From New Technologies to New Solutions
Exploiting FRAM Memories to Enhance Physical Security

Stéphanie Kerckhof[1], François-Xavier Standaert[1], Eric Peeters[2]

[1] ICTEAM/ELEN/Crypto Group, Université catholique de Louvain, Belgium.
[2] Texas Instruments, Dallas, Texas, USA.

`e-mails:` stephanie.kerckhof@uclouvain.be; fstandae@uclouvain.be; e-peeters@ti.com

Abstract. Ferroelectric RAM (FRAM) is a promising non-volatile memory technology that is now available in low-end microcontrollers. Its main advantages over Flash memories are faster write performances and much larger tolerated number of write/erase cycles. These properties are profitable for the efficient implementation of side-channel countermeasures exploiting pre-computations. In this paper, we illustrate the interest of FRAM-based microcontrollers for physically secure cryptographic hardware with two case studies. First we consider a recent shuffling scheme for the AES algorithm, exploiting randomized program memories. We exhibit significant performance gains over previous results in an Atmel microcontroller, thanks to the fine-grained programmability of FRAM. Next and most importantly, we propose the first working implementation of the "masking with randomized look-up table" countermeasure, applied to reduced versions of the block cipher LED. This implementation provides unconditional security against side-channel attacks (*of all orders!*) under the assumption that pre-computations can be performed without leakage. It also provides high security levels in cases where this assumption is relaxed (e.g. for context or performance reasons).

1 Introduction

Providing (physical) security against side-channel attacks is a challenging task for cryptographic designers [10]. This is especially true for low-cost embedded devices, with strongly constrained resources. Typical examples of countermeasures in this context include masking [3] and shuffling [9]. But in both cases, the concrete security levels attained by the protected implementations highly depend on hardware assumptions, in particular the amount of noise in the measurements which may not be sufficient, e.g. in 8-bit devices [19, 20]. The performance overhead they imply can also be significant [7, 15]. As a result, such countermeasures are usually combined in a somewhat heuristic manner, in order to ensure "practical" security against a wide enough category of adversaries [16].

In parallel to these advances, some more recent works have tried to formalize the problem of physical security, in order to extend the guarantees of provable security from algorithms and protocols to implementations. The main challenge in this case is to find relevant restrictions of the adversaries. A typical

example is the one of leakage-resilient cryptography, where the assumption is that the information leakage of a single algorithm run is bounded (see, e.g. [5] for an early reference, [17] for a recent one, and many other proposals in between). Alternatively, another line of work is based on the assumption of secure pre-computations, e.g. in order to prevent "continual memory attacks", as formalized by Brakerski et al. [2] and by Dodis et al. [4]. The one-time programs introduced at Crypto 2008 are another (extreme) way to exploit such secure pre-computations [6]: they essentially correspond to a program that can be executed on a single input, whose value can be specified at run time. Nevertheless, the practical relevance of these solutions is still limited by sometimes unrealistic hardware assumptions and (mainly), by large performance overheads.

In this paper, we start from the observation that both for practice-oriented and theory-oriented countermeasures against side-channel analysis, the exploitation of secure pre-computations is highly related to the problem of fast and efficient non-volatile storage. In this context, a significant drawback of the mainstream Flash memories is that write operations are slow and energy-consuming. Furthermore, their number of tolerated write/erase cycles is also limited (from 10k to 100k, typically), which may prevent their frequent use for cryptographic operations. Interestingly, the recently available Ferroelectric RAM (FRAM) provides a solution to these issues[1]. As a result, we investigate whether it can be used as a technology enabler to improve the performances and security of protected implementations. For this purpose, we first consider the shuffling countermeasure, and its instantiation for the AES algorithm based on randomized program memories proposed in [20]. We show that FRAM allows significantly improved performances in terms of pre-computation time. Next, we discuss the application of these new memories to the Randomized Look-Up Table (RLUT) countermeasure [18]. It can be viewed as a type of one-time program specialized to side-channel analysis, or as a generic masking scheme that provides unconditional security against side-channel attacks of all orders (i.e. independent of the statistical moment estimated by the adversary). We provide the first working implementation of this solution applied to reduced versions of the block cipher LED [8], and analyze its performances in various settings. In particular, we investigate the contexts of complete and secure pre-computations, and the tradeoffs corresponding to partial (and partially leaking) ones. We reach performances that are close to higher-order masking in the latter case [15], while complete and secure pre-computations also ensures much higher security levels at practically reachable cost. Therefore, our results suggest that FRAM is a promising solution for improving the security of low-cost tokens such as smart cards, especially when some pre-computations can be performed in a safe environment.

---

[1] Strictly speaking, FRAM is not a new technology as it was introduced as a high-security alternative to Flash memories back in the early 2000s by Fujitsu. However, FRAM-based smart cards did not make it to mass market at that time, due to excessive manufacturing costs and limited ability to reduce cell transistor size.

The rest of the paper is structured as follows. Section 2 provides the necessary background on FRAM and discusses our security model. Section 3 contains the implementation results of the shuffling with randomized program memory countermeasure, and their comparison with the previous work from Asiacrypt 2012. Section 4 describes the RLUT countermeasures, our proposed implementation and the various tradeoffs it provides. Finally, conclusions are in Section 5.

2 Background

2.1 FRAM microcontrollers

Standard solutions for non-volatile storage such as Flash and EEPROM usually suffer from long programming times, as well as a high voltage required to program bit cells with hot carrier injection or Fowler-Nordheim tunneling effect. In addition, the charge pump overhead as well as the high current supply they require make these technologies not ideal for applications where frequent data logging or ultra-low-power write operations are needed (e.g. all RF applications such as e-passport, RF Banking Card, ...). FRAM is a promising alternative that combines the advantages of non-volatile memories with much faster write speed (e.g. 125 ns per 64-bit word for the 130-nm TI FRAM technology exploited in MSP430FR devices), less power (82 uA/MHz active power in the same technology) and infinite (10^{15}) write-erase cycle performances.

FRAM stores information through the use of a stable electric dipole found in ferroelectric crystals (insensitive to the magnetic field). The polarization-voltage hysteresis loops for such materials are very similar to the B-H curve of magnetic materials. Exploiting this fact, a FRAM bit cell structure consists of a ferroelectric capacitor containing the crystal. The capacitor is connected to a plate line, bit lines, and a transistor switch to access it. This is also referred to as a 1T-1C memory cell mode (Figure 1, left). By contrast, 2T-2C memory cell modes (Figure 1, right) would store the data as 2 opposite values in each 1T-1C cell of its structure (similar to what is found in EEPROM for instance). Reading
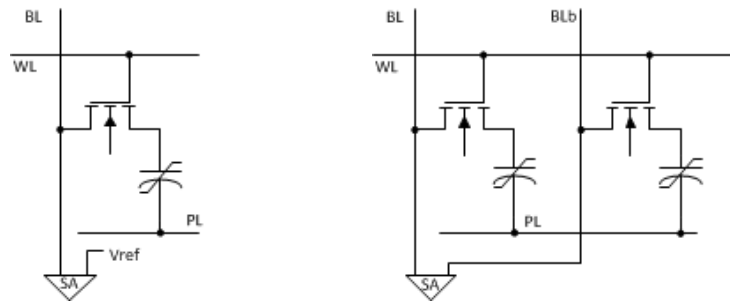

Fig. 1. FRAM bit cell modes: left: 1T-1C structure, right: 2T-2C structure.

the data from FRAM occurs by placing a voltage on the plate line. The idea is that for every read operation, one tries to set the cell to a 0 state. If the voltage causes the dipoles inside the capacitor to flip its orientation, then a large charge Q is generated on the bit line. On the contrary, if the orientation of the dipole is already negative prior to applying the voltage to the plate line in a read cycle, then the dipole direction does not flip, and only a small charge Q is induced on the bit line. The difference can be measured by a sense amplifier. An important consequence of this description is that FRAM reads are destructive, and therefore require a refresh process. Nevertheless, this process is automatically completed by the controller and therefore transparent to the user.

In this paper, we used a microcontroller of the MSP430FRxxxx family from Texas Instruments. This type of microcontrollers provides an ultra-low-power 16-bit RISC CPU, and a set of instructions performing operations on either 8 or 16 bits of data. The available non-volatile FRAM memory can have a size of up to 64 kilobytes (microcontrollers with 128-and 256-kilobyte capabilities are already announced). All the developed code was made for a MSP430FR5739 microcontroller, containing 16 kilobytes of FRAM, and was tested using the MSP-EXP430FR5739 experimenter's board and Code Composer Studio 5.3.

2.2   Security model

The following sections mainly aim at demonstrating the efficiency of FRAM-based cryptographic computations. Yet, since we consider implementations protected against side-channel attacks, it is important to say a few words about the security model we rely on. Both for the shuffling in Section 3 and for the RLUT countermeasure in Section 4, we can consider two alternatives:

1. *Secure pre-computations.* That is, the permutations used in shuffling and the randomized program used in RLUT are pre-computed without leakage, prior to the execution of the cryptographic algorithm. As a result, the security of the shuffling is exactly the one analyzed at Asiacrypt 2012 [20]. And the security of the RLUT countermeasure is unconditional: even an adversary accessing the (identity) leakage of all the intermediate computations in the target implementation would not recover any information about its key.
2. *Leaking pre-computations.* That is, the permutations used in shuffling and the randomized program used in RLUT are computed online, and leaking information. In this case, the security of both countermeasures is less investigated, and essentially depends on how much information is leaked during pre-computation (in fact, the same observation holds for most countermeasures exploiting randomness, e.g. masking). Note however that the randomness used to protect these implementations is generated on-chip, and is never output. Hence, adversaries can only mount SPA attacks against it. As a result, we can informally state that our implementations will remain secure in this context, as long as one can guarantee SPA security for this part of the computation (a similar informal separation between SPA and DPA was used to argue about the security of fresh re-keying schemes, e.g. in [12]).

Note that in terms of security, the main advantage of FRAM is to make the first model more realistic. Indeed, one could (at least theoretically) imagine to implement a randomized program with SRAM memories. But in addition to performances that would most likely be poor in this case, such a solution should anyway be implemented online (hence leaking), since SRAM is volatile.

3 Improving past results: the shuffling case

Shuffling the execution order of independent operations is a possible solution to improve security of cryptographic implementations against side-channel attacks. The goal of shuffling is to distribute the intermediate cipher values over a given period of time, so that an attacker will only be able to observe a chosen intermediate value at a particular moment in time with a certain probability. A typical example of independent operations that can be shuffled is the `SubBytes` layer in the AES. Indeed, whatever the order in which each of the 16 S-Box's outputs is generated, the result of the `SubBytes` layer will not be affected.

A previous work on shuffling, proposed 3 implementations of the AES on an Atmel ATMega644p microcontroller [20]: a basic one with double indexing, an optimized one with randomized execution path, and a variant with randomized program memory. In this paper, we focus on this third proposal, for which FRAM technology provides significant improvements (the two first ones lead to essentially similar performances, independent of the non-volatile memory used). Randomizing the program memory corresponds to rewriting the code in a randomized way before each algorithm run. In other words, a pre-computation phase modifies (inside the code) which registers and memory addresses will be used during the execution, which then remains essentially the same as an unshuffled one. While promising in principle, such an instantiation of the shuffling idea faces some limitations when implemented in Flash-based Atmel microcontrollers. First, even when only a few bytes need to be modified, a complete memory page must be erased and rewritten, which takes a lot of time (more or less 4.5 milliseconds each time a page is written or erased). Next, the memory can only be rewritten a limited number of times (namely 10 000). Hence, FRAM-based microcontrollers are natural candidates to relax these limitations as we now detail.

Implementing the AES algorithm with randomized program memory requires three main functions. First, a permutation generator must be defined - we used exactly the same implementation as proposed in [20]. Next, it is necessary to have an AES description with well defined sets of 16 independent operations, on which the shuffling can be applied. Although such operations are easily found for `SubBytes` and `AddRoundKey`, their specification is more difficult for `ShiftRows` and `MixColumns`, for which the 16 bytes are not manipulated independently. This implies that their output cannot be stored at the same location as their input, resulting in the need of 16 additional bytes of temporary storage. Furthermore, the FRAM microcontroller we use has only 12 CPU registers, which is not enough to store a complete AES state. Therefore, each independent operation needs to access the FRAM with absolute addressing, which is more time

consuming than working on registers. As for the implementations of Asiacrypt 2012, dummy key-schedule operations have also been added to the "on-the-fly" key-schedule, in order to obtain enough independent operations for this part of the implementation as well [20]. Eventually, the last function needed is the one randomizing the code before execution. This randomization was achieved by modifying the bytes of instructions referring to the cipher state's or round key's memory addresses. Interestingly, since the code and the data are both stored in the same FRAM memory, modifying some bytes of the code or some bytes of cipher state and round key takes exactly the same amount of time.

Our implementation results are available in Table 1 (and are given for encryption only). For reference, we first implemented an unshuffled version of the AES in the MSP430FR5739 microcontroller. Even if performance comparisons obtained with different technologies always have to be considered with care, it is worth noticing that it is slightly more time-consuming than Atmel ones (e.g. the open source AES Furious requires 3546 cycles to execute [13]). This is mainly a consequence of the limited number of registers available in FRAM microcontrollers, leading to more frequent memory accesses. By contrast and as expected, the pre-computation time required to shuffle the code is strongly reduced, from 18 milliseconds in Atmel devices to 0.19 milliseconds (running the chip at 16 MHz), which corresponds to a ratio of approximately 100. This is the main advantage of our implementation. Note finally the increased data size and cycle count for executing the AES in its shuffled version, which is essentially due to the previously mentioned execution of dummy key-schedules. We conclude that the overhead required to shuffle the AES algorithm based on a randomized program memory is now in line with practical applications constraints.

		Code Size	Data Size	Cycle Count
Unprotected AES		1076	52	5800
Shuffled AES	Perm. Generation	194	18	2240
	Code Shuffling	418	0	2751
	AES execution	2404	146	8479
	Total	3016	164	13470

Table 1. AES program size (in bytes) and cycle counts in the MPS430FR5739.

4 Making new results possible: the RLUT case

The previous section described how FRAM memories allow significant speedups for shuffled implementations exploiting randomized program memories. In this section, we show how similar ideas can be used to enable the efficient implementation of the RLUT countermeasure. For this purpose, we first recall the intuition behind this countermeasure, then describe its application to reduced versions of the block cipher LED, and finally discuss implementation results.

4.1 Description of the countermeasure

We will focus on the protection of a single S-box that is the most challenging part of the countermeasure. Intuitively, it is convenient to start from the first-order Boolean masking depicted in Figure 2. In this scheme, a random mask m is first added to the sensitive value x which is then sent trough the combination of a bitwise key addition $\oplus$ and S-box S. A correction function C is used (taking both $x \oplus m \oplus k$ and m as input) in order to produce the output mask q such that $\mathsf{S}(x \oplus m \oplus k) = \mathsf{S}(x \oplus k) \oplus q$. Such an implementation typically gives rise to 4 leakage points denoted as L_1, L_2, L_3 and L_4 on the figure (L_2 being the combination of two parts). It ideally guarantees that statistical moments of order 2 will have to be estimated by an adversary in order to recover secret information. The word "ideally" here refers to the fact that physical defaults such as glitches can lead to exploitable information in lower-order statistical moments [11]. For example, the leakage point L_2 on the figure corresponds to the manipulation of $x \oplus m \oplus k$ leading to $\mathsf{L}_2^a(x \oplus m \oplus k)$, and m leading to $\mathsf{L}_2^b(m)$, in parallel. It implies first-order exploitable information if these two parts of the leakage function are not independent[2]. Boolean masking can be naturally generalized to d shares ($d = 2$ in the example of Figure 2), leading to an (ideal as well) data complexity increase proportional to $(\sigma_n^2)^d$, with σ_n^2 the variance of the noise in the leakage samples, as demonstrated by Chari et al. [3].

From this description, a first step towards the RLUT countermeasure is the observation that if the master key is fixed and for n-bit S-boxes, one can replace the computation of $\mathsf{S}(x \oplus k)$ by a pre-computed table of size $2^n \times n$ (the correction function can be implemented similarly as a table of size $2^{2n} \times n$). It directly leads to the implementation of Figure 3, which is functionally equivalent to the previous one, but where the key addition has been "included" in a key-dependent permutations $\mathsf{P}_k(x)$. From the side-channel security point-of-view, it still corresponds to a first-order secure implementation. Next, and in order to provide unconditional security against side-channel attacks of all orders, the main idea is to replace the Boolean masking operation $x \oplus m$ by an extension to three shares denoted as $\mathsf{G}_i(x, m) = x \oplus m \oplus a_i$, where a_i is a n-bit random mask that is pre-computed in a leakage-free environment. These operations, illustrated in Figure 4, can also be implemented as tables of size $2^{2n} \times n$, so that the shares a_i will never be manipulated during the "online" execution of the algorithm. If the G_1 (resp. G_2) function is refreshed before each run of the protected implementation, it guarantees that no information can be extracted from the leakage points $(\mathsf{L}_1, \mathsf{L}_2)$ (resp. $(\mathsf{L}_3, \mathsf{L}_4)$). We additionally need G_1 and G_2 (i.e. their hidden a_i shares) to be independent, in order to avoid fourth-order leakages taking advantage of the correlation between the tables' inputs and outputs. Eventually, it remains to randomize the permutation $\mathsf{P}_k(x)$ (and the correction function C) in order to completely hide the key, even from identity leakage functions, as represented in Figure 5. This way, a non-linear S-box can be computed securely.

---

[2] As a typical example, $\mathsf{L}_2 = \mathsf{L}_2^a + \mathsf{L}_2^b$ would correspond to an ideal implementation, while $\mathsf{L}_2 = \mathsf{L}_2^a \cdot \mathsf{L}_2^b$ would leak first-order information, as discussed in [19].
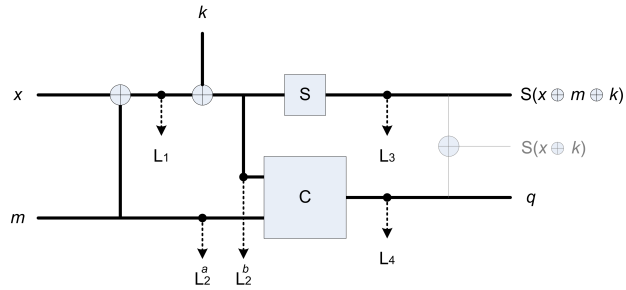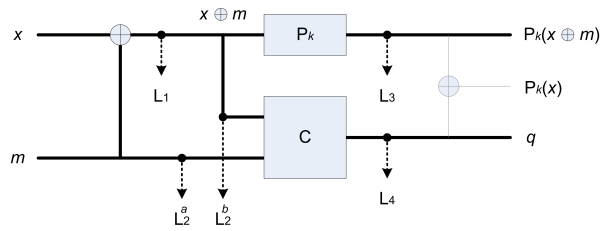
Fig. 2. Boolean masking.
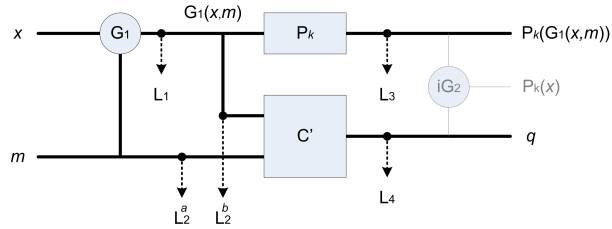

Fig. 3. Boolean masking with LUTs.

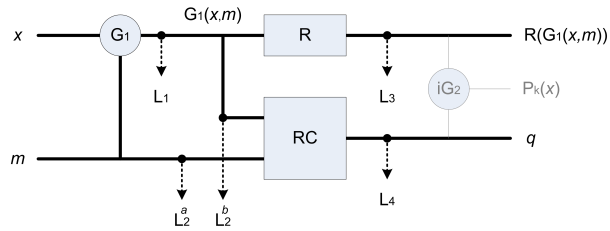**Fig. 4.** Randomized Boolean masking with LUTs.

Fig. 5. Randomized Boolean masking with randomized LUTs.

This leads to an implementation in which the operations G_1, G_2, R and RC are pre-computed according to Algorithm 1, and executed according to Algorithm 2. Extending this S-box computation to a complete cipher is straightforward: we just need independent tables for all the S-boxes. As for the linear operations, they have to be applied independently on the two shares that are explicitly manipulated by the leaking device, just as in standard Boolean masking.

Algorithm 1 - Table refreshing.

- **input:** P_k.
1. Pick $a_1 \xleftarrow{R} \{0,1\}^n$;
2. Pick $a_2 \xleftarrow{R} \{0,1\}^n$
3. Pick $a_3 \xleftarrow{R} \{0,1\}^n$;
4. Pre-compute $G_1(I,J) = I \oplus J \oplus a_1$;
5. Pre-compute $R(I) = P_k(I) \oplus a_2$;
6. Pre-compute $G_2(I,J) = I \oplus J \oplus a_3$;
7. Pre-compute $RC(I,J) = r(I) \oplus p_k(I \oplus J \oplus a_1) \oplus a_3$;
- **output:** G_1, R, G_2, RC.

Algorithm 2 - S-box evaluation on input x.

- **input:** G_1, R, RC.
1. Pick $m \xleftarrow{R} \{0,1\}^n$;
2. Compute $G_1(x,m)$;
3. Compute $R(G_1(x,m))$;
4. Compute $RC(G_1(x,m),m)$;
- **output:** $R(G_1(x,m))$, $RC(G_1(x,m),m)$.

4.2 Application to reduced LED

The previous section suggests that the RLUT countermeasure has high memory requirements, that strongly depend on the S-box size used in the block cipher to protect. In particular, given a N_r-round cipher with N_s S-boxes per round, the implementation of the RLUT countermeasure essentially requires storing:

- A *table map* that corresponds to all the a_i shares generated during pre-computation, with memory cost estimated as $(N_s \cdot N_r) \cdot 2 + N_s$ n-bit words (where the factor 2 corresponds to the fact that excepted for the first round, the share a_1 in Algorithm 1 is always provided by the previous round).
- A *randomized program* that corresponds to the tables R and RC, with memory cost estimated as $N_r \cdot N_s$ tables of size $2^n \times n$ and $2^{2n} \times n$, respectively.

Note that operations G_i are never explicitly used during the cipher execution, but for the first round to mask, and last round to unmask after a secure computation is completed. Following these estimations, and as discussed in [18], it is natural to consider a cipher with 4-bit S-boxes for this purpose. In the following, we will consider reduced (16-bit) versions of the LED cipher illustrated in Figure 6.
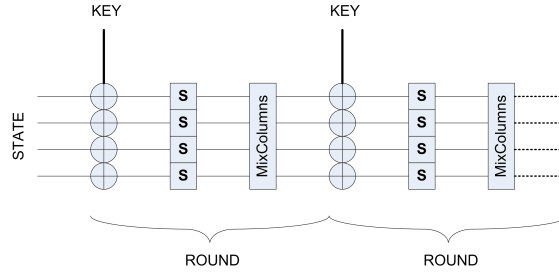
Fig. 6. Reduced version of the block cipher LED.

While such a cipher is naturally too small for being deployed in actual applications, we use it to refine our model for RLUT performance estimates. As will be discussed in the next sections, scaling to larger number of rounds and block size (e.g. the full 64-bit LED cipher) will be possible in soon available 128- and 256-kilobyte versions of our FRAM microcontroller. In the figure, the 4-bit S-boxes of LED are denoted as S, and its linear diffusion layer as MixColumns.

4.3 Implementation in FRAM microcontrollers

We now describe how to implement reduced (with up to 4 rounds) LED ciphers within the 16 kilobytes of FRAM available in our MSP430FR microcontroller.

The first building block required in a RLUT-masked implementation is a randomness generator (needed to produce the a_i values of Algorithm 1). For illustration, we used a LFSR with CRC-32 polynomial for this purpose (alternative ways of generating randomness could of be considered, e.g. using a leakage-resilient PRG if leaking pre-computations are considered [5, 17]).

Next, the part computing the randomized program can be implemented quite straightforwardly, following the description in Section 4.1. The trickiest bit was to efficiently arrange 4-bit outputs into memory bytes, without giving any unnecessary information on the RLUT input values[3]. Using one byte to store two consecutive RLUT outputs was rejected, since accessing one or the other value in the byte would have led to different code behaviors, depending on the LSB bit of the RLUT's input. Instead, we stored the outputs coming from two different RLUTs for the same input value in a single byte. This time, the LSB (resp. MSB) part of one byte will be accessed when an odd (resp. even) word of the state needs to be computed, giving no information on the word's value itself. Based on this strategy, the RLUTs R and RC can be generated efficiently from the cipher key, the S-Box and the table map, that are all stored in memory.

---

[3] This has no impact on the security in case of secure pre-computation, but may increase the information leakage in case of online randomization of the tables.

Eventually, the last piece of code concerns the execution of the block cipher itself. Again, the fact that the operations are performed on 4-bit words had to be taken into account while accessing the variables or tables stored in memory. One round of the reduced algorithm is executed by first reading the R and RC tables' outputs, corresponding to the cipher state and mask intermediate values. Then, the `MixColumn` layer is executed on each of the shares. It is implemented using an `Xtime` table, as suggested in the specifications of LED [8].

4.4 Results and discussion

As described in Section 4.2, an estimation of the memory size required to store the table map and randomized program of the RLUT countermeasure can be derived from the number of rounds N_r, the number of S-Boxes per round N_s and the S-Box bit-size n. This estimation is illustrated by the dashed line of figure 7, in the case where $N_s = 4$, $n = 4$ and N_r varies from 1 to 4. A plain line representing the actual results we obtained for our implementation is also plotted. The two curves follow the same trend, with the offset separating them corresponding to the code size needed to implement the cipher itself (i.e. excluding the tables for which the memory requirements are growing with N_r - see the detailed results in Appendix A, Table 2). Interestingly, these results suggest that for any parameters N_r, N_s and n, the memory requirements needed to implement a block cipher protected with the RLUT countermeasure can be quite accurately predicted. For example, such an implementation for a full (64-bit) version of the block cipher LED (corresponding to $N_r = 32$, $N_s = 16$ and $n = 4$) would roughly require a memory size of 70 kilobytes, and could therefore be implemented in the soon available 128-kilobyte FRAM microcontrollers.
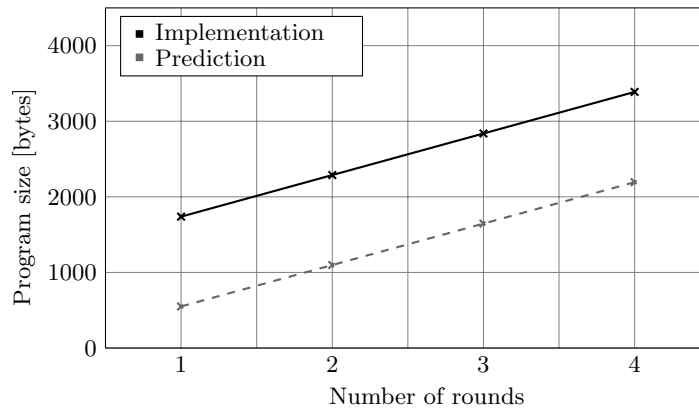

Fig. 7. Program size of the LED cipher protected with RLUTs.

The question of accurate predictions can also be asked for pre-computation time: estimates for this metric were similarly provided in [18]. Namely, the time needed to generate the RLUTs can be approximated with $((N_s.N_r).2 + N_s) + (N_s.N_r).2^n + (N_s.N_r).2^{2n}$ "elementary operations" (where the first term corresponds to randomness generation, and the later ones correspond to the refreshing of the R and RC tables). Our actual implementation results directly allow translating these "elementary operations" into a concrete number of clock cycles. The results in Figure 8 (also reported in Appendix A, Table 3) again confirm a nice correlation with predictions. Namely, the main difference between both curves is a factor 40, which presumably corresponds to the number of cycles needed to perform each elementary operations. Extrapolating these results to the full (64-bit) version of the LED block cipher suggests pre-computation time complexities around 140 000 elementary operations, corresponding to 5 600 000 cycles (i.e. an execution time of 35 milliseconds at 16MHz), which would be acceptable for some applications (and is likely to be improved with technology scaling).
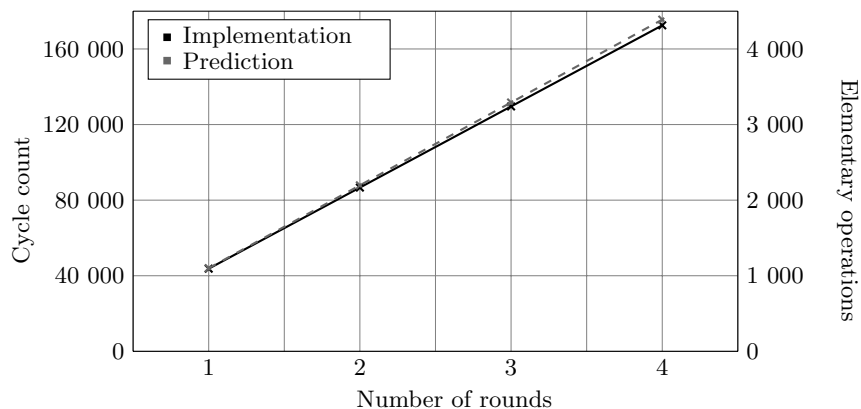

Fig. 8. Pre-computation time of the LED cipher protected with RLUTs.

Eventually, it is worth noticing that time and memory complexities could be reduced by exploiting some performance vs. security tradeoffs. A first solution would be an implementation for which only some rounds are masked with RLUTs, while the others use standard masking schemes. For example, one could protect only the first and last 4 rounds of (full, 64-bit) LED, i.e. 8 out of 32, resulting in an approximate reduction of the time and memory complexities down to 25% of their original values. However, this solution may be risky in front of advanced attacks such as algebraic ones, that can exploit the leakage of the middle rounds [14]. Another approach to reduce the pre-computation time consists in refreshing only a fraction of the table map (and randomized program) after each cipher execution, and to perform this refreshing randomly. Interestingly, such a tradeoff would also take advantage of the non-volatile capacities of FRAM mem-

ories, since the complete randomized program could be pre-computed offline, while only a part of it would be modified online. It is flexible since the fraction of RLUTs modified per cipher executions could be adapted to the application requirements. As an illustration, modifying 10% of the RLUTs in (full, 64-bit) LED would reduce the pre-computation time to approximately 560 000 cycles, which is getting close to the performances of a third-order masked AES (470 000 cycles in [15]). Combining the two appraoches would of course be possible as well, e.g. by randomizing the first- and last-round tables in priority.

5 Conclusion

Our results put forward that FRAM is a promising technology in the context of side-channel resistant cryptographic hardware, since it enables the efficient implementation of various countermeasures taking advantage of pre-computations. The case of RLUTs is particularly relevant to illustrate this observation. Indeed, they have never been implemented so far, they will soon be applicable to complete block ciphers, and may lead to high security levels for small embedded devices, independent of hardware assumptions that may be hard to fulfill. Important scopes for further investigations include the evaluation of the security levels obtained in the context of partially leaking pre-computations. In particular, analyzing the online refreshing of partially randomized programs mentioned in Section 4.4 would be very useful. Besides, the design of block ciphers that are well suited to implementations with RLUTs (e.g. with light(er) non-linear layers and strong(er) linear ones) is another interesting research avenue.

References

1. *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*. IEEE Computer Society, 2010.
2. Zvika Brakerski, Yael Tauman Kalai, Jonathan Katz, and Vinod Vaikuntanathan. Overcoming the Hole in the Bucket: Public-Key Cryptography Resilient to Continual Memory Leakage. In *FOCS* [1], pages 501–510.
3. Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In Michael J. Wiener, editor, *CRYPTO*, volume 1666 of *LNCS*, pages 398–412. Springer, 1999.
4. Yevgeniy Dodis, Kristiyan Haralambiev, Adriana López-Alt, and Daniel Wichs. Cryptography against Continuous Memory Attacks. In *FOCS* [1], pages 511–520.
5. Stefan Dziembowski and Krzysztof Pietrzak. Leakage-Resilient Cryptography. In *FOCS*, pages 293–302. IEEE Computer Society, 2008.

6. Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. One-Time Programs. In David Wagner, editor, *CRYPTO*, volume 5157 of *Lecture Notes in Computer Science*, pages 39–56. Springer, 2008.

7. Vincent Grosso, François-Xavier Standaert, and Sebastian Faust. Masking vs. Multiparty Computation: How Large Is the Gap for AES? In Guido Bertoni and Jean-Sébastien Coron, editors, *CHES*, volume 8086 of *Lecture Notes in Computer Science*, pages 400–416. Springer, 2013.

8. Jian Guo, Thomas Peyrin, Axel Poschmann, and Matthew J. B. Robshaw. The LED Block Cipher. In Bart Preneel and Tsuyoshi Takagi, editors, *CHES*, volume 6917 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2011.

9. Christoph Herbst, Elisabeth Oswald, and Stefan Mangard. An AES Smart Card Implementation Resistant to Power Analysis Attacks. In Jianying Zhou, Moti Yung, and Feng Bao, editors, *ACNS*, volume 3989 of *Lecture Notes in Computer Science*, pages 239–252, 2006.

10. Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks - Revealing the Secrets of Smart Cards.* Springer, 2007.

11. Stefan Mangard, Thomas Popp, and Berndt M. Gammel. Side-Channel Leakage of Masked CMOS Gates. In Alfred Menezes, editor, *CT-RSA*, volume 3376 of *Lecture Notes in Computer Science*, pages 351–365. Springer, 2005.

12. Marcel Medwed, François-Xavier Standaert, Johann Großschädl, and Francesco Regazzoni. Fresh Re-keying: Security against Side-Channel and Fault Attacks for Low-Cost Devices. In Daniel J. Bernstein and Tanja Lange, editors, *AFRICACRYPT*, volume 6055 of *Lecture Notes in Computer Science*, pages 279–296. Springer, 2010.

13. Bertram Poettering. Rijndael Furious, http://point-at-infinity.org/avraes/.

14. Mathieu Renauld and François-Xavier Standaert. Algebraic Side-Channel Attacks. In Feng Bao, Moti Yung, Dongdai Lin, and Jiwu Jing, editors, *Inscrypt*, volume 6151 of *Lecture Notes in Computer Science*, pages 393–410. Springer, 2009.

15. Matthieu Rivain and Emmanuel Prouff. Provably Secure Higher-Order Masking of AES. In Stefan Mangard and François-Xavier Standaert, editors, *CHES*, volume 6225 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2010.

16. Matthieu Rivain, Emmanuel Prouff, and Julien Doget. Higher-Order Masking and Shuffling for Software Implementations of Block Ciphers. In Christophe Clavier and Kris Gaj, editors, *CHES*, volume 5747 of *Lecture Notes in Computer Science*, pages 171–188. Springer, 2009.

17. François-Xavier Standaert, Olivier Pereira, and Yu Yu. Leakage-Resilient Symmetric Cryptography under Empirically Verifiable Assumptions. In Ran Canetti and Juan A. Garay, editors, *CRYPTO (1)*, volume 8042 of *Lecture Notes in Computer Science*, pages 335–352. Springer, 2013.

18. François-Xavier Standaert, Christophe Petit, and Nicolas Veyrat-Charvillon. Masking with Randomized Look Up Tables - Towards Preventing Side-Channel Attacks of All Orders. In David Naccache, editor, *Cryptography and Security*, volume 6805 of *Lecture Notes in Computer Science*, pages 283–299. Springer, 2012.

19. François-Xavier Standaert, Nicolas Veyrat-Charvillon, Elisabeth Oswald, Benedikt Gierlichs, Marcel Medwed, Markus Kasper, and Stefan Mangard. The World Is Not Enough: Another Look on Second-Order DPA. In Masayuki Abe, editor, *ASIACRYPT*, volume 6477 of *LNCS*, pages 112–129. Springer, 2010.

20. Nicolas Veyrat-Charvillon, Marcel Medwed, Stéphanie Kerckhof, and François-Xavier Standaert. Shuffling against Side-Channel Attacks: A Comprehensive Study with Cautionary Note. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT*, volume 7658 of *Lecture Notes in Computer Science*, pages 740–757. Springer, 2012.

A   RLUT Implementation Results

	Code Size	Data Size	Total
1 Round	1180	562	1742
2 Rounds	1180	1112	2292
3 Rounds	1180	1662	2842
4 Rounds	1180	2212	3392

Table 2. Program size of the LED cipher protected with RLUTs (in bytes).

	Randomness generation	RLUTs Generation	LED Execution	Total
1 Round	1422	41 910	404	43 736
2 Rounds	2208	83 807	642	86 657
3 Rounds	2991	125 716	883	129 590
4 Rounds	3770	167 617	1118	172 505

Table 3. Cycle counts of the LED cipher protected with RLUTs.