A First Step Towards Automatic Application of Power Analysis Countermeasures

Ali Galip Bayrak[1], Francesco Regazzoni[2,3], Philip Brisk[4],
François-Xavier Standaert[3], Paolo Ienne[1]

[1]École Polytechnique Fédérale de Lausanne (EPFL),
School of Computer and Communication Sciences, CH-1015 Lausanne, Switzerland.
{aligalip.bayrak,paolo.ienne}@epfl.ch
[2]ALaRI - University of Lugano, CH-6900 Lugano, Switzerland.
regazzoni@alari.ch
[3]UCL Crypto Group, Université Catholique de Louvain, B-1348 Louvain-la-Neuve, Belgium.
fstandae@uclouvain.be
[4]University of California, Riverside, 339 Engineering II, CA 92521 Riverside, USA.
philip@cs.ucr.edu

ABSTRACT

In cryptography, side channel attacks, such as power analysis, attempt to uncover secret information from the physical implementation of cryptosystems rather than exploiting weaknesses in the cryptographic algorithms themselves. The design and implementation of physically secure cryptosystems is a challenge for both hardware and software designers. Measuring and evaluating the security of a system is manual and empirical, which is costly and time consuming; this work demonstrates that it is possible to automate these processes. We introduce a systematic methodology for automatic application of software countermeasures and demonstrate its effectiveness on an AES software implementation running on an 8-bit AVR microcontroller. The framework identifies the most vulnerable instructions of the implementation to power analysis attacks, and then transforms the software using a chosen countermeasure to protect the vulnerable instructions. Lastly, it evaluates the security of the system.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems, smartcards*

General Terms

Design, Security

Keywords

Power Analysis Attacks, Software Countermeasure, AVR, Automation

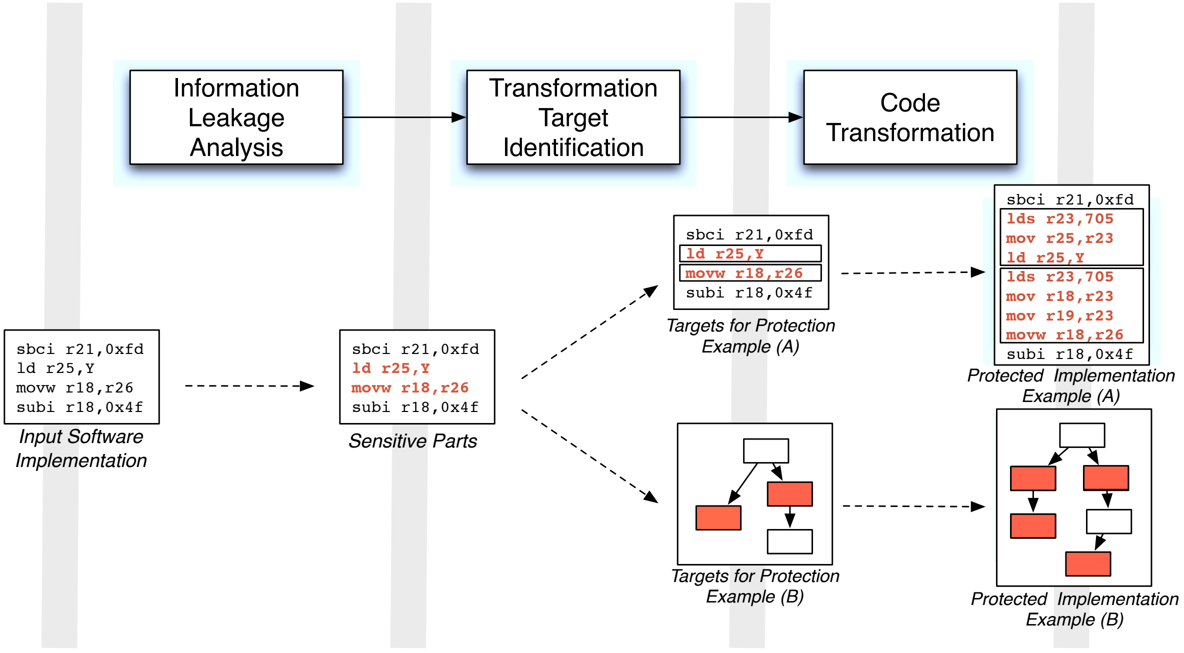
DAC 2011, June 5-10, 2011, San Diego, California, USA.

1. INTRODUCTION

Historically, attacks on cryptosystems have focused on exploiting mathematical weaknesses in cryptographic algorithms; *side channel attacks*, in contrast, attack the physical implementation of the system. Information such as power consumption [9], timing [8], or electromagnetic radiation [5] can reveal information that is otherwise secret. Both hardware and software countermeasures to these attacks have been introduced in prior literature; however, these countermeasures are generally inserted manually by Ph.D.-level experts who have a strong personal understanding of the cryptographic algorithm that they are protecting, along with its implementation. Even so, it is difficult to determine precisely which operations within a cryptographic implementation will actually leak side channel information.

This work makes a first attempt to automate the process of applying a given countermeasure to some implementation of a cryptographic algorithm. It does so by a sequence of steps indicated in Fig. 1: Firstly, our method takes the unprotected software implementation of a cryptographic algorithm and determines the instructions that leak the most information through a specific side channel (e.g., power consumption), irrespective of any specific attack which could be conceived (*Information Leakage Analysis*). Then, it identifies the exact targets of the code transformation which will make the code more robust to attacks (*Transformation Target Identification*). Such analysis is somehow related to the complexity of the countermeasures; in simple cases, as the one we have selected for our first experiment in this direction, each sensitive instruction or cluster of instructions is selected to apply code transformations to it; in more articulated case, a more rich data- and control-flow analysis may be necessary. Lastly, the code transformations implementing the countermeasure are applied to the identified targets (*Code Transformation*) and the result is protected code. How these steps are performed is explained in detail in the rest of the paper.

Our method successfully identifies sensitive instructions in a software implementation of AES running on an 8-bit AVR microcontroller. We automatically protect this software implementation using an application of *random precharging* [19].


Information Leakage Analysis
Transformation Target Identification
Code Transformation
Input Software Implementation
Sensitive Parts
Targets for Protection Example (A)
Protected Implementation Example (A)
Targets for Protection Example (B)
Protected Implementation Example (B)

```
sbci r21,0xfd
ld r25,Y
movw r18,r26
subi r18,0x4f
```

```
sbci r21,0xfd
ld r25,Y
movw r18,r26
subi r18,0x4f
```

```
sbci r21,0xfd
ld r25,Y
movw r18,r26
subi r18,0x4f
```

```
sbci r21,0xfd
lds r23,705
mov r25,r23
ld r25,Y
lds r23,705
mov r18,r23
mov r19,r23
movw r18,r26
subi r18,0x4f
```

Figure 1: Given a software, the target hardware platform, and a countermeasure, the sensitive parts of the program are identified and protected automatically using the given countermeasure.

In order to verify the effectiveness of this technique, we show its impact on information leakage analysis. Additionally, we performed a preliminary security analysis using a correlation-based *differential power analysis (DPA)*. We collected 20,000 power traces from the unprotected and from the automatically protected implementations of the AES software for this purpose.

2. INFORMATION LEAKAGE ANALYSIS

The purpose of the analysis of the information leakage is to identify the instructions which correspond to sensitive operations. The inputs of this process are the unprotected code and the hardware platform on which the protected code will run, and the output is a set of annotations of the original code which indicate how much critical information is leaked by the hardware system during the execution of each instruction. Fig. 2 illustrates this: each execution cycle of each instruction is annotated with an average value representing the relative information leakage. Instructions with high information leakage during their execution (above 0.4 in the example) are marked as *sensitive operations* and will be processed in the next step to identify the targets sections of the code to transform and protect.

This analysis consists of three main steps: (i) We start by compiling the given software implementation of a cryptographic algorithm for the target processor. Then, the executable is run on the processor with different *(plaintext, key)* pairs; power traces during the encryption process of each pair are recorded. During the measurement, we sample the power traces at high frequency (4GSa/s in our setup), and then compress the samples to get single power value for each clock cycle. For the compression, there are three popular methods which are shown to be very effective: *maximum extraction* [10], *integration* [10] and *principal component analysis* [1]. We used the first method. (ii) The second step is to analyze the traces using the metric that we introduce in Section 2.1; this determines the *sensitivity* of each clock cycle; sensitivity correlates strongly with information leakage, so a high sensitivity reading for a given clock cycle suggests that the instruction that executes during that cycle requires some protection. (iii) The last step is to associate each clock cycle with an assembly instruction from the dynamic execution trace. For multi-cycle instructions, we chose the most sensitive cycle as the instruction's sensitivity.

2.1 Metric for Sensitivity Evaluation

Our metric for sensitivity evaluation is based on an information theoretic metric originally proposed by Standaert et al. [18], which evaluates the resistance of a cryptographic implementation against the strongest possible power analysis attack. The metric establishes a relationship—i.e., mutual information—between the secret key that is used for encryption and the power traces. We limit the number of dimensions considered by the metric to 1, which makes it possible to simplify the formula. Since we are interested in observing the effects of single instructions, a 1-dimensional application of the metric is appropriate; higher dimensionality would be required in order to analyze second order effects.

Let K, X, and L respectively be random variables representing the secret key, plaintext, and information leakage from the physical device which is obtained via power trace analysis; and k, x and l be realizations of K, X, and L from an execution of the algorithm. Leakage L is normally distributed with mean μ and standard deviation σ—i.e., $\mathcal{N}(\mu_{k,x}, \sigma^2)$. The probability density function of L is

$$N_l(\mu_{k,x}, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{(l-\mu_{k,x})^2}{2\sigma^2}}, \qquad (1)$$

where $\mu_{k,x}$ represents the noiseless leakage value when (k, x) pair is executed and σ represents the constant noise standard deviation caused by the measurement. The conditional

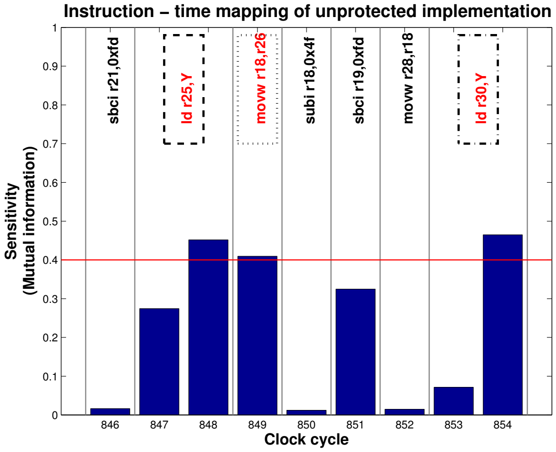


Figure 2: The Information leakage analysis associates an indicator of information leakage to each execution cycle, which in turn corresponds to a specific instruction being executed. The result is a set of sensitive instructions which will be object of the countermeasure in the subsequent stages of the automated protection process.

entropy of K given L is

$$H[K|L] = -\sum_{k} p(k) \cdot \sum_{x} p(x) \cdot \int p(l|k,x) \cdot log_2 p(k|l,x)dl, \tag{2}$$

which can be rewritten as

$$H[K|L] = -\sum_{k} \left\{ p(k) \cdot \sum_{x} \{ p(x) \cdot \right.$$
$$\left. \int_{-\infty}^{\infty} N_l(\mu_{k,x}, \sigma^2) \cdot log_2 \frac{N_l(\mu_{k,x},\sigma^2)}{\sum_{k^*} N_l(\mu_{k^*,x},\sigma^2)} dl \} \right\}. \tag{3}$$

The mutual information, which quantifies the sensitivity, is $I[K;L] = H[K] - H[K|L]$.

If the length of the plaintext and key are short, then it is possible to exhaustively enumerate all possible (X, K) pairs to compute the metric exactly; however, this is not generally the case: for example, AES-128 has 128-bit keys and plaintexts, which would require 2^{256} executions. To reduce the number of traces, we can exploit some properties of large deviation theory [22]: the result obtained from a randomly chosen subset of keys and plaintexts will be close to the result obtained from exhaustive enumeration with high probability, as long as the cardinality of the subset is sufficiently large. Our experiments demonstrate that the result converges for AES-128 when we consider 16 plaintexts in conjunction with 16 keys. We tried with different numbers of pairs and we observed that the instructions that can be classified as sensitive do not change after 8x8 pairs; so we used 16x16 pairs to ensure the fidelity of the results.

Note that the traditional approach for information theoretic evaluation is to apply a divide-and-conquer strategy, so that the key bytes can be exhaustively enumerated in order to obtain a precise estimation of the metric. In our present setting, we apply a slightly different approach as we do not aim to compare different countermeasures, nor to evaluate them precisely, but only to detect all the sensitive operations in an efficient manner. This approach could easily be combined with a divide-and-conquer one, where one would carefully estimate the amount of information leaked for various intermediate computations in the implementation.

3. TRANSFORMATION TARGET IDENTIFICATION

Depending on the chosen countermeasure, one needs to identify precisely where to insert a countermeasure in the vicinity of an instruction that has been identified as sensitive. For instance, if one were to use *masking*, it is necessary to define at what point in the code a variable should be masked (reasonably, before the first sensitive instruction) and at what point unmasked (after the last sensitive one). Additionally, semantic equivalence has to be preserved along every possible execution path between these two points, and this requires, for example, the identification of nonlinearities in the computations (such as S-box). The purpose of the transformation target identification process is to inform the compiler of where to insert the countermeasure.

For use in this study, we have selected a simple protection which, although effective, requires the simplest form of code transformation (the replacement of one or few instructions with one or few others—a *peephole* optimization). This implies that the target identification is practically trivial in our case: we will simply pass to the code protection engine the instructions whose sensitivity is above a chosen threshold (as in the example A of Fig. 1). We believe that target identification for more complex countermeasures is going to be definitely possible, as it is based on well known concepts of data- and control-flow analysis. This is graphically suggested for example B in Fig. 1 and we leave an investigation of this possibility open for future work.

4. CODE PROTECTION

The last step is to modify the code in the places identified previously. In this work, we have used *random precharging* method for protection, which requires only local code modifications. This mechanism randomly precharges the datapath before and after a critical instruction using random operand values. The software realization of the idea has been discussed by Tillich et al. [19]. Unlike Tillich et al., we do not need to use random charging *after* the critical instructions, since the critical instructions have already been identified.

In most of today's embedded systems, power consumption is dynamic and proportional to the Hamming distance between two consecutive cycles' data flowing through a wire, gate, or functional unit. If we randomize the values on the critical components, such as memory, register or data bus, the power consumption will also be randomized, since the Hamming distance between a uniformly distributed random variable and a fixed value has uniformly random behavior. Although the overall idea is the same, the operations performed for randomly charging the components might differ for each device, depending on the power consumption characteristics. Random precharging will not work for devices where the power consumption is proportional to the Hamming weight of the processed data, such as devices that use a precharged bus. Obviously, the countermeasure selected must be appropriate for the target device.

To apply random precharging properly, we ran some initial experiments to discern an appropriate way to implement

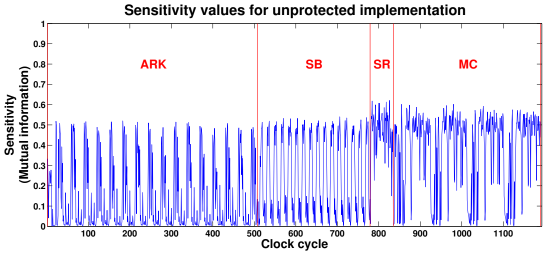


Figure 3: Sensitivity values of each clock cycle during the execution of one round of unprotected AES implementation. Higher sensitivity means less resistivity against power analysis attacks and needs protection. Fig. 2 is a detail of this same graph annotated with the instructions being executed.

the countermeasure on the given device—an 8-bit AVR microcontroller in our case. This needs to be done only once to understand the kind of transformations required, and is independent from the cryptographic algorithm. For instance, if the instruction `lds r24, 0xfae` is critical and needs to be protected, we replace it with the following three instructions:

```
lds r23, rnd ;rnd holds a random value
mov r24, r23 ;r23 is assumed to be unused
lds r24, 0xfae
```

The exact implementation of this peephole transformation is beyond the scope of this paper and is characteristic of the chosen countermeasure. As discussed by Tillich et al., random precharging does not guarantee perfect protection, and this limit is common for software countermeasures; however, it does increase the effort required to mount a successful attack. In the experimental section, we have provided the security analysis of the method.

5. EXPERIMENTAL RESULTS

We applied our automatic protection mechanism to an AES software implementation running on an 8-bit AVR microcontroller. The AES algorithm was implemented in C using AVR libraries and compiled with a `gcc` cross compiler using the optimization parameter `-Os` to get the unprotected assembly code. We used a straightforward implementation of AES without optimizations or security improvements, in order to see the effectiveness of our methodology on a naive implementation; however, the same method could be used to protect any other implementation as well. The code was assembled and loaded into the microcontroller and run with different randomly generated *(plaintext,key)* pairs to obtain power traces, which were then used to identify the sensitive instructions, as described in Section 2. Then, the necessary modifications on the unprotected assembly code are performed as described in Section 4, in order to increase the security. Finally, the security of both versions of the code is evaluated using different metrics.

In order to determine the noise standard deviation, σ, which is needed by the metric described in Section 2.1, we ran a small portion of the code that is independent of the key and plaintext to obtain power traces; the standard deviation was computed for each cycle, and we set σ to the maximum among all cycles.

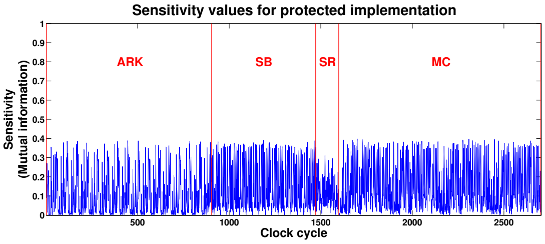


Figure 4: Sensitivity values of each clock cycle during the execution of one round of protected AES implementation. Sensitivity values decreased compared to the unprotected implementation, which means an increased security.

5.1 Measurement Setup

Our setup is comprised of a PC, microcontroller board, oscilloscope, and differential probe. We designed the microcontroller board, which includes an 8-bit AVR ATMEGA-8 microcontroller along with necessary supplementary components to facilitate power measurements. The internal RC oscillator of the microcontroller provides a 1 MHz clock. The microcontroller sets the trigger signal at the beginning of each run to align the traces. Voltage is measured across a 10Ω resistor that is connected in series to the microcontroller V_{cc} pin by the differential probe connected to the oscilloscope. The PC communicates with the microcontroller to execute code and to analyze the data collected by the oscilloscope. The components are calibrated to decrease electronic noise as much as possible; to eliminate random effects of noise, all measurements are repeated 25 times and averaged.

5.2 Identification of Sensitive Instructions

Fig. 3 shows the sensitivity values obtained for each clock cycle for the given implementation. The horizontal axis (time) is decomposed into the four main operations of the AES algorithm: AddRoundKey (ARK), SubBytes (SB), ShiftRows (SR), and MixColumns (MC). All four operations contain repeated patterns corresponding to information leaked from processing of different bytes of the state, which is a 4×4 array of bytes internal to the AES algorithm.

A non linear transformation, such as an SB, is a suitable attack point because an adversary can easily make a hypothesis on its output bits. Furthermore, the non-linear structure of the S-boxes highlights the differences between the correct and the wrong guesses and increases the possibility of a successful attack [14]. For microcontrollers, data transfers such as loads and stores are known to leak the most information, compared to, say, arithmetic and logical instructions. [10]. In Fig. 3, we can see that SR has the highest sensitivity peaks because it permutes the bytes of the state derived from the SB operation via load and store instructions, but does not modify the values of the bytes with any arithmetic or logical operations. Similarly, we can see high peaks during MC operation, at the clock cycles where the results derived from SB operation are loaded. In general, we observe that load operations tend to be more sensitive when compared to store operations, in our particular hardware. This does not explain the underlying mechanisms of the leakage, simply that it occurs. This would be a concern if our focus was hardware protection, but not for software.

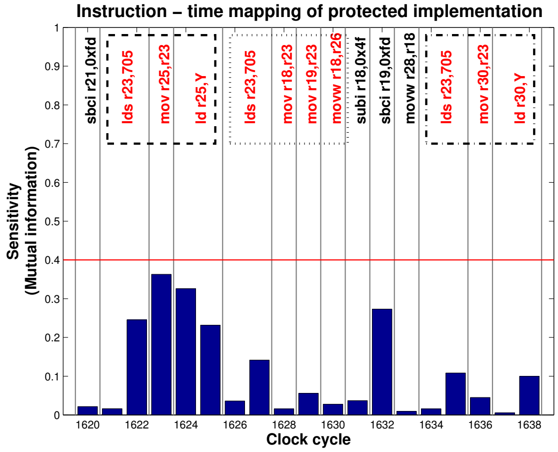


Figure 5: The same part of the program of Fig. 2 after automatic modification. The sensitivity has been reduced significantly and the effect of such reduction is assessed quantitatively in Section 5.4

We used `simulavr` tool together with `avr-gdb`, in order to simulate an execution of our code. After establishing a correspondence between sensitive clock cycles and the assembly instructions, we are ready to protect the sensitive instructions. We observed that after protection, the sensitivity of a critical instruction falls to just beneath 0.4, so we chose this value as our sensitivity threshold; in fact, 0.4 is the lower limit of protection that can be achieved using this particular software countermeasure. Instructions having greater sensitivity values are thus considered to be sensitive and are protected in the next step. Fig. 2 shows a detail of the data of Fig. 3 annotated with the instruction being executed; those in red bold typeface are sensitive.

5.3 Protecting the Sensitive Instructions

Assembly instructions that have been identified as sensitive are protected with random precharging, as described in Section 2. As we can see from Fig. 4, the sensitivity values decrease below the sensitivity threshold, most notably in SR and MC operations. Fig. 5 shows a detail of the data as in Fig. 2. As discussed before, random precharging does not completely prevent power analysis attacks; instead, it increases considerably the effort required to mount a successful attack. Again, this decrease in the sensitivity values is not equivalent to the information theoretic analysis proposed by Standaert et al. [18], hence does not provide any indication about the exact security of the countermeasures implemented. It is only used to confirm that the automated tools that we exploited are indeed able to reduce the leakage for all the sensitive operations.

5.4 Security Analysis

In order to evaluate the impact of our automatic application of countermeasures, the next step is to consider specific operations in our target implementation and to try taking advantage of their leakage with a standard divide-and-conquer approach. As an illustration, we used the output of the SB operations for this purpose. A complete evaluation would now require to analyze both the mutual information between the secret key byte involved in this S-box computation and the actual leakage, and the security against (i.e. success rates of) different actual distinguishers. Because of space constraints, we defer the complete application of this framework for an extended version of this work. Note that this part of the security evaluation tightly depends on the choice of a countermeasure that is not central in this work anyway (any countermeasure could be selected, with different tradeoffs between security and performance). Still, and as a first step, we applied a simple correlation power analysis using the Hamming weight leakage model, as done by Mangard et al. [10]. That is, was used 20,000 different power traces and computed Pearson's correlation coefficient, for each of the implementations. Figure 6 provides another heuristic confirmation that the proposed methodology is sound. A manual application of random pre-charging would lead to essentially the same results.

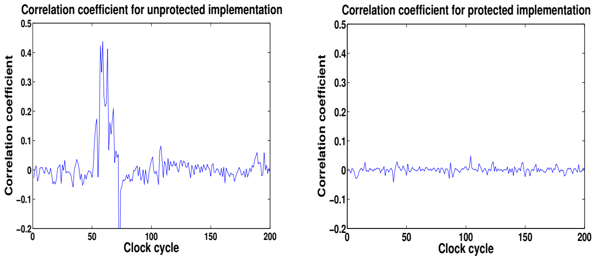


Figure 6: Correlation coefficient for unprotected and protected implementations. Higher correlation means decreased security. The number of needed power traces for a successful attack increases over 76 times for the protected implementation.

Table 1: Number of clock cycles during the execution of three different implementations.

Implementation	# of clock cycles
Unprotected	1190
Protect sensitive	2700
Protect everything	4212

5.5 Performance Analysis

Table 1 shows the number of clock cycles during the execution of three different implementations: the baseline (unprotected) implementation, the implementation generated by our framework, and a third implementation in which all instructions are protected regardless of their sensitivities. Protecting an instruction entails the insertion of additional instructions, so the third implementation is an upper bound on the runtime overhead that could result from an overzealous application of the countermeasure used in this study. Protecting only sensitive instructions yields a 36% runtime improvement compared to protecting all instructions.

6. RELATED WORK

Side channel attacks, particularly *Differential Power Analysis (DPA)*, represent a serious threat, since they do not require specific knowledge of the inner workings of the target device in order to be successful. The research community has aggressively developed countermeasures that protect against DPA. The countermeasures that have been proposed thus far, which are helpful, yet imperfect, include algorithmic techniques [4, 17], architectural enhancements [7, 11, 12], and hardware-related methods [13, 20, 16]; they all help to increase the efforts required to mount a successful attack.

At present, it is generally considered the designer's responsibility to ensure resistance of a system against DPA. The vast majority of prior work has addressed the problem from the perspective of hardware design and VLSI/CAD. For example, Tiri et al. [21] proposed a complete design flow for synthesis and place-and-route of the WDDL logic style, which is believed to be more resistant to DPA than traditional CMOS. Guilley et al. [6] introduced a back-end duplication to automate the place-and-route of DPA-resistant logic style. Regazzoni et al. [15], presented a fully automated design flow for realizing and simulating an embedded processor with instruction set extensions realized in MCML. A handful of projects have looked at DPA from perspectives other than hardware design, including the ongoing European project CACE [3]. Barbosa et al. [2] have analyzed the effects of a compiler on elliptic curve cryptography. Our work is similar to these papers, as we study a commercially available off-the-shelf processor without the possibility of inserting our own hardware-based countermeasures.

Most of the papers above try to identify the parts of a cryptographic algorithm that are most sensitive to power analysis attacks and protect these parts. Although this information is crucial to ensure correct implementation of countermeasures, the designer must perform these analyses manually and then decline a specific countermeasure for the program at hand. We believe to be the first ones in implementing the whole process automatically, albeit for a simple, well-behaved countermeasure.

7. CONCLUSIONS

This work has described an approach to automatically protect software implementations of cryptographic algorithm from power analysis attacks. The first step identifies the critical instructions of a cryptographic algorithm, in terms of how much side channel information they leak via power traces. The second step is to apply a software countermeasure to the critical instructions; our experiments used *random precharging*, but in principle, any fine-granularity software countermeasure could have been used. Lastly, we applied different tools for security analysis and provided precise performance estimates, for an implementation on an 8-bit AVR microcontroller. As a scope for further research, we aim to better investigate this tradeoff between security and performance. That is, we could analyze the impact of various types of countermeasures in function of these two criteria. For this purpose, it would be necessary to extend our correlation coefficient-based security analysis towards a more comprehensive one, considering both worst-case template attacks (i.e. an information theoretic analysis) and different heuristic distinguishers.

8. REFERENCES

[1] C. Archambeau, E. Peeters, F.-X. Standaert and J.-J. Quisquater. Template attacks in principal subspaces. In *Cryptographic Hardware and Embedded Systems –CHES 2006*, pages 1–14, 2006.

[2] M. Barbosa, A. Moss, and D. Page. Constructive and destructive use of compilers in elliptic curve cryptography. *Journal of Cryptology*, 22(2):259–281, April 2009.

[3] Computer Aided Cryptography Engineering (CACE European Project). http://www.cace-project.eu.

[4] J.-S. Coron and L. Goubin. On Boolean and arithmetic masking against differential power analysis. In *Cryptographic Hardware and Embedded Systems –CHES 2000*, pages 231–237, 2000.

[5] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems –CHES 2001*, pages 251–261, May 2001.

[6] S. Guilley, P. Hoogvorst, Y. Mathieu, and R. Pacalet. The "backend duplication" method. In *Cryptographic Hardware and Embedded Systems –CHES 2005*, pages 383–397, August 2005.

[7] J. Irwin, D. Page, and N. P. Smart. Instruction stream mutation for non-deterministic processors. In *13th International Conference on Application-Specific Systems, Architectures and Processors*, pages 286–295, July 2002.

[8] P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems. In *Advances in Cryptology –CRYPTO '96*, pages 104–113, September 1996.

[9] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology –CRYPTO '99*, pages 398–412, August 1999.

[10] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks - Revealing the Secrets of Smart Cards*. Springer, 2007.

[11] D. May, H. L. Muller, and N. P. Smart. Non-deterministic processors. In *Information Security and Privacy - ACISP '01*, pages 115–129, July 2001.

[12] D. May, H. L. Muller, and N. P. Smart. Random register renaming to foil DPA. In *Cryptographic Hardware and Embedded Systems –CHES 2001*, pages 28–38, May 2001.

[13] S. W. Moore, R. D. Mullins, P. A. Cunningham, R. J. Anderson, and G. S. Taylor. Improving smart card security using self-timed circuits. In *8th International Symposium on Advanced Research in Asynchronous Circuits and Systems - ASYNC 2002*, pages 211–218, April 2002.

[14] E. Prouff. DPA Attacks and S-Boxes. In *Fast Software Encryption –FSE 2005*, pages 424–441, 2005.

[15] F. Regazzoni, A. Cevrero, F.-X. Standaert, S. Badel, T. Kluter, P. Brisk, Y. Leblebici, and P. Ienne. A design flow and evaluation framework for DPA-resistant instruction set extensions. In *Cryptographic Hardware and Embedded Systems –CHES 2009*, pages 205–219, September 2009.

[16] F. Regazzoni, T. Eisenbarth, A. Poschmann, J. Großschädl, F. K. Gürkaynak, M. Macchetti, Z. T. Deniz, L. Pozzi, C. Paar, Y. Leblebici, and P. Ienne. Evaluating resistance of MCML technology to power analysis attacks using a simulation-based methodology. *Transactions on Computational Science*, 5430:230–243, 2009.

[17] A. G. Rostovtsev and O. V. Shemyakina. AES side channel attack protection using random isomorphisms. *Cryptology e-Print Archive*, March 2005.

[18] F.-X. Standaert, T. G. Malkin, and M. Yung. A unified framework for the analysis of side-channel key recovery attacks. In *Advances in Cryptology –EUROCRYPT '09*, pages 443–461, April 2009.

[19] S. Tillich and J. Großschädl. Power analysis resistant AES implementation with instruction set extensions. In *Cryptographic Hardware and Embedded Systems –CHES 2007*, pages 303–319, 2007.

[20] K. Tiri, M. Akmal, and I. Verbauwhede. A dynamic and differential CMOS logic with signal independent power consumption to withstand differential power analysis on smart cards. In *28th European Solid-State Circuits Conference*, pages 403–406, September 2002.

[21] K. Tiri and I. Verbauwhede. A digital design flow for secure integrated circuits. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 25(7):1197–1208, 2006.

[22] S. S. R. Varadhan. Large deviations. *Annals of Probability*, 36(2):397–419, 2008.