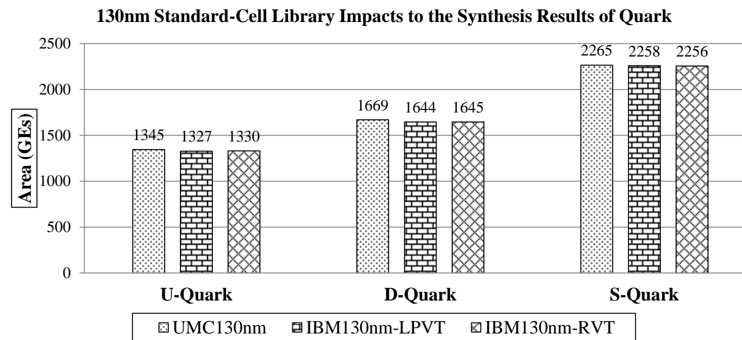


Fig. 4. Cross-comparison of Quark post-synthesis area with different 130nm CMOS standard-cell libraries.

4.1 Overview of CubeHash

CubeHash is among the 14 NIST SHA-3 Second Round candidates. Although it was not selected to be considered as one of the SHA-3 finalists, CubeHash attracted extensive third-party analysis and no practical attacks were found. CubeHash is also an exception among Second Round SHA-3 candidates in terms of hardware footprint. The smallest implementation of CubeHash was reported with 7630 GEs in 130 nm technology [11]. Because Cubehash offers a high security level with a very simple structure, it is worthwhile to look further into the lightweight implementation of this design.

CubeHash is a sponge-like hash algorithm that is based on the iteration of a fixed permutation, T, on a state of 128 bytes. A simple padding rule is applied to the input message. The algorithm consists of three stages: initialization, message injection, and finalization. In addition to the parameter, h, for the bit length of the message digest size, there are four tunable parameters: 1) the number of rounds of message injection, r; 2) the message word size, b, in bytes; 3) the initialization rounds, i; and 4) the finalization rounds, f. Thus, for the message injection stage of CubeHash$i + r/b + f - h$, b bytes of the message are XORed into the state before r iterations of T. The initialization and finalization stages each consist of i and f rounds of T on the state, respectively. Initialization may either be pre-computed to save time, or computed on-the-fly to save memory. Finalization is preceded by the flipping of a single bit of the state. No round constants or length indicators are used, other than the encodings of the parameters in the three initial-state words.

In this work we evaluated the CubeHash512 (Version November 2010), defined as CubeHash16+16/32+32-512. The security bits for preimages attacks is 384 and 256 for collision attacks. The operations in the round permutation, T, are modular additions, XORs, and rotations, which are very suitable for bit-sliced implementations.

4.2 VLSI Implementations

In Appendix A. we describe two general techniques, bit-slicing and memory structures, for lightweight hash designs. In this section, we will show how to use them in optimizing a new hash proposal, CubeHash [2], for lightweight implementations. By using CubeHash as a case study, we also show that with the combination of bit-slicing and register file as the main storage, CubeHash can also be a ideal lightweight hash candidate (less than 6,000 GEs with 130nm technology) but with much higher security levels. This makes Cubehash complementary to other new lightweight hash proposals with lower security margins. For practical aspects, we will also show the interaction between bit-slicing and register file configurations, and how this interaction will affect the hardware cost.

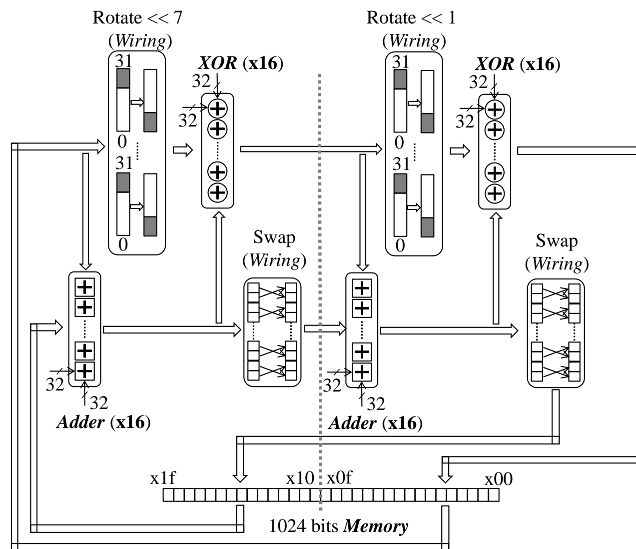


Fig. 5. The hardware architecture of CubeHash with 32 bits datapath.

The architecture of the original CubeHash with 32 bits datapath is shown in Fig. 5. The 32 bits datapath, as shown, can finish one round of CubeHash core function in one clock cycle. The left and right part of the datapath are almost identical except for the different rotations. Since rotation and swap operations can be realized through simple wiring in hardware without logics, the ALUs of datapath mainly consists of 32 bits adders and 32 bits XORs. The CubeHash state with the configuration of 1024 bits $\times$ 1 word (1024b1w) can only be implemented using flip-flops.

For the one-cycle-per-round CubeHash architecture, denoted as Cube1, the 32 bits datapath and 1,024 bits flip-flops will consume most of the circuit area. So, we can apply the bit-slicing technique to reduce the datapath bit width.

For bit-sliced ALUs of CubeHash datapath, we only need to consider the 32 bits adders. Here the ripple carry adder with full adders can be used. Although rotation and swap operations are free in hardware, for different degrees of bit-slicing preparing the corresponding data format and vector ordering needs to be carefully addressed. More details on how to map these operations in 2, 4, 8 bit-sliced versions of CubeHash can be found at [4,5,6].

We have implemented two bit-sliced versions of CubeHash with a 2-bit and a 4-bit datapath, respectively. We denote these two versions as Cube32 and Cube16. As a result of the bit-slicing, the CubeHash state memory configuration changes from the default '1024b1w' to '64b16w' and '128b8w', respectively.

We chose to define our design space only with these two bit-sliced versions for two reasons. First, the register file needs to have at least 8 fields, so that bitslicing above 4 bits would imply storage overhead. Second, after analyzing the results of Cube32 and Cube16, we found the throughput of the bit-serial implementation of CubeHash to be too slow to be competitive. This will be illustrated further.

4.3 Quantify the Storage Structure Impact

When optimizing the bit-sliced implementations of Cube32 and Cube16, in addition to the memory type selection we also have the option to implement both flip-flops and register file based memories with single-port (SP) or dual-port (DP). For single-port memories, since they have single set of address and controls with single access (read/write) to the memory, it will save some area due to the simpler peripheral circuits compared with dual-port memories with the same size. However, the CubeHash throughput will cut into half with SP-memories because in this case read and write operations need to be in separate clock cycles.

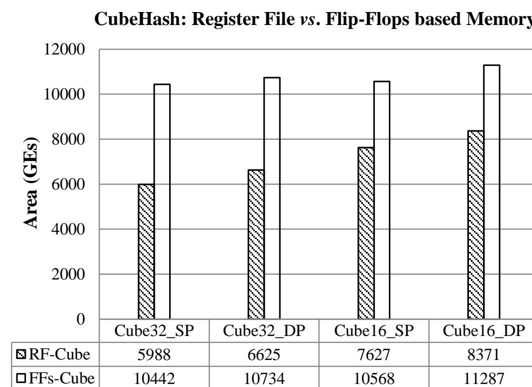
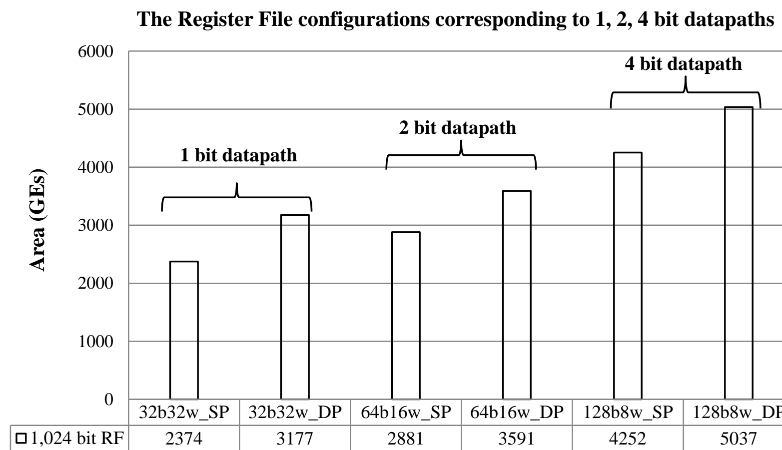


Fig. 6. Comparison of the impact of different memory types to the CubeHash area.(Note: the latency of Cube32_SP, Cube32_DP, Cube16_SP and Cube16_DP are 1024, 512, 512 and 256 cycles, respectively)

We emphasize that after choosing single- or dual-port memory types in hardware architecture design, the RTL code for a flip-flop based version and for a register-file based version is identical. Both of them share the same standard memory interface. However, after using the same synthesis constraints targeting smallest area to synthesize both of the designs with IBM MOSIS 130*nm* standard-cell technology (*scx3_cmos8rf_lpvt_tt_1p2v_25c*) and register file modules from Artisan Standard Library 130*nm* Register File Generator, the advantages of using register file becomes obvious, as shown in Fig. 6.

The RF-based CubeHash designs can save between 26% and 43% of the area over flip-flop based ones. As we look further into the memory efficiency metric in terms of GEs/bit, for the size of 1,024 bits the flip-flops based memories is almost constant with small variations between 7.0 GEs/bit and 7.8 GEs/bit; however, the register file based ones have the densest configuration of 32b32w_SP with highest efficiency at 2.3 GEs/bit and 4.9 GEs/bit for the lowest efficiency with 128b8w_DP configuration as shown in Fig. 7.

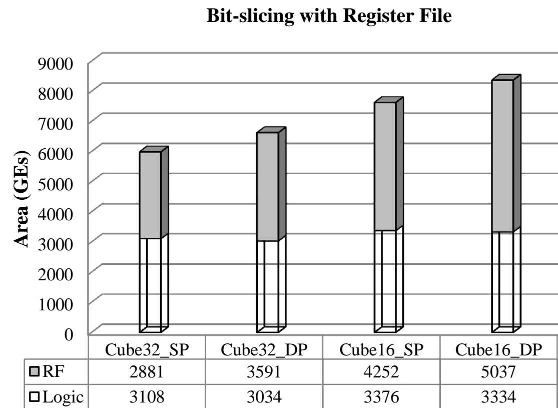


	32b32w_SP	32b32w_DP	64b16w_SP	64b16w_DP	128b8w_SP	128b8w_DP
1,024 bit RF	2374	3177	2881	3591	4252	5037

Fig. 7. Comparison of the cost of different configurations of the register file. (Note: for all single-port memories the read and write operations need to be in separate clock cycles)

After analyzing the area composition of RF-based CubeHash designs shown in Fig. 8, by reducing the datapath from 4 bits to 2 bits, we only save less than 300 GEs in average, which is less than 5% of the total CubeHash area; however, moving from Cube16_DP to Cube32_SP we can save 2,156 GEs, which is 26% area reduction of the overall Cube16_DP area.

This interesting comparison delivers an important message: bit-slicing may not provide a reduction in datapath area. In fact, in an extreme case (e.g. bit-serial implementation), bit-slicing may even increase the logic area with more incurred control overhead than the reduced area in ALUs; however, the associated changes



	Cube32_SP	Cube32_DP	Cube16_SP	Cube16_DP
RF	2881	3591	4252	5037
Logic	3108	3034	3376	3334

Fig. 8. Comparison of the storage ratio in different memory configurations.

in memory configurations may have greater impact on the overall area reduction. Note that there are also some ways to further compress the control unit design (e.g. ROM-based FSMs *vs.* conventional FSMs), but since the focus of this paper lies on the storage impact analysis we choose to use conventional FSMs based designs and leave the control overhead optimization as a future work.

For the bit-serial CubeHash design, according to the above analysis the logic area reduction will become negligible. Using register file with 32b32w_SP configuration can save 500 GEs, but this will make the latency as bad as 2048 cycles, which even cannot meet the RFID tag's requirement of 1800 cycles [8]; while for the other configuration with 32b32w_DP, with the same latency of 1024 cycles as Cube32-64b16w_SP the area of the register file will increase (possibly due to the memory shape). Therefore, bit-serial version of CubeHash is excluded in our design space.

We have also performed post-synthesis simulation and compared the power efficiency of flip-flops and register file based CubeHash designs. As we can see from Fig. 9, register file based designs are in general more power efficient than the flip-flop based ones. For all the cases, the power consumption at 100 KHz are way below the RFID's required power budget of 27 μW [8].

To compare lightweight CubeHash implementations with other published work, CubeHash can offer much higher security levels with satisfying throughput at an acceptable cost. Based on the lightweight hash design methodology proposed in this work, we have shown how to optimize a general purpose hash algorithm for lightweight applications. With the given security and cost, we think CubeHash is an ideal complement to other existing lightweight proposals. As also shown in Table 3, all these lightweight hash designs take advantages of bit-slicing but only our design makes a better usage of the memory structure.

Table 3. Comparison of the characteristics of different lightweight hash designs. (Note: 1) 'Dig.', 'Dp', 'Proc.' and 'Thr.' refer to digest size, datapath width, process and throughput, respectively; 2) throughput and power numbers are based on designs running at 100KHz; 3) a graphical presentation of selected metrics can be found in the Appendix B. for easier comparison)

Hash Function	Pre. [bit]	Coll. [bit]	Dig. [bit]	Latency [cycles]	Dp [bit]	Proc. [*nm*]	Area [GEs]	Thr. [*kbps*]	Power [μW]
DM-PRESENT-80 [12]	64	32	64	547	4	180	1600	14.63	1.83
DM-PRESENT-128 [12]	64	32	64	559	4	180	1886	22.90	2.94
PHOTON-80/20/16 [13]	64	40	80	708	4	180	865	2.82	1.59
PHOTON-128/16/16 [13]	112	64	128	996	4	180	1122	1.61	2.29
H-PRESENT-128 [12]	128	64	128	559	8	180	2330	11.45	6.44
U-Quark [1]	128	64	128	544	1	180	1379	1.47	2.44
ARMADILLO2-B [14]	128	64	128	256	1	180	4353	25.00	–
PHOTON-160/36/36 [13]	128	80	160	1332	4	180	1396	2.70	2.74
D-Quark [1]	160	80	160	704	1	180	1702	2.27	3.10
ARMADILLO2-C [14]	160	80	160	320	1	180	5406	25.00	–
C-PRESENT-192 [12]	192	96	192	3338	12	180	4600	1.90	–
PHOTON-224/32/32 [13]	192	112	224	1716	4	180	1736	1.86	4.01
S-Quark [1]	224	112	224	1024	1	180	2296	3.13	4.35
PHOTON-256/32/32 [13]	224	128	256	996	8	180	2177	3.21	4.55
ARMADILLO2-E [14]	256	128	256	512	1	180	8653	25.00	–
Keccak-*f*[200] [15]	64	32	64	900	8	130	2520	8.00	5.60
SPONGENT-88 [16]	80	40	88	990	4	130	738	0.81	1.57
SPONGENT-128 [16]	120	64	128	2380	4	130	1060	0.34	2.20
Keccak-*f*[400] [15]	128	64	128	1000	16	130	5090	14.40	11.50
SPONGENT-160 [16]	144	80	160	3960	4	130	1329	0.40	2.85
SHA-1 [8]	160	80	160	344	8	130	5527	148.88	2.32
SPONGENT-224 [16]	208	112	224	7200	4	130	1728	0.22	3.73
SPONGENT-256 [16]	240	128	256	9520	4	130	1950	0.17	4.21
Cube8/1-512 [11]	384	256	512	512	32	130	7630	2.00	–
Cube32_SP-RF	**384**	**256**	**512**	**1024**	**2**	**130**	**5988**	**25.00**	**2.91**
Cube32_DP-RF	**384**	**256**	**512**	**512**	**2**	**130**	**6625**	**50.00**	**3.91**
Cube16_SP-RF	**384**	**256**	**512**	**512**	**4**	**130**	**7627**	**50.00**	**4.33**
Cube16_DP-RF	**384**	**256**	**512**	**256**	**4**	**130**	**8371**	**100.00**	**5.68**

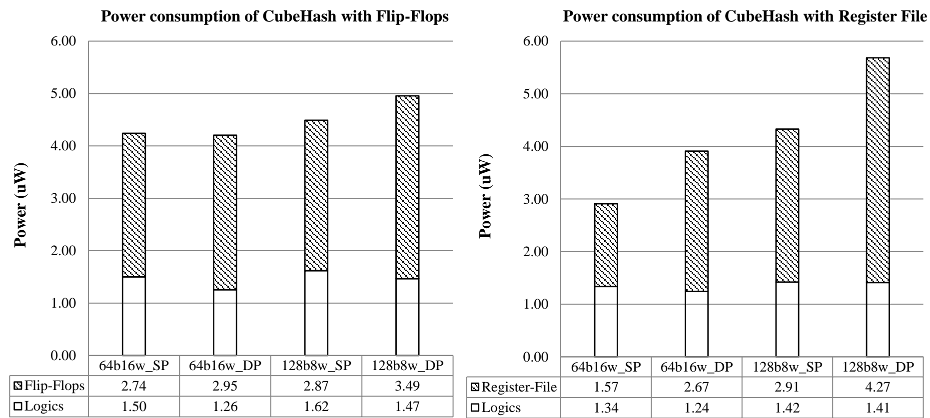


Fig. 9. Comparison of the power consumption with flip-flops and register file based CubeHash implementations.

5 Conclusions

Recent lightweight hash proposals have presented a tradeoff between security and cost; cryptographic engineers, on the other hand, have proposed more fine-grained optimizations to achieve the most efficient implementations. By quantifying technology impacts to the cost analysis of different lightweight hash implementations, this paper shows the benefits of making these two groups people working in an interactive design process and at different abstraction levels. Our technology dependent cost analysis may help cryptographic engineers have better presentation of the metrics and avoid some common pitfalls. The proposed lightweight hash design methodology establish the link between algorithm design and silicon implementation with a strong emphasis on the interaction between hardware architecture and silicon implementation. The cost model of lightweight hash designs reveals the interaction between bit-slicing and memory structures may divide the design space for lightweight implementation into two regions: one is mainly about the tradeoff between datapath folding and control overhead, and the other one needs to add the low-level memory structure as an additional tradeoff point.

Acknowledgment

This work is supported by a NIST grant, 'Environment for Fair and Comprehensive Performance Evaluation of Cryptographic Hardware and Software'. We acknowledge the support from Dr. Daniel J. Bernstein at University of Illinois at Chicago for providing bit-sliced CubeHash software implementations.

References

1. Aumasson, J.P., Henzen, L., Meier, W., Naya-Plasencia, M.: Quark: A Lightweight Hash. In Mangard, S., Standaert, F.X., eds.: Cryptographic Hardware and Embedded Systems, CHES 2010. Volume 6225 of LNCS. (2010) 1–15
2. Bernstein, D.J.: CubeHash: a simple hash function (May 2011) http://cubehash.cr.yp.to/index.html.
3. NIST: CRYPTOGRAPHIC HASH ALGORITHM COMPETITION (May 2011) http://csrc.nist.gov/groups/ST/hash/sha-3/index.html.
4. Bernstein, D.J.: CubeHash: 8-cycle-per-round hardware implementation strategy (May 2011) http://cubehash.cr.yp.to/hardware8/hash.c.
5. Bernstein, D.J.: CubeHash: 16-cycle-per-round hardware implementation strategy (May 2011) http://cubehash.cr.yp.to/hardware16/hash.c.
6. Bernstein, D.J.: CubeHash: 32-cycle-per-round hardware implementation strategy (May 2011) http://cubehash.cr.yp.to/hardware32/hash.c.
7. TOSHIBA: Toshiba CMOS Technology Roadmap for ASIC (May 2011) http://www.toshiba-components.com/ASIC/Technology.html.
8. O'Neill, M., Robshaw, M.: Low-cost digital signature architecture suitable for radio frequency identification tags. Computers Digital Techniques, IET **4**(1) (january 2010) 14 –26
9. Feldhofer, M., Wolkerstorfer, J.: Strong Crypto for RFID Tags - A Comparison of Low-Power Hardware Implementations. In: Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on. (may 2007) 1839 –1842
10. Aumasson, J.P., Henzen, L., Meier, W., Naya-Plasencia, M.: Quark: a lightweight hash (October 2011) http://131002.net/quark/.
11. Bernet, M., Henzen, L., Kaeslin, H., Felber, N., Fichtner, W.: Hardware implementations of the SHA-3 candidates Shabal and CubeHash. Circuits and Systems, Midwest Symposium on (2009) 515–518
12. Bogdanov, A., Leander, G., Paar, C., Poschmann, A., Robshaw, M., Seurin, Y.: Hash Functions and RFID Tags: Mind the Gap. In Oswald, E., Rohatgi, P., eds.: Cryptographic Hardware and Embedded Systems C CHES 2008. Volume 5154 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2008) 283–299
13. Guo, J., Peyrin, T., Poschmann, A.: The PHOTON Family of Lightweight Hash Functions. In Rogaway, P., ed.: Advances in Cryptology, CRYPTO 2011. Volume 6841 of LNCS. Springer Berlin / Heidelberg (2011) 222–239
14. Badel, S., Dağtekin, N., Nakahara, J., Ouafi, K., Reffé, N., Sepehrdad, P., Sušil, P., Vaudenay, S.: ARMADILLO: A Multi-purpose Cryptographic Primitive Dedicated to Hardware. In Mangard, S., Standaert, F.X., eds.: Cryptographic Hardware and Embedded Systems, CHES 2010. Volume 6225 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2010) 398–412
15. Kavun, E., Yalcin, T.: A Lightweight Implementation of Keccak Hash Function for Radio-Frequency Identification Applications. In Ors Yalcin, S., ed.: Radio Frequency Identification: Security and Privacy Issues. Volume 6370 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2010) 258–269
16. Bogdanov, A., Knežević, M., Leander, G., Toz, D., Varici, K., Verbauwhede, I.: SPONGENT: A Lightweight Hash Function. In Preneel, B., Takagi, T., eds.: Cryptographic Hardware and Embedded Systems, CHES 2011. Volume 6917 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2011) 312–325
17. Cannière, C., Dunkelman, O., Knežević, M.: KATAN and KTANTAN – A Family of Small and Efficient Hardware-Oriented Block Ciphers. In: Proceedings of the

11th International Workshop on Cryptographic Hardware and Embedded Systems. CHES '09, Berlin, Heidelberg, Springer-Verlag (2009) 272–288
18. Weste, N., Harris, D.: CMOS VLSI Design: A Circuits and Systems Perspective (3rd Edition). Addison-Wesley (2004)
19. Guo, X., Srivistav, M., Huang, S., Ganta, D., Henry, M., Nazhandali, L., Schaumont, P.: Pre-silicon Characterization of NIST SHA-3 Final Round Candidates. In: 14th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2011). (2011)
20. Guo, X., Srivistav, M., Huang, S., Ganta, D., Henry, M., Nazhandali, L., Schaumont, P.: ASIC Implementations of Five SHA-3 Finalists. In: Design, Automation Test in Europe Conference Exhibition (DATE), 2012. (March 2012) 1 –6
21. Guo, X., Srivistav, M., Huang, S., Ganta, D., Henry, M., Nazhandali, L., Schaumont, P.: Silicon Implementation of SHA-3 Finalists: BLAKE, Grostl, JH, Keccak and Skein. In: ECRYPT II Hash Workshop 2011. (May 2011)
22. Guo, X., Huang, S., Nazhandali, L., Schaumont, P.: Fair and Comprehensive Performance Evaluation of 14 Second Round SHA-3 ASIC Implementations. In: The Second SHA-3 Candidate Conference. (August 2010)
23. Guo, X., Huang, S., Nazhandali, L., Schaumont, P.: On The Impact of Target Technology in SHA-3 Hardware Benchmark Rankings. Cryptology ePrint Archive, Report 2010/536 (2010) http://eprint.iacr.org/2010/536.
24. Kim, M., Ryou, J., Jun, S.: Efficient Hardware Architecture of SHA-256 Algorithm for Trusted Mobile Computing. In Yung, M., Liu, P., Lin, D., eds.: Information Security and Cryptology. Volume 5487 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2009) 240–252

A. Lightweight Hash Implementation Methodology

Below we propose a methodology for lightweight hash designs and created a cost model to help designers understand when changing memory structure becomes necessary in order to further reduce the circuit area.

Hardware Architecture Optimization: Datapath Folding The complexity of hash algorithms determines the ALUs used in the datapath to complete complex arithmetic operations. For some hash algorithms with 32 bits datapath, these ALUs may become the dominant factor in the overall cost.

Fig. 10 shows an example hash datapath with four logic operations between the IN and OUT states. There are two common ways to reduce the area of ALUs in this datapath. First, through horizontal folding, we can save half of the ALUs at the cost of introducing control overhead for selecting the inputs for each round of operations. Second, through vertical folding (also known as bit-slicing), we can cut the bit width of processing elements and this will also cause control overhead.

There is no universal answer on how to select the horizontal or vertical or combined folding for the optimal architectural design. The choice has to be made by a tradeoff between ALU complexity and control unit cost. However, we recommend that lightweight hash in general may benefit more from vertical folding (or bit-slicing). One of the key reasons is that the bit-slicing will also change the memory structure as shown in Fig. 10. This may greatly affect the

optimization at the lower silicon implementation level, as will be shown through the Cubehash design further in this paper.

Silicon Implementation Optimization: Memory Structures The importance of memory design was recently identified in the new lightweight block cipher designs [17]. The authors found that most of existing lightweight block cipher implementations not only differ in the basic gate technology, but also in the memory efficiency, namely the number of GEs required for storing a bit. Most of the lightweight block ciphers have more than 50% of the total area dedicated for memory elements, mostly flip-flops (FFs), and a single flip-flop in UMC 130nm standard-cell library ($fsc0l_d_sc_tc$) may cost 5 to 12 GEs depending on the different types of FFs in use.

Compared with lightweight block ciphers, the memory requirement for a hash is much more demanding as hash functions usually have a much larger state size (e.g. typically 512 bits or 1024 bits). However, in most published lightweight hash implementations we can hardly find discussions on optimization of memory structures. In contrast, we claim that lightweight hash implementations can benefit even more from careful memory design.

There are generally three types of read/write memory in ASIC standard-cell technologies shown in Table 4. Flip-flops, SRAMs, and register files each represent a different configuration of memory cells. They each have a different density (GE/bit) and power dissipation. In addition, SRAMs and register files typically require a minimum size, defined by the technology library used. Considering our earlier discussion on architecture folding, register files and SRAMs obviously will be very useful to support vertical folding.

To summarize the selection of memory structures, for lightweight hash designs Register File (RF) is the ideal memory structure if the required memory size fits in the applicable range and configurations. Fig. 11 represents our key observations so far.

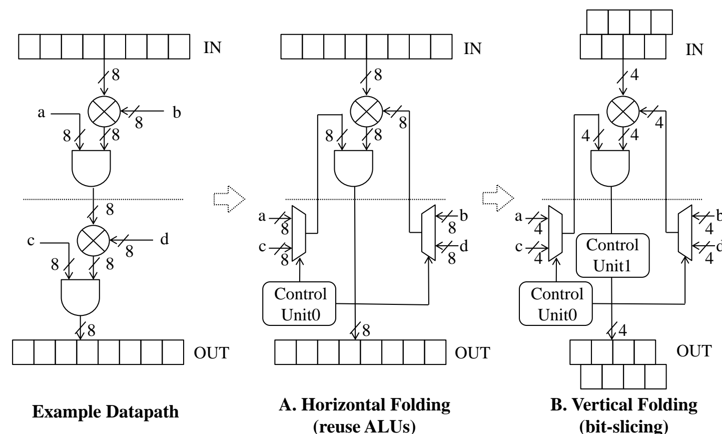


Fig. 10. The two techniques of datapath folding.

Table 4. Compare the characteristics of different memory structures in typical ASIC standard-cell technologies [18]. (*: the size parameters are for Artisan standard library 130nm SRAM and Single-Port Register File generation.)

	Flip-Flops	SRAM	Register File
Density	Low	High	Medium High
Power	High	Medium	Low
Speed	Low	High	Medium
Latency	None	1 clock cycle	1 clock cycle
*Size Range	No limit	512 bits (256 words×2 bits) to 512 kbits	64 bits (8 words×8 bits) to 32 kbits
Application	Very small size memory	Large size memory	Small-to-Medium size memory

If an algorithm has limited state size, and a register file cannot be used, then lightweight hardware implementation is mainly about the tradeoff between datapath folding and control overhead. However, as shown in Fig. 11, if an implementation can use a register file, then the cost of storage should be included in the area tradeoffs. Also, we will show that the impact of low-level memory implementation can be substantial, and that it can affect design decisions.

From the hardware engineers' perspective, following the curves in Fig. 11, one can make optimization decisions according to different hash specifications at different levels. If low cost is the primary target, one may also follow this graph to check whether your design is at the optimal point. By fully analyzing the interactions between bit-slicing and register file configurations under a target ASIC technology, one may gain a better understanding of when bit-slicing will become meaningful.

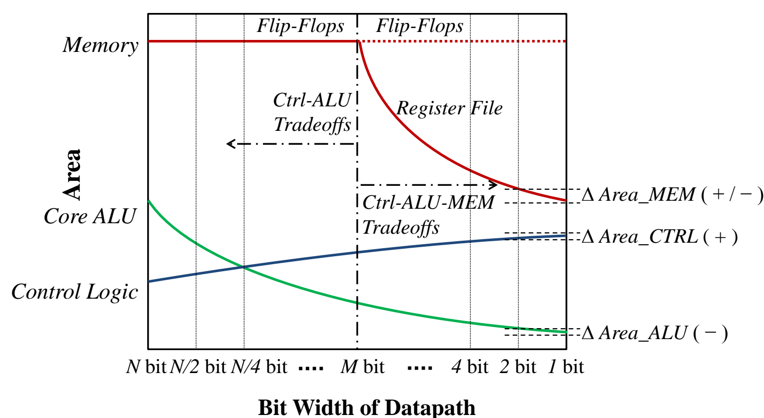


Fig. 11. The cost model for lightweight hash designs.

B. Summary of Lightweight Hash Implementations

Lightweight hash function designs fall in two research categories: lightweight hash proposals and lightweight implementation techniques.

Since DM/H/C-PRESENT [12] hash functions were proposed at CHES2008, we have seen several new lightweight hash proposals afterwards. DM/H/C-PRESENT [12] hash functions based on PRESENT block cipher and optimize for different hash constructions. Digest sizes from 64 bits to 192 bits can be achieved for a hardware cost of 2213 GEs (gate-equivalent) to 4600 GEs in 180 *nm* technology. ARMADILLO [14] hash family was first proposed at CHES2010 as a dedicated hardware optimized hash proposal. It provides several variants providing digests between 80 and 256 bits with area between 2923 and 8653 GEs. The Quark [1] hash family is based on a sponge construction with a digest size from 128 bits to 224 bits and area from 1379 GEs to 4640 GEs in 180 *nm* technology. The most recently published SPONGENT [16] and PHOTON [13] hash families are also based on sponge construction, and for the first time both of them offer one variant under 1000 GEs. SPONGENT is based on a wide PRESENT-type permutation with a digest size from 88 to 256 bits with very small footprint in hardware from 738 to 1950 GEs, respectively. PHOTON has an AES-like internal permutation and can produce digest size from 64 to 256 bits with very close hardware cost as SPONGENT from 865 to 2177 GEs.

The ongoing SHA-3 competition aims at selecting the next generation of hash standard for general applications with high demands on security requirements. According to [19,20,21], the five SHA-3 finalists in the current phase and even the fourteen Second Round SHA-3 candidates [22,23] are unlikely to be considered as lightweight hash candidates due to their high cost in hardware (more than 10,000 GEs) [1]. Nevertheless, we expect there will be additional effort dedicated to lightweight implementations of SHA-3 finalists. For some earlier work on existing hash standards, the smallest SHA-1 implementation with 160 bits digest costs 5,527 GEs in 130 *nm* technology [8]; SHA-256 with 256 bits digest can be implemented with 8,588 GEs in 250 *nm* technology [24].

To summarize previous works, two observations can be made. First, the new lightweight hash proposals emphasize the design of simplified hash core functions, rather than optimizing the implementation. Second, existing lightweight implementations focus on fine-grained or algorithm-specific optimizations. They do not provide general guidelines of how a given hash algorithm can benefit most from high level hardware architectural optimizations and low level technology optimizations.

Our work is complementary to the previous work. In this paper, we focus on the technology impacts to the cost analysis of lightweight hash designs and their relation to lightweight hash implementation techniques. Indeed, mapping a standard hash algorithm into a lightweight implementation is at least as important as stripping down hash algorithms into lightweight-security versions.

Comparison between existing lightweight hash designs and the lightweight optimization of CubeHash described in this work can be found in the graph below:

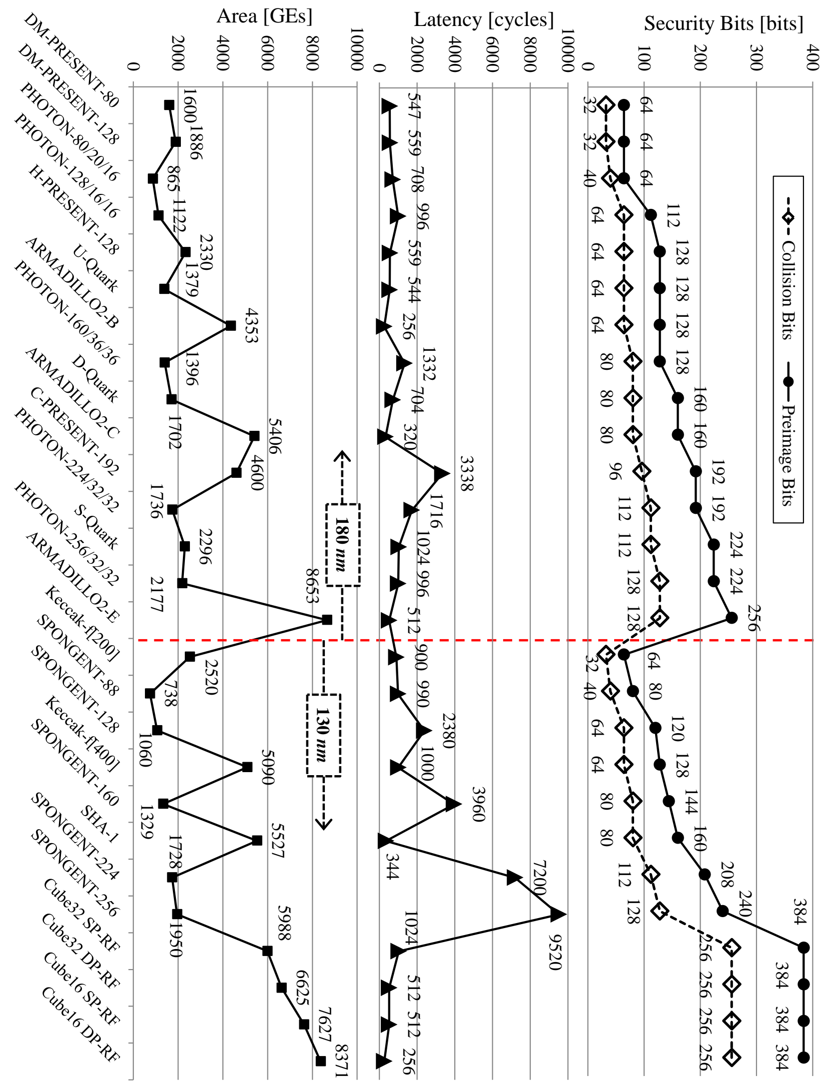


Fig. 12. Summary of existing lightweight hash implementations (see Table 3 for more comparison metrics).

Enabling Standardized Cryptography on Ultra-Constrained 4-bit Microcontrollers*

Tino Kaufmann[1,**] and Axel Poschmann[2]

[1] Gemalto, Singapore
[2] PACE, Nanyang Technological University, Singapore
tino.kaufmann@gemalto.com
aposchmann@ntu.edu.sg

Abstract. 4-bit microcontrollers (MCUs) are among the simplest, cheapest and most abundant computing devices that are embedded in a wide variety of daily-life objects. These objects, when connected to a network, could become a substantial part of the Internet of Things. Despite the fact that quite a number of applications are security sensitive, no implementation of standardized cryptography has been available yet. In this work we present the first implementation of the Advanced Encryption Standard (`AES`) on a 4-bit MCU and thus, by closing this gap, enable security functionalities on myriads of legacy devices. Besides, we describe the first software implementation of `PRINTcipher`, a recently proposed block cipher optimized for printed electronics. We describe and apply various optimization techniques to develop time and code-size efficient implementations on the MARC4. As a result we gain the most energy efficient implementations of a cryptographic algorithm on a 4-bit MCU.

Key words: 4-bit microcontroller, `AES`, `PRINTcipher`, lightweight software implementation

1 Introduction

4-bit microcontrollers (MCUs) are among the simplest, cheapest and most abundant computing devices that are embedded in a wide variety of daily-life items ranging from toys and watches to home appliances. Furthermore, 4-bit MCUs are also embedded in security sensitive applications, such as remote access and control systems, banking tokens, pervasive healthcare, car immobilizers, and all kind of sensors, such as Tire Pressure Monitoring systems (TPMS) of cars [2].[3]

The later use-case is expected to increase significantly since the U.S. Government passed the “Transportation Recall Enhancement, Accountability, and Documentation” Act (Tread Act) in the year 2000, stating that it is mandatory

* The authors were supported in part by the Singapore National Research Foundation under Research Grant NRF-CRP2-2007-03.

** The work was done while the author was at PACE, NTU.

[3] Please note that modern TPMS also use 8-bit MCUs.

for every new car in the U.S. to be equipped with a TPMS [24]. From the beginning of 2012, new passenger cars in the European Union also need to be fitted with TPMS systems [28]. However, in a recent attack on TPMS, Rouf et al. were even able to force the emergency flat tire warning LED to flash from a remote by-passing car just by sending fake sensor readings [23].

Though this real-world example highlights the strong needs for security and safety on these ultra-constrained devices, cryptography is barely used for TPMS or other applications of 4-bit MCUs. In fact, the first implementation of a cryptographic algorithm on a 4-bit MCU was published in 2009 [29]. The authors chose ATMEL's MARC4 MCU [1] and the block cipher `PRESENT` [7], as it uses 4-bit S-boxes and they assumed this to be an advantage over 8-bit S-boxes as used in e.g. the Advanced Encryption Standard (`AES`) [20]. Shortly after this proof-of-concept, the newly proposed, and already broken [17], lightweight block cipher `HUMMINGBIRD` [11] has been implemented on the same MCU.

In contrast, there is a rich literature on efficient implementations on 8-bit MCUs, and so it is no wonder that there are highly efficient implementations available. Rinne et al. describe a speed-optimized implementation of the `AES` on an 8-bit AVR microcontroller that requires 3,766 clock cycles for encryption, while at the same time occupying 3,410 bytes of memory [22]. On a PIC16xxx microcontroller `AES` requires only 1,274 bytes of program memory (ROM), 38 bytes of data memory (RAM) and 5,273 cycles to encrypt one block [18]. Recall that our goal was not to compete against these highly efficient `AES` implementations on 8-bit MCUs, but rather to enable standardized cryptography on 4-bit microcontrollers.

Thus our main contribution is to present the first implementation of `AES` on a 4-bit MCU. Besides, we also describe the first software implementation of `PRINTcipher`, a recently proposed [14] block cipher optimized for printed electronics, which are hailed to be enablers for the Internet of Things [26]. We provide a variety of timing-attack resistant implementations optimized for speed or code size, offering encryption-only, decryption-only, or encryption-and-decryption functionality. We also investigated bit-sliced implementations, however, only `PRINTcipher` fit into the scarce memory resources. Our `AES` implementations are up to approximately 3 times faster than `HUMMINGBIRD` and around 8 times faster than `PRESENT`, while our `PRINTcipher` implementations need by far the least amount of code size compared to any other implemented block cipher on a 4-bit MCU.

To summarize our contribution, by implementing `AES` on a 4-bit MCU, we enable security functionalities -such as secure code update, authentication, confidentiality etc.- based on standardized cryptography on these ultra-constrained yet vastly deployed devices. Thus, myriads of legacy devices already deployed over the last couple of decades, and gradually becoming part of the Internet of Things, might benefit from our implementation.

One might argue that Moores Law will provide abundant computing power in the near future. However, Moores Law needs to be interpreted contrary here: rather than doubling the performance, it will halve the price for constant com-

puting power each 18 months. As a consequence, cheaper, i.e. lightweighter, applications can always be deployed earlier than costlier applications, and hence will always be of high demand [21].

The remainder of this paper is organized as follows: in Section 2 the MARC4 microcontroller, the tool chain to program it and the qFORTH programming language are described. Then the block ciphers `AES` and `PRINTcipher` are recalled briefly in Section 3. Subsequently, in Sections 4 and 5, the evolution of our naïve `AES` and `PRINTcipher` implementations to their speed and code-size optimized implementations is described. In Section 6, we discuss our results and compare them with `HUMMINGBIRD` and `PRESENT`. The paper is concluded in Section 7.

2 MARC4 Microcontroller

In this section we describe the MARC4 starter kit, before we detail the MARC4 microcontroller, and finally introduce the programming language qFORTH.

2.1 The MARC4 Starter Kit

A complete programming kit for the MARC4 microcontroller comes with Atmels *MARC4 Starter Kit*. It includes a software suite (based on the Microsoft Windows operating system) with an integrated qFORTH compiler, a flash programmer [4], and a simulator to test the MARC4 core functionalities. The hardware components of the *MARC4 Starter Kit* are a programmer, a ready-to-run application board, and 5 samples of the ATAM893-D microcontroller.

2.2 The MARC4 Microcontroller

The MARC4 ATAM893-D microcontroller consists of a stack based 4-bit CPU core which is based on a Harvard architecture with 4 Kbytes of ROM (extensible to 10 Kbytes) and a RAM size of 256·4-bit. It inherits a Reduced Instruction Set Computing (RISC) core, with 72 8-bit instructions. These instructions are optimized for the qFORTH language. The MARC4 does also contains an EEPROM, as well as several on-chip peripherals, such as bidirectional I/O ports, timer and counters [1].

The supply voltage for the MARC4 ranges from 1.8V to 6.5V, with a current consumption of less than 1mA in active mode [2]. Due to its low power consumption, high speed, and its operability in a wide range of temperatures, ranging from -40 to 125 degree Celsius, the ATAM893-D can be used in a wide variety of applications. It can be found in home automation systems, such as remote door openers, it is used for industrial applications, such as remote control systems (e.g. cranes), as well as in the automotive industry, where it can be found in remote keyless entry and tire pressure monitoring systems [2].

Core Components The core components of the MARC4, includes the Read Only Memory (ROM), Random Access Memory (RAM), the RAM address registers, the Condition Code Register (CCR), the Arithmetic Logic Unit (ALU) and its interrupt structure [1]. A depiction of the MARC4 core is presented in Figure 5 in the Appendix.

- Data Memory (RAM) - The MARC4 contains a 256·4-bit wide Random Access Memory (RAM). The RAM is used as data memory and for the Expression and the Return stack. With the help of the 8-bit RAM address registers X and Y any 4-bit element in the RAM can be accessed. Another way to access the RAM is by either using the Expression Stack Pointer (SP) or the Return Stack Pointer (RP) [3].
- Program Memory (ROM) - The programmed code for the MARC4 is stored in the ROM. The ROM consists of a basebank with a size of 4 Kbytes and four ROM banks, each of size 2 Kbytes. The code within the banks is accessed by the MARC4 Program Counter (PC). With the basebank and the four ROM banks a theoretical maximum of 12 Kbytes program code can be accessed and stored. Since 2 Kbytes are reserved for testing purposes, or for predefined start addresses for the interrupt service routines, the effective memory size is 10 Kbytes [1].
- Stacks - The MARC4 contains two stacks, the expression and the return stack, which are addressed by the Expression Stack Pointer (SP) and Return Stack Pointer (RP), respectively. The operands for all arithmetic, I/O, and memory operations, are provided by and returned to the expression stack. The element on top of the expression stack is denoted as *Top of Stack*, and abbreviated as TOS. The element on the second highest position is denoted as TOS-1, and so forth. For storing the return addresses of functions and interrupts, the return stack is used [1].
- ALU - All arithmetic operations are conducted by the ALU of the MARC4. The ALU takes the TOS and TOS-1 as inputs and writes the result of the operation back on TOS.
- Interrupts - The MARC4 has an integrated interrupt controller and a total of eight interrupts, while each interrupt is assigned with a different priority. It is hereby possible to generally activate or deactivate the interrupts for the MARC4, or individually for each of the eight interrupts. All of the MARC4 interrupts can be triggered by internal and external hardware, or by a software interrupt [1].

Peripheral Communication The MARC4 contains a total of 16 bi-directional I/Os in five ports (port1, port2, port4, port5, and port6). All ports can be used by the MARC4 to receive and send data. Since each port has its own bitwise programmable port control register, it is possible to set each of the 16 pins as input or output [3].

2.3 Programming Language

In contrast to many other microcontrollers, the MARC4 is not programmed in assembly, but in qFORTH. This high-level programming language is a 4-bit version of the FORTH-83 standard [27]. It is a stack-oriented programming language that operates on Reverse Polish Notation (RPN) [8]. In this section we will give a brief description of the qFORTH language and present some of the differences to the programming language C.

Reverse Polish Notation In languages using RPN all operands are first pushed on the stack before an operand will fetch and process them. After the operation has been finished the result will be pushed on the stack again. **Example** If we want to add 3 to the value of 4 we would normally write : $3 + 4$
In RPN we would write the same operation using the following order: 3 4 +.

Introduction to qFORTH The stack-based qFORTH language uses the described expression stack to perform arithmetic operations on data. Data in qFORTH is in general considered as unsigned integer values, for both memory addresses or data values. The expression stack is hereby used to fetch the input data for an arithmetic operation and to temporarily store its result, before it is written back to a variable [1].

Due to the RPN used for the MARC4 it is necessary to first write the data on the stack before they can be manipulated. Basic stack operations, s.a. `SWAP`, as well as basic arithmetic and logical operations, s.a. `ADD`, `XOR`, `ROL` etc. require 1 clock cycle, assuming that the data is available on TOS (and TOS-1, respectively). Once the X register has been set, it takes 1 clock cycle for memory read/write operations (2 clock cycles if X register was not set).

qFORTH vs. C Common structures of languages, such as C can also be found in qFORTH. These include, but are not limited to, conditional, loop and comparison structures. Since qFORTH is a stack based language, its structures follow a different syntax compared to its equivalent structures used in non stack based languages, such as C. Table 1 presents the qFORTH syntax for various conditional and loop structures, in comparison to its C language equivalents.

Table 1. Comparison of the qFORTH and C language syntaxes for various structures, based on [1] and [5].

Structure	qFORTH	C
Conditional structures		
IF	$< cond > IF$ $< ops > THEN$	$IF\,(< cond >)$ $\{< ops >\}$
IF...ELSE	$< cond > IF < ops >$ $ELSE < ops > THEN$	$IF\,(< cond >) < ops >$ $ELSE\,\{< ops >\}$
LOOP control structures		
WHILE LOOP	$BEGIN < cond >$ $WHILE < ops > REPEAT$	$WHILE\,(< cond >)$ $\{< ops >\}$
FOR LOOP	$< limit > < start > DO$ $< ops > < offset > +LOOP$	$FOR\,(< start >; < limit >$ $; < offset >)\{< ops >\}$

3 Introduction to `AES` and `PRINTcipher`

In this section we will briefly recall the block ciphers `AES` and `PRINTcipher`.

3.1 The Advanced Encryption Standard

The Advanced Encryption Standard (`AES`) is a symmetric block cipher with a block size b of 128 bits and a key size of 128, 192 or 256 bits, called `AES-128`, `AES-192`, and `AES-256`, respectively [20]. Since `AES-128` has been implemented for the MARC4, the remainder of the `AES` description will only focus on the algorithm with a 128 bit key size. Two states are needed for the `AES`, the *key*

and the *cipher state*. Each state consists of *r=4* rows and *c=4* columns. The indexes of the state elements of the *cipher state* are denoted as $s[r,c]$ with $s[r,c] \in S$, S being the *cipher state*. The state elements of the *key state* are denoted as $k[r,c]$ with $k[r,c] \in K$, K being the *key state* [20]. An AES round is composed of the following four operations applied to S in consecutive order:

- **SubBytes** Every byte of S is substituted independently with its corresponding entry of the 8-bit AES S-box.
- **ShiftRows** A cyclic shift of the state bytes $s[r,c] \in S$, with $0 \leq r,c \leq 3$ is performed to compute the state array elements $s'[r,c]$, with $s'[r,c] = ShiftRows(s[r,c]) = s[r, c - r \, mod \, 4]$.
- **MixColumns** The MixColumns layer is the only AES operation that does not operate on a byte, but column level. It can be described as a matrix multiplication, of a fixed matrix M and the *cipher state* S.

$$S' = \overbrace{\begin{pmatrix} 02\;03\;01\;01 \\ 01\;02\;03\;01 \\ 01\;01\;02\;03 \\ 03\;01\;01\;02 \end{pmatrix}}^{M} \cdot S$$

- **KeyAddition** XOR of all *cipher state* elements $s[r,c]$ with their corresponding round dependent key state elements $k[r,c]$.

AES consists of an initial round, followed by 9 consecutive rounds and a final round. The initial round is composed of the KeyAddition operation, while for the final round, all but the MixColumns layer are performed. The interested reader is referred to [20] and [10] for further details on AES.

The AES Key Schedule generates all round keys and applies the following operations to K: Its last column entries are first rotated by one byte and subsequently substituted by the same S-box as used in the cipher-state computation. Then a round counter is added to *K[0,3]* and finally the result is added to column c_0. The remaining three columns c_i, with $1 \leq i \leq 3$, are obtained by the computation of $c_i \leftarrow c_i \oplus c_{i-1}$.

3.2 PRINTcipher

PRINTcipher [14] is a recently developed block cipher designed for ultra-constrained devices and in particular for printed electronics. It encrypts blocks of size b, with b=48 or 96 bits, denoted as PC-48 and PC-96, through $r = b$ rounds with the help of a key K of size $\frac{5}{3} \cdot b = 80/160$ bits. K is hereby the concatenation of two subkeys *sk1* and *sk2*, where *sk1* has a size of b bits and *sk2* $= \frac{2}{3} \cdot b$ bits respectively [14]. One round of PRINTcipher consists of the following five consecutive layers:

- **Key Addition** Performs a bitwise XOR of all *cipher state* bits with their corresponding sk_1 entries.
- **Permutation** A function F performing a b to b-bit permutation of each bit i of the *cipher state*.

- **Round Counter Addition** For each round r a counter RC_r is added to the lowest i bits of the *cipher state*, with $i = 6$ for PC-48 and $i = 7$ for PC-96 respectively. The round counter is updated at the beginning of each round and is created by a linear feedback shift register.
- **Keyed Permutation** Performs a key dependent (i.e. dependent on key *sk2*) permutation of the *cipher state* bits. The *cipher state* is hereby divided into words i of size 3, whereas *sk2* is divided into words j of size 2. Depending on j, the positions of the 3 bits (bit2— bit1—bit0) within i are permuted.
- **Substitution** `PRINTcipher` operates on 3-bit S-boxes which map the inputs x to their corresponding outputs $S[x]$.

As one of the design features, `PRINTcipher` has no key-schedule. The interested reader is referred to [14] for further details. A recent cryptanalysis of `PRINTcipher` by Leander et al., identified a subset of weak keys of `PRINTcipher` susceptible to chosen plaintext distinguishing attacks. In addition, the authors have also proposed a countermeasure to thwart their attack on `PRINTcipher` which only consists of assigning two counter bits to each S-box [16].

4 Implementation of `AES`

Starting from our first `AES` implementation approach, subsequently denoted as naïve implementation, the `AES` encryption routines and their optimizations will be presented in detail. Optimizations for our implementation include methods to achieve a timing-attack resistant implementation that is optimized for either high speed or small code size.

To implement the 128 bit `AES` *cipher states* and *key states* for a 4-bit architecture it is necessary to realize each of the 8-bit `AES` state entries as two 4-bit values $state_{rcn}$, where $n = 0{,}1$ denotes the high or low nibble (similarly k[r,c] is mapped to w_{rcn}).

4.1 Naïve implementation

In the following, we will give a brief overview of our naïve `AES` implementation and summarize the implementation results in Table 2.

Substitution Layer Lookup tables are, contrary to most other MARC4 constructs, operating on 8 bit, which allows to store the 8-bit entries of the `AES` substitution tables directly in the ROM of the microcontroller. The 256 entries of the `AES` S-box are split into i tables $sbox_i$, with $0 \leq i \leq 15$, each containing 16 entries that are stored consecutively in the ROM starting from the address 200h. With the help of two temporary variables, *temp* (addresses $sbox_i$) and *temp2* (contains $sbox_i$ entries), the address offset for each of the 256 S-box entries is computed. By adding the offset to the ROM address containing the first S-box element (200h) we can obtain the substituted values for each *cipher state*. The substitution itself is realized with the help of two functions, *SubBytes* and *Sbox*. The *SubBytes* operation gets the *cipher state* values and stores them in *temp* and *temp2*. These serve as input for the *Sbox* function. The naïve implementation performs the substitution layer as a case statement that first pushes the base ROM address and the address of the corresponding $sbox_i$ on

the stack ($\underbrace{2}_{TOS-2}\ \underbrace{temp}_{TOS-1}\ \underbrace{0}_{TOS}$) followed by the addition of the second offset *temp2* ($\underbrace{2}_{TOS-2}\ \underbrace{temp}_{TOS-1}\ \underbrace{temp2}_{TOS}$). Depending on the value of *temp*, a different amount of cycles is needed to substitute one *cipher state*, which can be exploited by timing attacks. A high level overview of the substitution function is depicted in Figure 1.

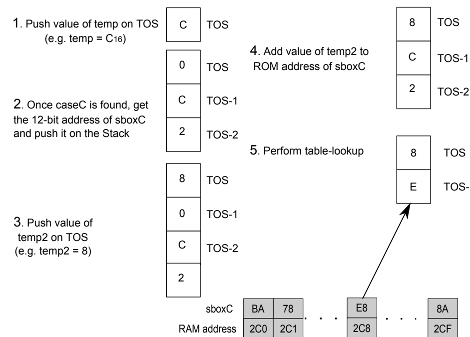


Fig. 1. High level overview of the naïve `AES` SubBytes implementation, with example values $temp = C_{16}$ and $temp2 = 8$.

ShiftRows For the ShiftRows layer, the values of each $s[r, c]$ of the rows 1-3 are first pushed on the stack and afterward written back to their new *cipher state* entries.

MixColumns The MixColumns multiplications are implemented by making use of simple shift and `XOR` operations. The coefficients of M are either one, two or three, and have been implemented using the techniques described in [25]. A multiplication with two is a simple left shift of the 8-bit on TOS and TOS-1 by one bit. In case TOS was greater than 7, we additionally need a modulo reduction step, that is an `XOR` with the 8-bit word "1B". This procedure is also known as the *xtime* step. A multiplication with three is simply multiplying the 8-bit value with two and afterward add the result to its original. For our implementations, we did not precompute and store the *xtime* results in lookup tables but computed them on the fly in order to minimize the code size. In order to check whether we have to perform the modulo reduction step, we have to first know whether the high nibble is greater than 7. In the naïve implementation of the MixColumns layer this was done with an *IF...ELSE* structure, which is data dependent and, hence, vulnerable against timing attacks.

KeyAddition And Key Update Before the *key state* can be added to the *cipher state*, it is necessary to first update all *key states* with a new set of keys, for all but the initial `AES` round. This is done within the `AES` Key Schedule function. A part of the Key Schedule operation is hereby the substitution of the last row of the *key state* with the help of the *Sbox* function. After the new keys have been

computed, they are applied to the state entries. Since the execution time of the *Sbox* function for the naïve Substitution layer is not fixed, a non-fixed time is also needed to update the *key state* which makes the naïve KeyAddition layer vulnerable against timing attacks.

Implementation results of the naïve AES Encryption Routine All in all our naïve implementation requires 2,666 bytes of memory and between 34,233 and 46,297 clock cycles for encrypting one block. Table 2 presents the implementation results for the naïve AES encryption routine layers.

Table 2. Implementation results for the naïve AES encryption routine layers.

Layer	Code size (bytes)	Time (cycles)
SubBytes	422	677-1,573
ShiftRows	66	68
MixColumns	953	2,353-2,449
KeyAddition	619	511-735

4.2 Speed Optimization

In the following paragraphs we will describe the iterative approach to find a speed-optimized AES implementation. For all speed-optimized AES and PRINTcipher implementations we realized their rounds as subroutines in which we sequentially hardcoded their layers.

The Substitution Layer and ShiftRows Optimization Process
We took the following four steps to gain a speed improvement of up to 85% and a code-size reduction of up to 75% for the Substitution and ShiftRows layers. An overview of the speed and code-size improvements for each of the four steps is presented in Table 3.

I Omitting the Case Structure Since the table lookup can be performed directly after the three words, 2, *temp*, and *temp2* have been pushed on the stack we can omit the case structure. As a consequence, there is no data-dependent time consumption anymore and, hence, the substitution layer is resistant against timing attacks.

II Direct ROM Addressing Instead of first storing $s[r, c]$ in two temporary variables, it is possible to directly push $s[r, c]$ on the stack, process them within the *Sbox* function and then store the substituted values back in $s[r, c]$.

III Consecutive Substitution using Fixed RAM Addressing of Variables The previous implementation of the substitution layer substituted one *cipher state* (2 nibbles) at a time and wrote the result back afterward. The disadvantage of this approach is the fact that we have to reset the X register content after every 8-bit substitution, since X contains the lower address of the substituted nibble, which is two addresses away from the next value to be substituted. In order to further reduce the time consumption of the substitution layer it is necessary to adjust the processing order within the *SubBytes* function. Instead

of substituting and writing back one state entry at a time, it is more efficient to first substitute all values and afterward write them back in reverse order. This requires us to store the variables at consecutive memory addresses.

IV Combining the SubBytes and ShiftRows Layers After the *cipher state* values have been substituted they are located on the stack and can directly be written back to their shifted locations.

Table 3. Time and code sizes of the final optimized combined `AES` SubBytes and ShiftRows implementation in comparison to its previous, and its naïve implementations.

		Implementation		Di erence (%)
		Naïve	I	
	Time (cycles)	(677 to 1,573) + 68	485+68	-25.78 to -66.30
	Code size (bytes)	422 + 66	205 + 66	-44.37
		I	II	
	Time (cycles)	485+68	277+68	-37.61
	Code size (bytes)	205 + 66	133+66	-26.57
		II	III	
SubBytes+Shiftrows	Time (cycles)	277+68	261+68	-4.64
	Code size (bytes)	133+66	133+66	-0.00
		III	IV	
	Time (cycles)	261+68	246+5	-23.71
	Code size (bytes)	133+66	118+5	-38.19
		Naïve	IV	
	Time (cycles)	(677 to 1,573) + 68	246+5	-66.31 to -84.70
	Code size (bytes)	422 + 66	118+5	-74.80

MixColumns Layer

A more efficient way to check whether the high nibble is greater than 7 prior to the *xtime* calculation is to use MARC4 `CCR` register instead of an *IF...ELSE* structure. The `CCR` register contains a carry bit that is set whenever an operation causes an overflow of a MARC4 word. Is the carry bit set after the high nibble was multiplied by two, the modulo reduction step has to be performed, which consumes 5 clock cycles for the `IF` branching. The `ELSE` branch, however, only requires 2 clock cycles, hence three `nop` operations are inserted to ensure data-independent timing. To further speed up the MixColumns layer we applied the efficient MixColumns layer implementation technique proposed in [10] and presented in (1). With our speed optimized MixColumns layers we achieved a 57% speed up with a code-size increase of 69%.

Pseudo-code for the efficient implementation of the `AES` MixColumns layer:

$$t \leftarrow column_0 \oplus column_1 \oplus column_2 \oplus column_3 \quad (1)$$

$$u \leftarrow column_0$$

$$v \leftarrow (column_0) \oplus (column_1) \cdot 2; column_0 \leftarrow column_0 \oplus v \oplus t$$

$$v \leftarrow (column_1) \oplus (column_2) \cdot 2; column_1 \leftarrow column_1 \oplus v \oplus t$$

$$v \leftarrow (column_2) \oplus (column_3) \cdot 2; column_2 \leftarrow column_2 \oplus v \oplus t$$

$$v \leftarrow (column_3) \oplus (u) \cdot 2; column_3 \leftarrow column_3 \oplus v \oplus t$$

KeyAddition and KeyUpdate
A part of the round key computation is the substitution of the last row of the *key state* [20]. Here we benefit from the optimizations of the *Sbox* routine described above and as a result derive a smaller and faster *KeyAddition* routine that is also resistant against timing attacks. After the *key state* has been updated, the new key is XORed to the state entries (+53% speed up and 49% code-size reduction).

The total code size of the speed-optimized encryption routine is slightly higher (around 3%) than the one of the naïve implementation. This is due to the increased size of the speed-optimized MixColumns layer. On the other hand, we were able to reduce the encryption time with our speed-optimized implementation by around 54% to 66%, compared to the naïve AES encryption routine.

4.3 Code-Size Optimization

The foundation of the AES code-size optimization process is the speed-optimized AES implementation. To reduce the code size for our implementations, it is necessary to avoid that repeating operations result in repeating source code, but instead are processed within loop constructs whenever possible. We have therefore coded the rounds for all code-size optimized AES and PRINTcipher implementations as loops. The operations within the different layers of a round are realized as subroutines that are processed within loops again to minimize their code size.

For the speed optimization it is convenient to make use of variables to load and store values, instead of directly working with the X and Y registers. During compilation, these are automatically converted to operations, in which either the X or Y register is used instead of the user defined variables. A major advantage of using the X and Y RAM address registers instead of defined variables is the fact, that it is possible to work and iterate over neighboring RAM addresses, which allows to significantly reduce the code size.

An overview of all code size and speed-optimized AES and PRINTcipher implementations is presented in Table 7 in the appendix. As one can see, the time for the code-size optimized implementation of the AES encryption routine is increased by around 50%, in comparison to its speed-optimized implementation, while its code size is reduced by around 58%.

5 Implementation of PRINTcipher

In the following section we will describe the evolution of our PRINTcipher implementations from a naïve approach to our speed and code-size optimized encryption and decryption routines. Due to the high similarity between PC-48 and PC-96, we are omitting the description of the latter. The implementation results for their speed and code-size optimized implementations are presented in Table 5, together with the combined encryption and decryption routines and our bit-sliced PC-48 implementations.

5.1 Naïve implementation

Our naïve implementation of PRINTcipher was far more advanced than the naïve implementation of AES, since we incorporated many of the tricks straight from

the beginning. We have applied all the previously described optimization techniques I - III to PRINTcipher, and of course inserted nop operations to ensure data-independent timing. In contrast to AES, our naïve PRINTcipher implementation was already timing-attack resistant.

Key Addition Within the PRINTcipher Key Addition layer all *sk1* bits are added to their corresponding *cipher state* bits. The X and Y registers are first filled with the addresses of the lowest cipher and key words, added, and afterward written back. Afterward it is possible to iterate over X and Y to compute the remaining 11 cipher/key pairs.

Permutation Layer The PRINTcipher Permutation layer performs a 1 to 1 bit permutation. Since it is not possible to access a single bit of a qFORTH word directly, but only the complete 4-bit word, various manipulations have to be performed on each word to access and permute each *cipher state* bit. We are hereby not permuting the bits of just a single state one after another, but permute three states at a time. Figure 2 displays the mapping of *F(i)* for three states of our 4-bit word implementation of the PRINTcipher Permutation layer.

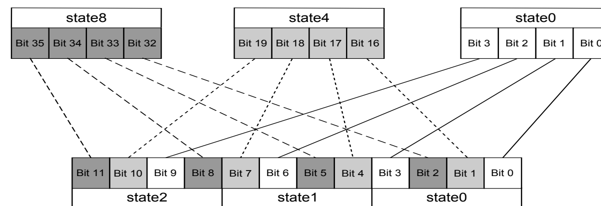


Fig. 2. Mapping of each bit i of the PRINTcipher -48 Permutation layer for a 4-bit word implementation.

Round Counter Addition The round counter is added to the lowest 6 bits of the *cipher state*. The 48 round counters are stored in three tables, each containing 16 values, in the ROM of the MARC4.

Keyed Permutation For the Keyed Permutation layer, three consecutive *cipher state* bits are permuted at a time, depending on the value of two *sk2* bits. This is done by the *keyperm* function, as part of the Keyed Permutation layer. The *cipher states* as well as the *pkey* states have to be preprocessed by shifting the necessary bits for the *keyperm* function in temporary variables. The two *pkey* bits for each keyed permutation operation are shifted into the variable *temp* and three bits of the *cipher state* bits are shifted into *temp2*, respectively. Depending on the value of *temp*, the three bits in *temp2* are permuted among another.

Substitution Layer PC-48 makes use of a 3- to 3-bit S-box, that is applied in parallel 16 times to all 48 *cipher state* bits. Since the *cipher state* bits are stored in 4-bit words, we can not perform the substitution on the complete 4-bit word, but first have to shift the corresponding 3-bits of each *cipher state* into a temporary variable and afterward write it back to its 4-bit entry. Figure 3 displays the mapping of the *cipher state* bits to their S-boxes.

Implementation results for the Naïve PRINTcipher Encryption Routine The implementation results for the naïve PRINTcipher encryption layers are

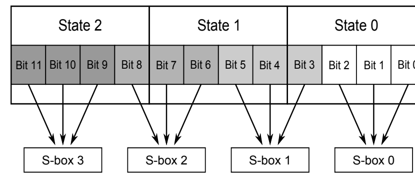


Fig. 3. Mapping of the 4-bit *cipher states* to the 3- to 3-bit S-box.

summarized in Table 4. All in all our naïve implementation requires 1,397 bytes of memory and 89,726 clock cycles for encrypting one block.

Table 4. Implementation results for the naïve PC-48 encryption layers.

Layer	Code size (bytes)	Time (cycles)
Key Addition	52	54
Round Constant Addition	62	26
Permutation	300	382
Keyed Permutation	445	931
Substitution	324	468

5.2 Speed optimization

We took the following three steps to gain a speed improvement of up to 66% for the `PRINTcipher` implementations.

I Combining the Keyed Permutation and SubBytes Layer
As stated in [14], it is possible to combine the Keyed Permutation and the Substitution layer, to a new *Combined Substitution* layer using four virtual S-boxes $V_0[x]$ - $V_3[x]$ (+21% speed up).
II 3-Bit Word Implementation
The naïve implementations of `PRINTcipher` were based on the idea of using the complete wordsize (4-bit) of the MARC4 for its implementation. The advantage of such an approach is the minimized amount of storage variables for the *cipher state* bits and $sk1$. However, since most layers of PC-48 (and PC-96) are operating on 3-bits, there is a large conversion overhead (both in code size and in time) from a 4-bit representation to a 3-bit representation and vice versa. Thus, by using a 3-bit representation, that is only the 3 LSBs of every variable are used, we were able to significantly reduce the time and code size of our `PRINTcipher` implementations (+27% speed up).
III Merging the Key Addition layer with the Virtual S-boxes
Our first optimization combined the Keyed Permutation with the Substitution layer and resulted in four virtual S-boxes V_0 to V_3 as proposed by the designers of `PRINTcipher` in [14]. A possibility to further speed up the `PRINTcipher` implementation is to change the processing order of the `PRINTcipher` layers of one round and merge the Key Addition layer with the Combined Substitution

layer. Instead of performing the Key Addition layer prior to the Permutation layer we are swapping their order and perform the Permutation layer first. This step requires us to permute the *sk1* bits once, to ensure that we are applying the correct *sk1* bits to the *cipher state* bits. Since the *Round Constant Addition* and *Key Addition* layer both just perform `XOR` operations that do not effect the structure of the cipher or key bits, we can also swap their order. The processing order within one `PRINTcipher` encryption round is now:

Permutation → RC Addition → Key Addition → Combined Substitution

We can now combine the *Key Addition* with the *Combined Substitution* layer in the following way: All entries of each of the four V_i are added with all possible 3-bit values of *sk1*. As a result we have to store 32 virtual S-boxes in the ROM. By this, we can execute the *Keyed Permutation*, *Substitution*, and *Key Addition* layers for each 3-bit *cipher state* entry at once by a single table lookup (+42% speed up).

Throughout the optimization process from a naïve to a speed-optimized implementation a significant time reduction of almost 66% was achieved for the encryption routine of PC-48, while reducing its code size by around 11%. For the speed optimized decryption routine of PC-48, we reduced the time and code size by around 63% and 6%.

5.3 Code-size optimization

Starting from the 3-bit word optimization we have developed code-size optimized implementations for the PC-48 using similar techniques as for the `AES` routines. The code optimized implementation scales nicely, since it needs around double the time to encrypt/decrypt a 48 bit message in comparison to its speed-optimized counterpart, with a code size that is only 39% of the speed-optimized implementation.

5.4 Bit-sliced implementation

Introduction to Bit-slicing The term bit-slicing describes an implementation technique, presented by Eli Biham in the year 1997 [6]. The idea is to change the order of the bits in such a way, that an n-bit CPU can be viewed as a collection of n one-bit processors. The processor is hereby viewed as a Single-Instruction-Multiple-Data (SIMD) processor [13]. By this, it is possible to perform single-bit operations, such as permutations, `XOR`s, and so forth, in parallel which can considerably reduce the time consumption of an implementation. The interested reader is referred to [6], [9], and [13] for further details on bit-slicing and the SIMD architecture.

Bit Order The PC-48 *cipher state* consists of 48 bits, stored in 12 words. The MARC4 word size of 4-bit allows us to compute 4 slices at a time. For these, a total of 48 4-bit words are needed. Before we can start the actual enciphering or deciphering of the four slices it is necessary to reorder them first. The lowest bit of the first cipher/*sk1* state is stored at the lowest position of $state_0/subkey_0$. The bit on position 1 is written to the lowest position of state_1 and so forth. This is done for all of the 4 bit positions of each of the 48 4-bit words.

The 32 *sk2* bits are hereby not written to 32, but 16 words (pkey_0 to pkey_{15}), each containing two *sk2* bits, to allow an efficient implementation of the Keyed Permutation layer. The initial reordering of the 48 *cipher state/sk1* bits and the 32 *sk2* bits and their inverse reordering at the end of the encryption needs a total 3,285 cycles and consumes 1,437 bytes of code size.

Key Addition The Key Addition layer operation is equivalent to its counterpart of the speed-optimized `PRINTcipher` layer with the only difference that we have a total of 48 state and keywords to process.

Permutation Layer Due to the fact that the corresponding bits of each slice are all stored in the same *cipher state*, we can implement the Permutation layer by just pushing all the 46 *cipher states* we want to permute on the stack (state_1 to state_{46}) and afterward write them back to their permuted positions[1].

Round Counter The 6-bit round counter ($x_5x_4x_3x_2x_1x_0$) is added to the lowest 6 *cipher states*. The formulas to implement the PC-48 Round Counters for the encryption and decryption routines are introduced in [14] and presented as follows:

Encryption:	Decryption:
$t = 1 + x_5 + x_4$	$t = 1 + x_5 + x_0$
$x_i = x_i - 1$	$x_i = x_i + 1$
$x_0 = t$	$x_5 = t$

Keyed Permutation For the Keyed Permutation layer, three *cipher state* bits are pushed on the stack, permuted by the 2-bit value of their corresponding permutation key and afterward written back.

Substitution Layer For the non bit-sliced implementation, the substitution layer is realized by table lookup operations. A table lookup implementation for the bit-sliced implementation would be very inefficient, since input bits for the S-boxes would have to be combined using bits of different variables and therefore involve various manipulations of the *cipher states* to address the correct bits for each substitution. For an efficient implementation of the substitution layer, it is necessary to find an alternative to the use of S-boxes. A suitable approach is to convert the `PRINTcipher` encryption and decryption S-boxes to their algebraic normal forms and operate on them [6]. The translation from the S-boxes to their algebraic normal forms was conducted with the help of the *Sbox2ANF* Python script [30]. The algebraic normal form for the `PRINTcipher` encryption routine is hereby presented in (2).

The 3-bit S-box input x that is mapped to $S[x]$ consists of the concatenation of the bits x_2, x_1 and x_0, where x_2 = MSB of x and x_0 = LSB of x and $S[x] = S[x_2]|S[x_1]|S[x_0]$.

Algebraic normal form representation of the encryption S-box:

$$S[x_0] = x_0 \oplus x_1 \oplus x_2 \oplus x_1 \cdot x_2 \tag{2}$$

[1] The two *cipher state* words state_0 and state_{47} are not pushed on the stack since their entries remain unchanged.

$$S[x_1] = x_1 \oplus x_2 \oplus x_0 \cdot x_2$$
$$S[x_2] = x_0 \cdot x_1 \oplus x_2$$

The implementation results for the bit-sliced PRINTcipher implementation (including the time and space consumption for the initial and inverse reordering of the *cipher/key states*) are presented in Table 7 (PC-48-B).

6 Results and Discussion

In this section we are going to compare our implementations of the AES and PRINTcipher algorithms on the MARC4 with all other published crypto implementations for that microcontroller. These are to our best knowledge, only HUMMINGBIRD [12], which is already broken [17], and PRESENT [29]. We focus on three main optimization goals for cryptographic implementations on constrained environments: achieving an either small code size, high throughput, or high throughput with respect to a small code size implementation. Table 5 compares our results to previous work with respect to key length, block length, code size, cycles per block, cycles per byte, throughput and code cycles. We also provide estimated energy per bit consumptions based on the following calculation: Energy/bit $= \frac{1.8V*200\mu A*cycles}{1MHz*blocksize}$.

6.1 Notation

We use the following notation: E denotes encryption, D denotes decryption, ED denotes combined encryption and decryption, ED/E stand for the encryption routine of the combined ED implementation and ED/D for the decryption routine of it, respectively; an appended -A stands for code-size optimized, -S for speed optimized; -P/-N denote precomputed/non precomputed last round key for the decryption routine; -B denotes a bit-sliced implementation. For example, AES-ED/E-A-P denotes an AES encryption routine of the combined encryption and decryption implementation (ED/E) that is code-optimized (A) and uses precomputed last round keys for the decryption routine (P).

6.2 Code-Size Optimized

Compared with PRESENT and HUMMINGBIRD, our PC-48-A implementations need around 42% and 68% less code size for the encryption and around 48% and 68% less code size for the decryption routines, respectively. While a comparison of the decryption routines of our code-size optimized AES implementation and HUMMINGBIRD shows that both implementations have roughly the same code size (1,637 and 1,559 bytes, respectively), a significant difference can be found for the encryption routines, where our AES encryption routine needs 34.03% less code size compared to HUMMINGBIRD.

6.3 Speed Optimized

For some applications with very few data to encrypt it might be important to have a low encryption/decryption time for a single message, which can be

measured by cycles per block. Naturally, it depends on the block size of the block cipher used whether the message can fit into one block or not. Since, AES, HUMMINGBIRD, PRESENT, and PRINTcipher have different block sizes, we also provide cycles per byte and the throughput at 1 MHz (in kilo bits per second) in Table 5 to have a fair comparison. While our speed optimized AES implementations are among the ones with the largest code size, they are also the fastest implementations. For both optimization goals, the relative rankings for HUMMINGBIRD[1] and PRESENT are somewhat similar in comparison with AES and PRINTcipher implementations. It is noteworthy to stress that our code size optimized AES encryption routine is the second fastest implementation, while having a reasonable code size.

6.4 Throughput per Code Size Optimized

Inspired by the time-area product metric for hardware implementations, we introduce a new metric for software implementations, which binds the time to the code size: code cycles $= \frac{\text{block size}}{\text{cycles/block} \cdot \text{code size}}$.

Table 5. Comparison of the encryption and decryption routines for AES, HUMMINGBIRD, PRESENT, and PRINTcipher. See Section 6.1 for notation.

Encryption routine		Key size (bit)	Block size (bit)	Code size (byte)	Cycles per Block	Cycles per byte	Tp @ 1 Mhz (kbit/s)	Energy/bit @ 1 Mhz (nJ)	Code Cycles ($\cdot 10^6$)	Rel. Code cycles	Rel. Tp	Rel. Code size
PC-48	E-A	80	48	490	62,490	10,415	0.76	469	1.57	0.33	0.09	1.00
PC-48	E-S	80	48	1,250	30,079	5,013	1.60	226	0.68	0.27	0.20	2.55
PRES. [29]		80	64	841	55,734	6,967	1.15	314	0.94	0.29	0.14	1.72
PC-48	E-B	80	192	2,788	67,670	2,819	2.84	127	1.02	0.22	0.35	5.69
AES	E-A	128	128	1,143	23,828	1,489	5.37	67	4.70	1.00	0.67	2.33
AES	E-S	128	128	2,747	15,848	991	8.08	45	2.94	0.63	1.00	5.61
PC-96	E-A	160	96	761	245,370	20,448	0.39	920	0.51	0.11	0.05	1.55
PC-96	E-S	160	96	2,397	129,936	10,828	0.74	487	0.31	0.07	0.09	4.89
Humm. [12]		256	16	1,532	5,773	2,887	2.77	130	1.81	0.39	0.34	3.13
Decryption routine												
PC-48	D-A	80	48	493	63,450	10,575	0.76	476	1.53	0.60	0.12	1.00
PC-48	D-S	80	48	1,280	31,375	5,229	1.53	235	1.20	0.47	0.25	2.60
PRES. [29]		80	64	945	65,574	8,197	0.98	369	1.03	0.41	0.16	1.92
PC-48	D-B	80	192	2,775	67,046	2,793	2.86	126	1.03	0.41	0.46	5.63
AES	D-A-P	128	128	1,637	30,887	1,930	4.14	87	2.53	1.00	0.67	3.32
AES	D-S-P	128	128	3,343	20,736	1,296	6.17	58	1.85	0.73	1.00	6.77
PC-96	D-A	160	96	794	247,482	20,624	0.39	928	0.49	0.19	0.06	1.61
PC-96	D-S	160	96	2,408	130,992	10,916	0.73	491	0.30	0.12	0.12	3.88
Humm. [12]		256	16	1,559	5,212	2,606	3.07	117	1.97	0.78	0.49	3.16

6.5 Discussion

As Figure 4 points out, all of our AES implementations are faster than any other implemented cipher on the MARC4. The fastest implementation is AES-E-S. The second fastest encryption only implementation is AES-E-A. It needs around 50% more time to process a single byte (1,489 cycles/byte), but just

[1] The throughput of the HUMMINGBIRD encryption and decryption routines, obtained from [12] are without the initialization phase of 22,949 cycles.

around 42% (1,143 bytes) of the code size of AES-E-S. A further comparison of AES-E-A with the other implemented ciphers shows that it has a very good overall performance.

In contrast to the broken HUMMINGBIRD, AES-E-A is not only around 48% faster but also around 25% smaller. Despite an overhead in code size of around 36% of AES-E-A in comparison with PRESENT, we need around 79% less time to encrypt a single byte. An even higher performance advantage can be seen when comparing AES-E-A with the implementations of PC-48 and PC-96. The code size of AES-E-A in comparison to PC-48-E-A is increased by 130% (50% compared to PC-96-E-A), but its time is significantly reduced by a factor of almost 6 (around 12.7 compared to PC-96-E-A). The advantages of AES-E-A over the speed optimized PC-48 and PC-96 encryption routines are even more obvious, since they are both slower and need more code size. A similar picture can be drawn for the decryption routines, where both, our AES speed and code size optimized routines are again the fastest among all implementations.

Figure 4 also presents the combined encryption and decryption implementations for AES and PRINTcipher. Their execution time is roughly the same as the ones of their non combined counterparts. The encryption and decryption routines have various synergies that can be exploited to efficiently include both routines in a single implementation. The savings of all combined implementations over their encryption/decryption only routines are presented in Table 6 in the Appendix.

7 Conclusions

We have implemented standardized cryptography on 4-bit microcontrollers for the first time and also provided optimized and timing attack resistant implementations of the AES that are faster than any previously published implementations. Besides, the energy consumption of only 45 nJ per bit is surprisingly low. We hope that our implementations open the door for a wide variety of security functionalities on one of the most abundant computing platforms. Especially legacy devices already embedded in myriads of every-day objects and deployed over the last couple of decades could benefit from additional security functionalities, such as secure code update, authentication, confidentiality etc. Applying countermeasures to secure our implementations against various side-channel attacks, especially Simple Power Analysis (SPA) and Differential Power Analysis (DPA) [15] was out of the scope of this work, but clearly is a next step for future work. To the best of our knowledge, no previous work has been conducted on asymmetric key cryptography for 4-bit MCUs. Due to its small key size, Elliptic Curve Cryptography (ECC) [19] is a promising asymmetric key candidate. We believe that ECC with a 160-bit curve can be implemented on a 4-bit MCU.

References

1. Atmel Corporation. *MARC4 4-Bit Microcontrollers - Programmers Guide*, 2004.
2. Atmel Corporation. *Zero-power Microcontrollers for Low-power and High-temperature Applications*, 2004.
3. Atmel Corporation. *Flash Version for ATAR080, ATAR090/890, ATAR092/892, and ATAM893-D*, 2005.
4. Atmel Germany GmbH. *MTP Programmer ICP I.*
5. B. Kernighan and D. Ritchie. *The C Programming Language Second Edition.* Prentice-Hall, Inc., Upper Saddle River, New Jersey, USA, 1988.
6. E. Biham. A fast new DES implementation in software. In *Proceedings of FSE 1997*, FSE '97, Volume 1267 of LNCS, pages 260–272. Springer-Verlag, 1997.
7. A. Bogdanov, G. Leander, L. R Knudsen, C. Paar, A. Poschmann, M. J.B Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT - an Ultra-Lightweight block cipher. In *Proceedings of CHES 2007*, Volume 4727 of LNCS, pages 450–466. Springer-Verlag, 2007.
8. A.W. Burks, D.W. Warren, and J.B. Wright. An analysis of a logical machine using parenthesis-free notation. *Mathematical Tables and Other Aids to Computation*, 8, No.46(46):53–57, April 1954.
9. C. Paar. *Lecture Notes: Imlementation of Cryptographic Schemes 2.* Chair for Embedded Security, Ruhr-Universität Bochum, August 2010.
10. J. Daemen and V. Rijmen. *The Design of Rijndael: AES. The Advanced Encryption Standard.* Springer-Verlag, April 2002.
11. D. Engels, X. Fan, G. Gong, H. Hu, and E.M. Smith. Ultra-lightweight cryptography for low-cost RFID tags: Hummingbird algorithm and protocol. Technical report, Centre for Applied Cryptographic Research (CACR), 2009.
12. X. Fan, H. Hu, G. Gong, E.M. Smith, and D. Engels. Lightweight implementation of Hummingbird cryptographic algorithm on 4-bit microcontrollers. International Conference for Internet Technology and Secured Transactions, 2009. ICITST 2009, pages 1–5, Nov. 2009.
13. M.J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.
14. L. Knudsen, G. Leander, A. Poschmann, and M.J.B. Robshaw. PRINTcipher: A Block Cipher for IC-Printing. In Stefan Mangard and Francois-Xavier Standaert, editors, *Proceedings of CHES 2010*, Volume 6225 of LNCS, pages 16–32. Springer-Verlag, 2010.
15. P.C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In Michael J. Wiener, editor, *CRYPTO '99*, Volume 1666 of LNCS, pages 388–397. Springer-Verlag, 1999.
16. Gregor Leander, Mohamed Ahmed Abdelraheem, Hoda AlKhzaimi, and Erik Zenner. A cryptanalysis of printcipher: the invariant subspace attack. In *Proceedings of the 31st annual conference on Advances in cryptology*, CRYPTO'11, pages 206–221, Berlin, Heidelberg, 2011. Springer-Verlag.
17. M-J O.Saarinen. Cryptanalysis of Hummingbird-1. In *Proceedings of FSE 2011*, volume 6733 of *LNCS*, pages 328 – 341. Springer-Verlag, 2011.
18. Microchip Technology Inc. *AN821: Advanced Encryption Standard Using the PIC16XXX*, 2002.
19. V.S. Miller. Use of elliptic curves in cryptography. In *CRYPTO '85*, Volume 218 of LNCS, pages 417–426. Springer-Verlag, 1986.
20. National Institute of Standards and Technology. Announcing the Advanced Encryption Standard (AES). Federal Information Processing Standards (FIPS) Publication 197, November 2001.

21. Axel Poschmann. *Lightweight Cryptography - Cryptographic Engineering for a Pervasive World*. Number 8 in IT Security. Europäischer Universitätsverlag, 2009. Published: Ph.D. Thesis, Ruhr University Bochum.
22. S. Rinne, T. Eisenbarth, and C. Paar. Performance Analysis of Contemporary Light-Weight Block Ciphers on 8-bit Microcontrollers. In *ecrypt workshop SPEED*, 2007.
23. I. Rouf, R. Miller, H. Mustafa, T. Taylor, S. Oh, W. Xu, M. Gruteser, W. Trappe, and I. Seskar. Security and privacy vulnerabilities of in-car wireless networks: a tire pressure monitoring system case study. In *Proceedings of USENIX'10*, pages 323–338, Berkeley, CA, USA, 2010.
24. Senate and House of Representatives of the United States of America. Transportation Recall Enhancement, Accountability, And Documentation (TREAD) Act, 2000.
25. W. Stallings. *Cryptography and network security: principles and practice*. The William Stallings Books on Computer and Data Communications. Pearson/Prentice Hall, Upper Saddle River, New Jersey, USA, 2006.
26. H. Sundmaeker, P. Guillemin, P. Friess, and S. Woelfflé. Vision and challenges for realising the internet of things. *Cluster of European Research Projects on the Internet of Things, European Commision*, 2010.
27. Forth Standards Team. *FORTH-83 Standard*. Mountain View Press, 1983.
28. The European Parliament - EC 661/2009. Concerning type-approval requirements for the general safety of motor vehicles, their trailers and systems, components and separate technical units intended therefor, July 2009.
29. M. Vogt, A. Poschmann, and C. Paar. Cryptography is Feasible on 4-Bit Microcontrollers - A Proof of Concept. In *International IEEE Conference on RFID*, pages 267–274, Orlando, USA, April 2009.
30. B. Zhu. Shanghai Jiao Tong University, Cryptography and Information Security Lab, A Simple Python Script for Translating Sbox to ANF Boolean Functions. http://cis.sjtu.edu.cn/index.php/A_Simple_Python_Script_for_Translating_Sbox_to_ANF_Boolean_Functions.

Appendix

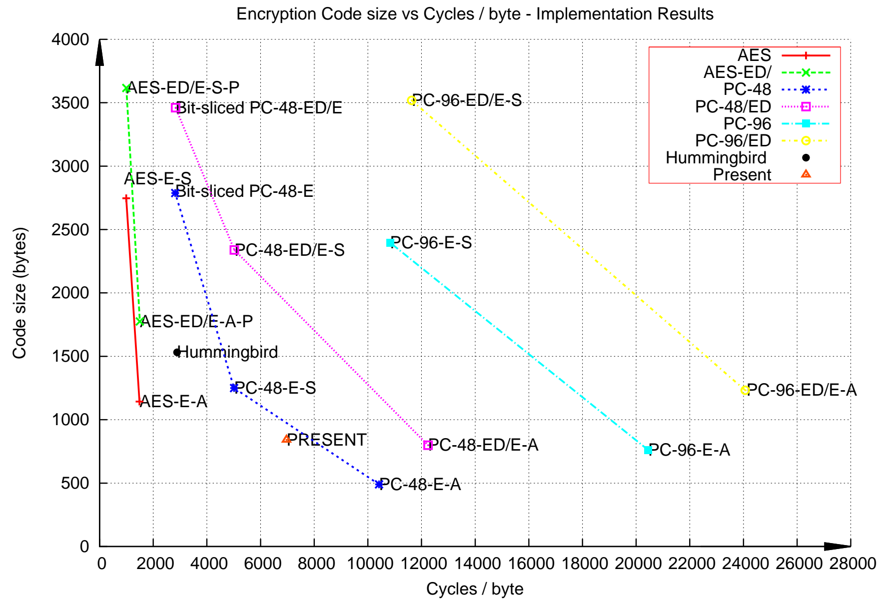


Fig. 4. Code size vs cycles per byte for the speed and code size optimized encryption routines for `AES`, `HUMMINGBIRD`, PC-48, PC-96, and `PRESENT`.

Table 6. Comparison of the code sizes of the combined encryption and decryption routines and the accumulated code sizes of combining their encryption/decryption only routines.

	Opt.	Routine	Code size (bytes)	Sum	Comb.	Di . (%)
`AES`	Speed	Enc.	2,747	6,090	3,616	-40.62
		Dec.	3,343			
	Code size	Enc.	1,143	2,780	1,775	-36.15
		Dec.	1,637			
PC-48	Speed	Enc.	1,250	2,530	2,339	-7.55
		Dec.	1,280			
	Code size	Enc.	490	983	799	-18.72
		Dec.	493			
PC-96	Speed	Enc.	2,397	4,805	3,519	-26.76
		Dec.	2,408			
	Code size	Enc.	761	1,555	1,232	-20.77
		Dec.	794			
PC-48-B		Enc.	2,788	5,563	3,461	-37.79
		Dec.	2,775			

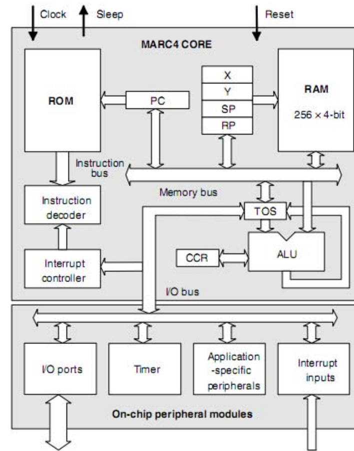


Fig. 5. MARC4 core [1].

Table 7. Time, code size, and throughput for the speed and code-size optimized, single and combined implementations. E = encryption only, D = decryption only, ED = combined encryption and decryption, ED/E = encryption routine of the combined ED implementation, ED/D = the decryption routine of ED; -A = code-size optimized, -S = speed optimized; -P/-N = precomputed/non precomputed last round key for the decryption routine; -B = bit-sliced implementation.

Algorithm	Optimization	Time (cycles)	Code size (bytes)	Tp @500 Khz (kbit/s)	Max Stack depth Exp. (S0)	Ret. (R0)
AES	E-S	15,848	2,747	4.04	33	16
	E-A	23,828	1,143	2.69	33	24
	D-S-P	20,736	3,343	3.09	33	16
	D-S-N	22,944	3,509	2.79	33	16
	D-A-P	30,887	1,637	2.07	33	24
	D-A-N	34,010	1,712	1.88	33	16
	ED/E-S-P	15,977	3,616	4.01	33	20
	ED/E-S-N	15,977	3,679	4.01	33	20
	ED/D-S-P	20,801	3,616	3.08	33	20
	ED/D-S-N	23,051	3,679	2.78	33	20
	ED/E-A-P	24,012	1,775	2.67	33	24
	ED/E-A-N	24,012	1,827	2.67	33	24
	ED/D-A-P	31,054	1,775	2.06	33	24
	ED/D-A-N	34,432	1,827	1.86	33	24
PC-48	E-S	30,079	1,250	0.80	19	5
	E-A	62,490	490	0.38	19	11
	D-S	31,375	1,280	0.76	19	5
	D-A	63,450	493	0.38	19	11
	ED/E-S	30,148	2,339	0.80	24	6
	ED/D-S	31,450	2,339	0.76	24	6
	ED/E-A	73,464	799	0.33	24	12
	ED/D-A	74,423	799	0.32	24	12
PC-48-B	E	67,670 (4 slices)	2,788	1.42	51	16
	D	67,046 (4 slices)	2,775	1.43	51	16
	ED/E	67,963 (4 slices)	3,461	1.41	56	20
	ED/D	67,964 (4 slices)	3,461	1.41	56	20
PC-96	E-S	129,936	2,397	0.37	35	5
	E-A	245,370	761	0.20	35	11
	D-S	130,992	2,408	0.37	35	5
	D-A	247,482	794	0.19	35	11
	ED/E-S	139,568	3,519	0.34	40	6
	ED/D-S	140,625	3,519	0.34	40	6
	ED/E-A	288,800	1,232	0.17	40	12
	ED/D-A	289,370	1,232	0.17	40	12

Elliptic Curve Cryptography in JavaScript

Laurie Haustenne, Quentin De Neyer, and Olivier Pereira*

Université catholique de Louvain
ICTEAM – Crypto Group
B-1348 Louvain-la-Neuve – Belgium

Abstract. We document our development of a library for elliptic curve cryptography in JavaScript. We discuss design choices and investigate optimizations at various levels, from integer multiplication and field selection to various fixed-based EC point multiplication techniques.
Relying on a small volume of public precomputed data, our code provides a speed-up of a factor 50 compared to previous existing implementations. We conclude with a discussion of the impact of our work on a concrete application: the Helios browser-based voting system.

1 Introduction

Current browsers offer fairly limited support for performing cryptographic operations on the client-side of web applications. The support of the TLS/SSL protocols enables secure client-server communications, but these protocols can only be useful in settings where the server is trusted by the client, and the implemented cryptographic libraries are not exposed for other uses by web applications.

There are numerous applications, however, in which it is not desirable to ask web application users to trust a server. E-voting is one of them: encrypting ballots on the client side using a key that does not allow the server to decrypt the vote content not only limits the trust that the voters need to place in the voting server, but also substantially decreases the incentives for an attacker to hack the voting server, since the server then only sees information that it cannot interpret. While e-voting was the initial motivation for our work, being able to run cryptographic protocols on the client-side also offers very interesting perspectives for many other web applications, e.g., browser synchronization [18] or auctions [6].

The JavaScript engine appears to be the most convenient choice for computing on the client side of web applications: a JavaScript engine is provided with all major browsers. The interest of a cryptographic library in JavaScript is however not limited to browsers, as JavaScript is also available and increasingly used in other contexts in which cryptography is useful: one can think for instance about documents such as PDF or OpenOffice files, but also about server-side environments like Node.js.

* Olivier Pereira is a Research Associate of the Belgian Funds for Scientific Research F.R.S.-FNRS.

These various applications indicate that cryptographic libraries in JavaScript would be very useful, and it is therefore not surprising that various such libraries have been proposed already [4, 10, 20–22]. Even though some of these libraries offer some level of support for ECC, the design criteria of these libraries are essentially undocumented.

Running cryptographic operations in JavaScript in a browser presents constraints that are quite different from those appearing in classical cryptographic applications. On the one hand, despite tremendous improvements during the last two years, the performance of JavaScript code remains extremely low compared to optimized compiled code executed on the same computer. On the other hand, compared to other slow platforms like smart-cards, browsers offer an amount of memory that is larger by orders of magnitude. Such constraints motivated our independent study.

Our contributions. We present our development of elliptic curve cryptographic primitives in JavaScript, offering the first documented study on this topic. In particular:

- We compare several integer multiplication algorithms, determining when the grade-school multiplication technique becomes outperformed by asymptotically more efficient algorithms like the Karatsuba multiplication.
- We compare the performances of operations in various finite fields (binary, prime order, OEF).
- We define new NIST-style elliptic curves that are optimized for JavaScript implementation.
- We compare several fixed-base point multiplication algorithms, and determine which ones are the most efficient as a function of the number of points that one desires to store during precomputation.

Our implementation of EC point multiplication is more than 50 times faster than the most efficient stable one [22], offering comparable security levels. We stress however that this implementation does not rely on precomputation, while we rely on a small volume of public precomputed data.

The remaining parts of this document are organized as follows. In Section 2, we document our experiences with integer multiplication and various field operations, leading to the selection of new curves. In Section 3, we discuss various point multiplication strategies. We then discuss applications of our work in the context of voting protocols in Section 4, and conclude.

2 Field operations

We discuss the results of our investigation of arithmetic in prime fields. Our investigation however also involved binary and optimal extension fields, but they showed to be less efficient for our purpose. A summary of our results for these other types of fields is provided at the end of this section, and a detailed account is available in a separate report [12].

2.1 Big integer representation

JavaScript does not offer any support for the manipulation of big integers: one single numeric literal exists [14], and numbers are represented as IEEE-754 doubles.

In order to tackle this limitation, various strategies have been adopted. One possibility is to use the LiveConnect feature of web browsers that enables JavaScript to intercommunicate with a Java Virtual Machine: support for big integers and for basic operation on these integers is then provided by the JVM. This is the approach that was adopted in the Helios voting system for instance [1, 2, 8]: Helios performs big integer manipulations like modular exponentiation through LiveConnect, but all higher level algorithms (ElGamal, ...) are implemented in JavaScript directly. While this allows taking benefit of the JVM, this approach is also fairly limited in terms of algorithmic efficiency since only basic modular exponentiation is available: more efficient algorithms, for fixed-based exponentiation or multi-exponentiation for instance, are therefore not used.

Another approach, which became practical very recently due to the tremendous performance improvements of the JavaScript engines available in the major browsers, is to develop a pure JavaScript big integer library (educational implementations of such libraries have however been available for quite a long time). This is the approach we want to adopt here, as it removes the dependence of any external browser plug-in.

We take as our starting point the JSBN library by Tom Wu [22], which is, to the best of our knowledge, the most advanced big integer JavaScript library. In this library, big integers are stored as arrays of smaller integers, the length of which depends on the detected browser. Indeed, while JavaScript exposes signed 32 bits integers, considerable slowdowns appear when one computes with integers that come close to these 32 bits, as demonstrated in Table 1. Our experiments show that using arrays of 28 bit integers provides the most efficient results that are usable on the major browsers. For these measurements, we used an average netbook: Intel Core 2 Solo processor SU3500 (1.4 GHz) running Windows Vista. The browser version were as follows: FFX: Mozilla FireFox 4.0.1; IE: Internet Explorer 9.0.1; CHR: Google Chrome 11.0.696.71; SAF: Safari 5.0.5.

Table 1. Timings for multiplication in μs

	FFX	IE	CHR	SAF
28 bit words	5.3	7.6	3.2	8.2
30 bit words	13	16	4.3	12

As a result, in order to be able to exploit the integer representation in the choice of the field in which we compute, we decided to only use the 28 bits representation instead of having an adaptive integer representation according to the browser type.

2.2 Integer Multiplication

JSBN uses long (or grade-school) multiplication. It was not clear however whether performance improvements could come from using asymptotically more efficient algorithms. Therefore, we implemented the classical Karatsuba algorithm [15], which allows moving from $\mathcal{O}(n^2)$ complexity to approximately $\mathcal{O}(n^{1.585})$ complexity.

We provide an typical depiction of our experiments results in Figure 1, based on the Safari browser. As can be observed on this picture, Karatsuba multiplication becomes efficient for integers that are more than 1300 bits long. This bound is however strongly dependent of the browser that is used: on Firefox 3.6.23, the switch happens for 600 bit integers, while it happens only for integers around 1800 bit long on Chrome 14 (on the same Ubuntu laptop).

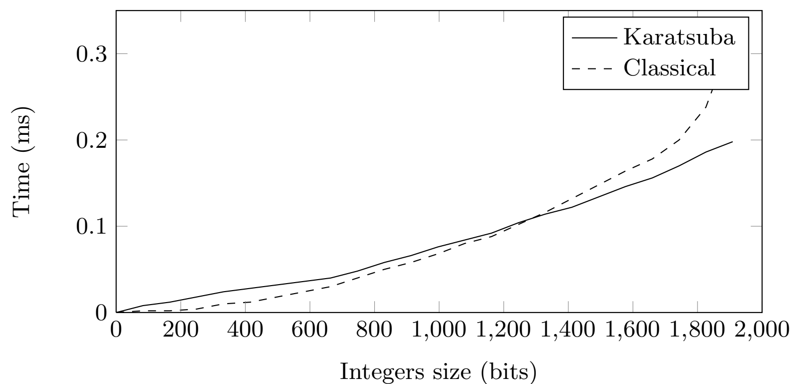


Fig. 1. Karatsuba multiplication becomes efficient around 1300 bit integers on our netbook running Safari.

While these integer lengths remain considerably longer than the integers we will manipulate for elliptic curve operations, this supports the adoption of Karatsuba multiplication (or of variants of it, e.g., Knuth or Toom-Cook) if one wishes to perform operations on larger integer. This might happen for cryptographic protocols that rely on the hardness of factoring, e.g., RSA encryption which was part of the motivations for the JSBN library, or Paillier encryption, but also if one desires to work in subgroups of $\mathbb{Z}_p^*$ for instance.

Nevertheless, for our purpose, we adopted standard grade-school multiplication. It would be interesting to see whether its efficiency could be further improved by using scanning techniques such as those proposed in [13] for instance.

2.3 Modular reductions

While computing in a prime field $\mathbb{F}_p$, reduction modulo p is a common and potentially expensive operation. In order to mitigate the cost of modular reductions, various ECC standards recommend using specially chosen primes that facilitate those reductions. For instance, the NIST prime p_{224} is equal to $2^{224} - 2^{96} + 1$ [19], in which we can observe that both 224 and 96 are multiples of 32, an expected word size for most implementations.

This 32-bit oriented choice is however clearly not optimal in our case, since our computation is based on 28-bit words. As a result, we looked for similar pseudo-Mersenne primes and found that $p^{28}_{224} = 2^{224} + 2^{140} + 2^{56} + 1$ is the prime integer with the fewest non-zero coefficients b_i in the set of integers of the form $2^{224} + b_7 2^{196} + b_6 2^{168} + b_5 2^{140} + b_4 2^{112} + b_3 2^{84} + b_2 2^{56} + b_1 2^{28} + b_0$ with $b_i \in \{-1, 0, 1\}$.

The JSBN library does not take into account the specific structure of the modulus when it performs reduction, and therefore does not exhibit any performance change when using p^{28}_{224}instead of p_{224}. Substantial changes appear, though, when using a specific modular reduction function, tailored for p^{28}_{224}.

The resulting performance of the prime field operations is given in Table 2, in which all timings include modular reduction. The squaring and inversion implementations are those from the JSBN library (except for the reductions), that is, the squaring is based on [17, Algorithm 4.16], and the inversion on [17, Algorithm 4.61].

Table 2. Timings for modular prime field operations in μs

	FFX	IE	CHR	SAF
addition	0.28	0.34	0.13	0.41
multiplication	**5.9**	**7.7**	**3.4**	**10**
squaring	4.9	6.2	3	8.5
inversion	900	1050	550	1100

This table shows fairly important discrepancies between the browsers. These values can however change substantially with browser updates. As expected, the inversion operation is by far the most expensive.

2.4 Result outline in other fields

Binary field arithmetic is typically slower than prime field arithmetic in software since integer multiplication is directly provided by processors and more efficient than repeated bitwise operations. As a result, multiplication showed to be on average 10 times slower on binary fields than on prime fields.

We also investigated optimal extension fields (OEF) [3]. These provide performances that are slightly slower than those of prime field arithmetic, except for inversion which is a bit more than 10 times faster.

Our efficiency measurements are summarized on a logarithmic scale in Figure 2 and detailed in [12]. The relative performance of operations in these fields is essentially in line with traditional results appearing in the literature for software implementation [11].

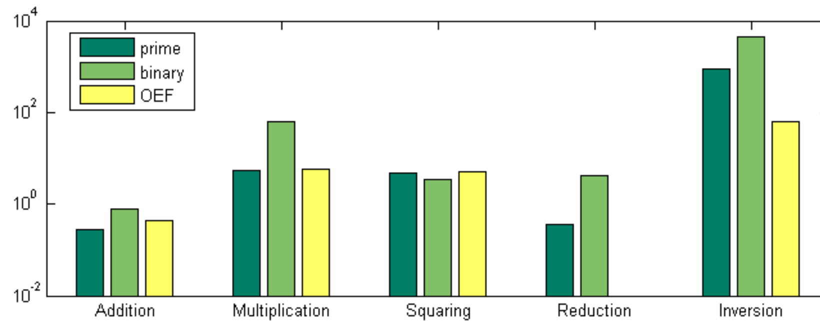


Fig. 2. Timing comparison of different field operations, in μs.

Since the main benefit of OEF, i.e., inversion, is not of interest for our applications, and since elliptic curves on OEF remain more experimental (they do not appear in the main ECC standards), we decided to adopt the prime field $\mathbb{F}_p$with our specially chosen prime $p_{224}^{28}$for the rest of our work.

3 Curve selection and operations

3.1 Curve selection

We selected a NIST-style pseudo-random curve [19] for p_{224}^{28}, that is, a curve of the form $E : y^2 = x^3 - 3x + b \bmod p_{224}^{28}$ of prime order n with base point (G_x, G_y). Our curve has the following parameters:

- $b = 13675174559945691270660091572714686899958220410447750995672981802966$
- $n = 26959946667150639794667016480816204352639545292933842228829888218579$
- $G_x = 15022218326251922240529090945393257414013962585837380057002596801053$
- $G_y = 24039939147593575364998439277103263076917813793017813680357106378307$

3.2 Choice of coordinates

The choice of a specific point representation has a substantial impact on the efficiency of point addition and doubling operations. Following the analysis provided by Hankerson et al. [11, Table 3.3], we decided to store points in affine coordinates, which is also the most efficient from a memory point of view, and to keep intermediate computation results in Jacobian coordinates. Using the algorithms from [11], the performance of point addition and doubling appear in Table 3. Investigating more recent techniques (e.g., those described in [5]), would certainly provide new improvements.

3.3 Point multiplication

The cryptographic protocols we consider involve a potentially large number of point multiplications, but only with a very small number of fixed bases (2 for ElGamal encryption for instance). Since a fairly large amount of memory is available in browsers for precomputation, exploring fixed-base point multiplication algorithms is particularly promising.

These algorithms could be used in two ways: either the browser performs precomputation himself and uses it later, or the precomputation is performed on the server side and provided to the browser as part of the web application (it could be certified and provided as part of the public key for instance).

We decided to adopt the second option, as requiring the browser to download a few extra kilobytes of public information is not an issue in our context. To fix the ideas, we decided to allow a volume of precomputed data of around 50 kB per base point, which corresponds to the volume of a small photograph. As we will see, a 10 times smaller volume of precomputed data already provides a very substantial acceleration, and nothing prevents to enable browser-based precomputation in bandwidth constrained environments (though different algorithmic choices should probably be made in that case).

We then explored various fixed point multiplication techniques, surveyed in [11] for instance: fixed-base windowing [7] based on standard and NAF representation and comb methods based on one or two precomputation tables (these methods are also detailed in [12]).

The relative complexity of these fixed point multiplication techniques is described in Figure 3, where the point doubling/addition ratio comes from our measurements of Table 3. We can observe that the two windowing techniques do not provide any extra benefit when more than 70 points are stored, while the two comb methods keep improving, the one based on two tables (comb2) being the most efficient. Generalization of the comb approach to more tables were also explored [16], but do not provide any improvement for the data volumes we have in mind.

Our limit of 50 kB of storage allows us to exploit more than 500 precomputed points. In this case, the complexity of a point multiplication is slightly lower than 50 point doubling operations. We observe that, by decreasing the number of stored points by a factor 10, the point multiplication complexity increases by a factor less than two, which might still be convenient if one desires to decrease the volume of precomputed data.

3.4 Point multiplication efficiency

The performance of our point operations is given in Table 3, based on the same computer and browser versions as before.

As before, those results are quite sensitive to the browser and computer that are used. For instance:

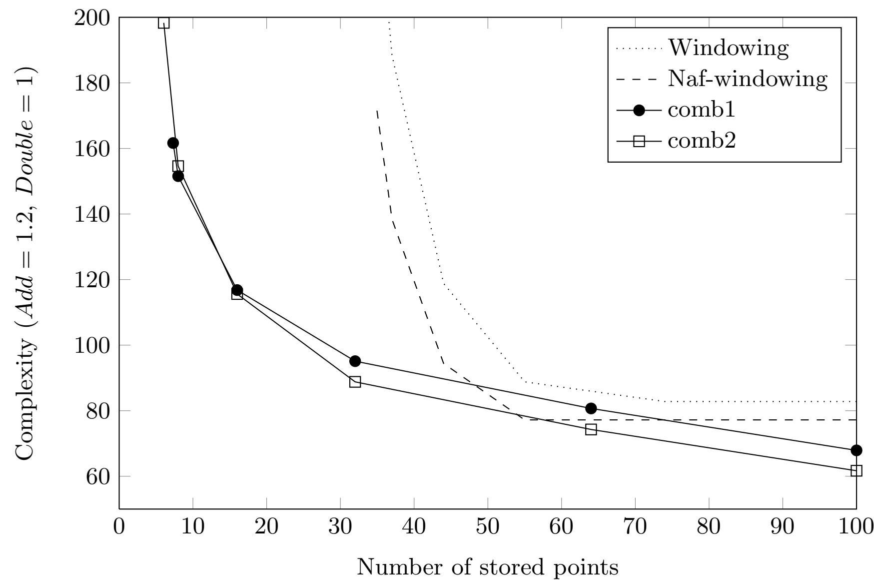


Fig. 3. Complexity of point multiplication as a function of the number of stored points. Windowing methods reach a minimum around 60 points, while comb methods keep improving.

Table 3. Timings for EC point operations in μs

	FFX	IE	CHR	SAF
addition	83	95	55	120
doubling	73	81	49	104
multiplication	**3300**	**3300**	**1900**	**4200**

– on a recent laptop (Intel Core i7-640M Processor at 2.8GHz) and using Chrome 14, a point multiplication operation takes $550\mu s$, which is already almost 4 times faster than the time reported for Chrome in Table 3,
– on an iPad 2, a point multiplication takes $14100\mu s$.[1]

Our implementation can also be compared to the one provided in the JSBN library [22], which is based on standard NIST curves, uses NAF point multiplication, and does not use precomputation. The point multiplication that took $550\mu s$ on the recent laptop mentioned above takes then around $30000\mu s$ with the JSBN implementation, presenting a slowdown of a factor 54 for the same security level. This gain comes from the various changes we made compared to the JSBN implementation: optimized modulus choice, specific modular reduction algorithm, choice of point representation, and precomputation.

[1] We thank Benoît Dumoulin for taking this measurement.

4 Application to e-voting

One possible use context for our ECC library is the Helios open-audit voting system [1, 2, 8], which has been used with two different cryptographic protocols on the client side:

1. The commonly deployed version, proposed in [2], is based on homomorphic tallying and uses a variant of the CGS protocol of Cramer et al. [9].
2. For some elections, mixnet-based tallying has also been used [8].

The homomorphic tallying approach enables a very simple election workflow, where the work of the election trustees is minimal: they only need to decrypt the election outcome, which is even cheaper than preparing a ballot. However, the ballot preparation procedure is fairly expensive for the voter, as it requires the equivalent of 6 point multiplications per candidate.

This computational complexity was the actual motivation for the adoption of a mixnet-based approach, when an election involving around 250 candidates was organized: adopting mixnets reduced the amount of computation to the equivalent of 5 point multiplications per ballot, but implied a substantially more complicated tallying procedure, including the setup of mix servers and requiring the trustees to decrypt all mixed ballots individually.

The fixed point multiplication techniques we explored in the previous section are particularly suitable for the CGS protocol. Indeed, all point multiplications are performed with respect to only two bases: a public group generator and an ElGamal public key which is made of a single point.

Using the Chrome browser on the average netbook described with our previous measurements, the time required to perform 1500 point multiplications when preparing a ballot for 250 candidates would be around 3 seconds, which is quite usable. It is not even necessary to require the voter to wait during those 3 seconds, as all point multiplications can be made independent of the voter choices, which can be encoded through point additions performed at the end of the ballot preparation procedure. The point multiplication operations can then be performed in separated worker threads while the voter performs his choices.

So, the library we presented in this paper provides an answer to the efficiency concern in Helios for elections involving a large number of candidates, and is expected to substantially increase the proportion of elections that can benefit from the simplicity of homomorphic tallying procedures.

5 Conclusion

Starting from the work of Tom Wu in the JSBN library for the support of big interger operations in JavaScript, we explored various strategies for the implementation of elliptic curve cryptography in pure JavaScript. Our resulting implementation, relying on a limited amount of precomputed data, offers a speedup of a factor 50 compared to the one proposed in the JSBN library. The efficiency of our implementation opens the way of substantial improvements in various

JavaScript applications, and we discussed the Helios voting system as an example.

There are a number of directions that remain open for further research.

- We concentrated our effort on NIST-type elliptic curves. It would be very interesting to explore whether other curve families would provide better results.
- Our library assumes that the precomputed data for fixed point multiplication are provided by an external application server. Including the cost of precomputation in the choice of the point multiplication technique would be another very interesting direction.

The adoption of our cryptographic library for real world applications remains currently limited by the lack of availability of secure randomness in JavaScript. Some efforts were already realized [4, 21], based on variants of the Fortuna design for entropy accumulation. More recently, since version 11, the Chrome browser exposes secure randomness through a new window.crypto.getRandomValues API, which provides a much more convenient and reliable solution. We hope to see secure randomness become available in other browsers within a near future.

Acknowledgments

We would like to thank Nicolas Veyrat-Charvillon for his support and the anonymous LC 2011 referees for their useful comments.

References

1. Ben Adida. Helios: web-based open-audit voting. In *Proceedings of the 17th USENIX Security Symposium*, pages 335–348, Berkeley, CA, USA, 2008. USENIX Association.
2. Ben Adida, Olivier de Marneffe, Olivier Pereira, and Jean-Jacques Quisquater. Electing a University President Using Open-Audit Voting: Analysis of Real-World Use of Helios. In T. Moran D. Jefferson, J.L. Hall, editor, *Electronic Voting Technology Workshop/Workshop on Trustworthy Elections*. Usenix, August 2009.
3. Daniel V. Bailey and Christof Paar. Optimal extension fields for fast arithmetic in public-key algorithms. In Hugo Krawczyk, editor, *Advances in Cryptology - CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 472–485. Springer, 1998.
4. Marco Barulli and Giulio Cesare Solaroli. Clipperz. `http://www.clipperz.org`. Accessed on Oct 10, 2011.
5. Daniel J. Bernstein and Tanja Lange. Faster addition and doubling on elliptic curves. In *Advances in Cryptology - ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 29–50. Springer, 2007.
6. Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In *Financial Cryptography and Data Security*, pages 325–343, Berlin, Heidelberg, 2009. Springer-Verlag.

7. Ernest F. Brickell, Daniel M. Gordon, Kevin S. McCurley, and David Bruce Wilson. Fast exponentiation with precomputation (extended abstract). In *Advances in Cryptology - EUROCRYPT '92*, volume 658 of *Lecture Notes in Computer Science*, pages 200–207. Springer, 1992.
8. Philippe Bulens, Damien Giry, and Olivier Pereira. Running mixnet-based elections with Helios. In H. Shacham and V. Teague, editors, *Electronic Voting Technology Workshop/Workshop on Trustworthy Elections*. Usenix, 2011.
9. Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient multi-authority election scheme. In Walter Fumy, editor, *Advances in Cryptology - EUROCRYPT '97*, volume 1233 of *Lecture Notes in Computer Science*, pages 103–118. Springer, 1997.
10. Zhi Guan, Zhen Cao, Xuan Zhao, Ruichuan Chen, Zhong Chen, and Xianghao Nan. WebIBC: Identity based cryptography for client side security in web applications. In *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, pages 689–696. IEEE Computer Society, 2008.
11. Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to elliptic curve cryptography*. Springer-Verlag, 2004.
12. Laurie Haustenne and Quentin de Neyer. Elliptic curve cryptography in javascript with application for eVoting. Master's thesis, Universite catholique de Louvain, 2011.
13. Michael Hutter and Erich Wenger. Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 459–474. Springer, 2011.
14. ECMA International. ECMAScript Language Specification – ECMA-262 rev. 5.1, 2011.
15. A. Karatsuba and Yu. Ofman. Multiplication of many-digital numbers by automatic computers. In *Proceedings of the USSR Academy of Sciences*, volume 145, page 293–294, 1962.
16. Chae Hoon Lim and Pil Joong Lee. More flexible exponentiation with precomputation. In *Advances in Cryptology - CRYPTO '94*, volume 839 of *Lecture Notes in Computer Science*, pages 95–107. Springer, 1994.
17. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, July 1999.
18. MozillaWiki. Weave cryptography developer overview. `https://wiki.mozilla.org/Labs/Weave/Developer/Crypto`, February 2010. Accessed on Oct 10, 2011.
19. NIST. FIPS PUB 186-3 – Digital Signature Standard (DSS). `http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf`, 2009.
20. Dave Shapiro. RSA in JavaScript. `http://ohdave.com/rsa/`. Accessed on Oct 10, 2011.
21. Emily Stark, Michael Hamburg, and Dan Boneh. Symmetric cryptography in Javascript. In *Twenty-Fifth Annual Computer Security Applications Conference, ACSAC 2009*, pages 373–381. IEEE Computer Society, 2009.
22. Tom Wu. jsbn - BigIntegers and RSA in JavaScript. `http://www-cs-students.stanford.edu/~tjw/jsbn/`. Accessed on Oct 10, 2011.