Cryptanalysis of the CHES 2009/2010 Random Delay Countermeasure

François Durvaux*, Mathieu Renauld**, François-Xavier Standaert***,
Loic van Oldeneel tot Oldenzeel†, Nicolas Veyrat-Charvillon‡

Université catholique de Louvain, UCL Crypto Group,
B-1348 Louvain-la-Neuve, Belgium.

Abstract. Inserting random delays in cryptographic implementations is often used as a countermeasure against side-channel attacks. Most previous works on the topic focus on improving the statistical distribution of these delays. For example, efficient random delay generation algorithms have been proposed at CHES 2009/2010. These solutions increase security against attacks that solve the lack of synchronization between different leakage traces by integrating them. In this paper, we demonstrate that integration may not be the best tool to evaluate random delay insertions. For this purpose, we first describe different attacks exploiting pattern recognition techniques and Hidden Markov Models. Using these tools, we succeed in cryptanalyzing a (straightforward) implementation of the CHES 2009/2010 proposal in an Atmel microcontroller, with the same data complexity as an unprotected implementation of the AES Rijndael. In other words, we completely cancel the countermeasure in this case. Next, we show that our cryptanalysis tools are remarkably robust to attack improved variants of the countermeasure, e.g. with additional noise or irregular dummy operations. We also exhibit that the attacks remain applicable in a non-profiled adversarial scenario. Overall, these results suggest that the use of random delays may not be effective for protecting small embedded devices against side-channel leakage. They also confirm the need of worst-case analysis in physical security evaluations.

1 Introduction

Protecting small embedded devices against side-channel attacks is a challenging task. Following the DPA book [17], masking and hiding are two popular solutions to achieve this goal. Masking can be viewed as a type of data randomization technique, in which the sensitive (key dependent) intermediate values in an implementation are split in different shares. One of its important outcomes is that, under certain physical assumptions (e.g. that the leakage of the different

* PhD student funded by the Walloon region MIPSs project.
** PhD student funded by the Walloon region SCEPTIC project.
*** Associate researcher of the Belgian fund for scientific research (FNRS-F.R.S.).
† PhD student funded by the 7th framework European project TAMPRES.
‡ Postdoctoral researcher funded by the ERC project 280141 (acronym CRASH).

shares can be considered as independent), the security of a masked implementation against side-channel attacks increases exponentially with the number of shares [3]. On the drawbacks side, masking usually implies significant performance overheads. In addition, the exponential security increase it theoretically guarantees is only effective when the amount of noise in the measurements (e.g. power or EM) is sufficient [25]. Hence, it is hardly useful as a standalone countermeasure for small cryptographic devices, and usually has to be combined with hiding. Roughly speaking, hiding aims at reducing the side-channel information by adding randomness to the leakage signal (rather than to the data producing it) and can take advantage of different methods. For example, the direct addition of physical noise, or the design of dual-rail logic styles [27], are frequently considered options. Exploiting time randomization is a more flexible alternative, as it requires less (or no) modification of the underlying hardware.

Among the different time randomization techniques proposed in the literature, e.g. [5, 14, 28], one can generally distinguish the software ones, e.g. based on Random Delay Interrupts (RDIs), from the hardware ones, e.g. based on increasing the clock jitter. Quite naturally, the more hardware-flavored is the countermeasure, the more signal-processing oriented are the solutions to overcome them [11, 19, 29]. In this paper, we pay a particular attention to the software-based solutions exploiting RDIs. In this setting, it is interesting to notice that most previous evaluations of the countermeasures impact (e.g. [16]) pre-process the leakage traces by integrating them. Somewhat biased by this evaluation technique, recent works such as the ones of Coron and Kizhvatov at CHES 2009/2010 [6, 7] mainly focused on how to improve the statistical distribution of the random delays, in order for their integration to produce the most noisy traces. However, looking at the source codes provided in these papers, that alternate actual cipher computations with dummy operations, a natural question is to ask whether techniques based on pattern recognition could not be used to directly remove the delays. In other words, could it happen that, at least in certain contexts, this countermeasure can be strongly mitigated, or even reversed.

In this paper, we answer this question positively and show that, when implemented in an Atmel 8-bit microcontroller, designs protected with the CHES 2009/2010 countermeasures can be as easy to attack as unprotected ones. We start by observing that simple tools based on correlation analysis can be used to detect different types of patterns in leakage traces. For example, one can (even visually) identify the RDI headers and dummy loops, or block cipher operations such as AddRoundKey, SubBytes or MixColumn in the AES, opening the door to various attacks. On the one hand, this suggests that integrating the leakage traces is definitely not the appropriate approach to evaluate the security of such implementations. On the other hand, heuristic tools based on correlation analysis are inherently limited in more complex situations, where the dummy operations are less regular than in [6, 7], or when the random delays exploit the hardware interrupt feature of the underlying microcontroller. For this purpose, the second part of the paper takes advantage of Hidden Markov Model (HMM) cryptanalysis, as a generic modeling tool to capture any variant of the RDI countermeasure.

As previously observed in [10, 15], HMMs provide a very natural tool to deal with implementations in which some operations are randomized. We show experimentally that by adequately modeling a protected AES implementation as a HMM, we are able to produce traces that exactly correspond to the ones of an unprotected implementation, with very high probability. Eventually, it remains that the addition of RDIs prevents the use of averaging to improve the quality of an adversary's measurements. Hence, we additionally evaluate the amount of noise that should be added to our measurements, in order for the countermeasure to become actually effective (i.e. to get closer to the security increases predicted in previous works using integration of the traces). It turns out the the application of HMMs is remarkably robust to noise addition. We conclude the paper by discussing possible improvements of the countermeasure and their limitations, as well as a non-profiled variant of our HMM-based cryptanalysis.

We note that the authors in [6, 7] mention that it could be possible to detect random delays because of their regular pattern. They propose to hinder this by processing random dummy data during the delays, and acknowledge that this problem is not addressed in their papers. In the next sections, we show that identifying random delays in the power traces can be easy and that processing random dummy data is not sufficient to prevent attacks exploiting HMMs.

Related work. In a recent and independent work, Strobel and Paar investigated the use of pattern matching for removing random delays in embedded software. Their proposal can be viewed as an alternative to our correlation-based techniques in Section 3. Namely, the work in [26] uses pattern matching in order to detect each random delay *independently* and exploits *hard information* made of a string of Hamming weights obtained from power measurements. By contrast, our method in Section 4 models the complete assembly code of a protected AES implementation as a HMM (i.e. considers all the delays *jointly*) and exploits *probabilistic information* from the power traces. As a result, we obtain a better robustness to noise and hence, a more objective evaluation tool.

2 Background: the CHES 2009/2010 countermeasure

Overall, the addition of random delays in an implementation can be viewed as a trade-off between performance overheads (measured in code size and cycle count) and the variance added to the position of a target operation in side-channel measurement traces. In this section, we describe the countermeasure introduced at CHES 2009 (and improved at CHES 2010) by Coron and Kizhvatov, highlight their important characteristics and present our implementation.

Summarizing, both proposals focus on finding a good statistical distribution for the (random) lengths of the delays. First, the CHES 2009 paper analyzes the so-called floating point method. Its goal is to decrease the average total length of random delays while increasing the variance they imply for the position of side-channel attack target samples. Next, in the CHES 2010 paper, the authors remark that a bad choice of parameters for the floating mean method can lead to

security weaknesses. As a result, they proposed an improved solution, together with a new criterion to evaluate the security of RDI. In both cases, their implementations ran on an 8-bit Atmel AVR platform, similar to the one we consider in this paper. In practice, the random delays were inserted at 10 places per AES round: once before `AddRoundKey`, once before each 32-bit `SubBytes` block, once before each 32-bit `MixColumn` block and once after the last `MixColumn` block.

Our implementation of the RDI countermeasure followed the same guidelines as in [6, 7] and was based on the AES-128 "furious" design, available as open source in [1]. Note that, as our goal is to identify and remove the delays from the traces, the actual distribution of their lengths has no incidence on our results. In other words, our focus is on *how* the delays are inserted in the normal flow of the AES instructions, not on *how much* delay is inserted at each step[1].

Algorithm 1 Random delay insertion function

```
randomdelay:
   (1) rcall randombyte                    3 cycles
   (2) tst RND                             1 cycle
   (3) breq zero                           1 cycle (2 if true)
   (4) nop                                 1 cycle
   (5) nop                                 1 cycle

dummyloop:
   (6) dec RND                             1 cycle
   (7) brne dummyloop                      2 cycles (1 if false)

zero:
   (8) ret                                 4 cycles

randombyte:
   (9) ld RND, X+                          2 cycles
   (10) ret                                4 cycles
```

More precisely, the code we used in our evaluations is given as Algorithm 1. It is essentially the same as the one presented in [6], with the simplified `randombyte` function that only fetches some random numbers from a register[2], and can be read as follows. Whenever a random delay needs to be inserted, the `randomdelay` function is called. This function in turn calls (`rcall`) the `randombyte` function that provides a value `RND` (that has to be carefully chosen in order to get good statistical distribution for the delay lengths). Depending on the value `RND`, there

[1] Similarly, the recommendation to add 3 dummy AES rounds with delays at the start and at the end of the AES computation, so that the target operations at the first and last rounds are separated from the synchronization points by at least 32 random delays, has little impact on our attacks exploiting HMMs.

[2] A fully functional implementation would require a more complex `randombyte` function, further simplifying the detection of the random delay function.

are two cases: either `RND` $= 0$ and the function directly terminates by calling `zero` and returning (`ret`) to the normal flow of the AES instructions; or `RND` $\neq 0$ and we enter the `dummyloop`. This loop simply consists in decrementing (`dec`) `RND` until it reaches 0, and the function terminates. The right part of Algorithm 1 indicates the number of cycles required by the different operations in our Atmel device. Hence, not considering the case where `RND` starts at 0, each delay is constituted of a header of 16 cycles, (`RND` $- 1$) dummy loops of 3 cycles, a final dummy loop of 2 cycles, and at last a `ret` instruction of 4 cycles.

3 Pinpointing useful operation leakages

The previous section described the RDI countermeasure and the source code that we ran in an Atmel microcontroller. In this section, we show that the different operations in this target device produce significantly different leakages, that can be detected with simple tools based on correlation analysis. Beforehand, we briefly describe the setup we used to perform our experiments.

3.1 Measurement setup and preprocessing

Our target device is an Atmel ATMega644P. Its power consumption has been measured at maximum clock frequency (i.e. 20Mhz[3]) and taken over a shunt resistor inserted in the supply circuit of the microcontroller. Sampled data was acquired with a Tektronix TDS7k oscilloscope. In order to facilitate our attack, we applied a simple preprocessing step. Namely, we first split the traces into consecutive clock cycles, using the Fast Fourier Transform to recover the rising edges of the clock signal. This was achieved by filtering the frequency spectrum around the clock frequency and its harmonics, then applying the inverse transform on the filtered signal. This preprocessing provides a sequence of peaks indicating where the rising edges of the clock signal are. Following, we were able to work on a sequence of clock cycles instead of raw side-channel traces. It both reduced the difficulty of the attack and its computational cost. The steps of this filtering are illustrated in Appendix, Figure 7: the original trace (a) is first filtered in the spectral domain around the harmonics of the clock signal (b). By taking the local maxima, we get a decomposition of the trace into clock cycles.

3.2 Correlation based attacks

Let us first have a look at the power traces of an AES implementation protected with the RDI countermeasure. As illustrated by Figure 1, a simple visual inspection allows determining the different parts of the code. In other words, there are significant operation leakages that can be detected with Simple Power Analysis. Two main approaches can be considered for this purpose.

[3] This work frequency result in slightly more noisy traces than if running at lower frequency. However, as our goal was to show that the proposed cryptanalysis is robust to noise, we did not make efforts in improving this part of our setup. As will be clear in the next section, we event went the opposite way and artificially added noise to our measurements in order to discuss this robustness issue.

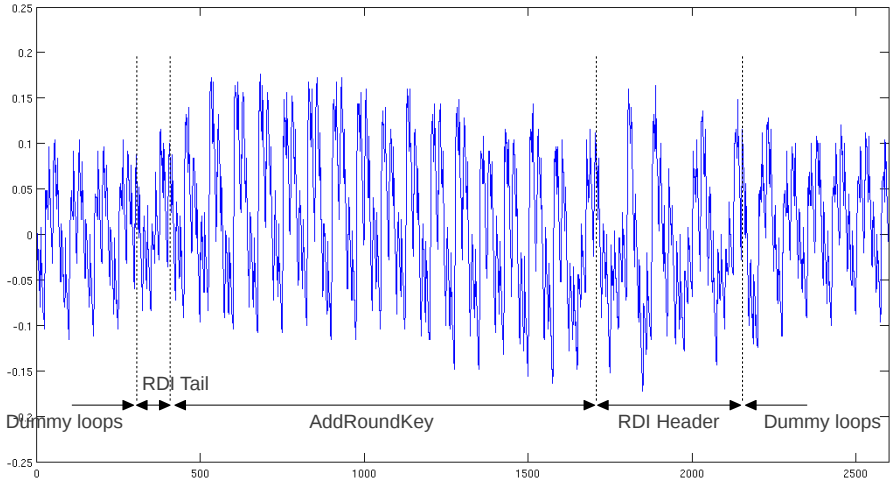

Fig. 1. AddRoundKey operation protected with the random delay countermeasure.

On the one hand, one can target the dummy operations, i.e. find the clock cycles during which the `dummyloop` is executed. As emphasized in Figure 2, random delays present a very distinctive outlook, since they contain short repetitive patterns, each loop lasting only three clock cycles. As a result, given that one can extract the pattern of this loop (including its header or tail), it is possible to match this pattern to clock cycles in the side-channel traces, by computing the cross-correlation between them. Eventually, the adversary can filter the traces by removing any cycle which is highly correlated with the delay pattern and attack the filtered traces, e.g. with Correlation Power Analysis (CPA) [2].

On the other hand, the adversary can also target the actual AES operations, instead of the delays. Indeed, while the inserted delays are of variable length and their shape can be changed, the AES operations are fixed and *have* to be executed by the program. It turns out that this detection is specially easy when large sequences of operations are executed without dummy operations. For example, AddRoundKey is rather distinctive in the proposal of [6, 7], as it consists in sixteen three-cycle loops surrounded by RDI delays. Hence, if the part of the power trace that corresponds to this sequence of operations can be extracted in a preliminary profiling, it can be compared with other side-channel traces using cross-correlation, just as for the detection of dummy operations. An exemplary result of this detection technique is given in Figure 3. It highlights that significant information on the executed operations is available in our measurements, and that integrating traces is not the best approach for analyzing RDIs.

These preliminary results are worth a few general words of discussion.

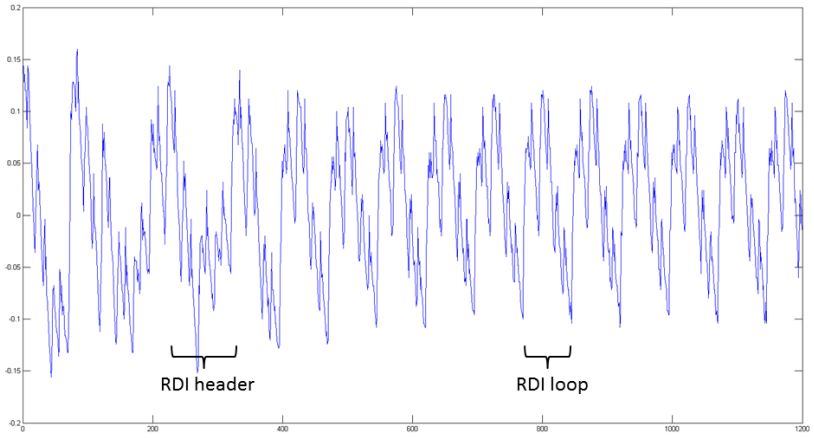

Fig. 2. Zoom on the RDI header and dummy operations.

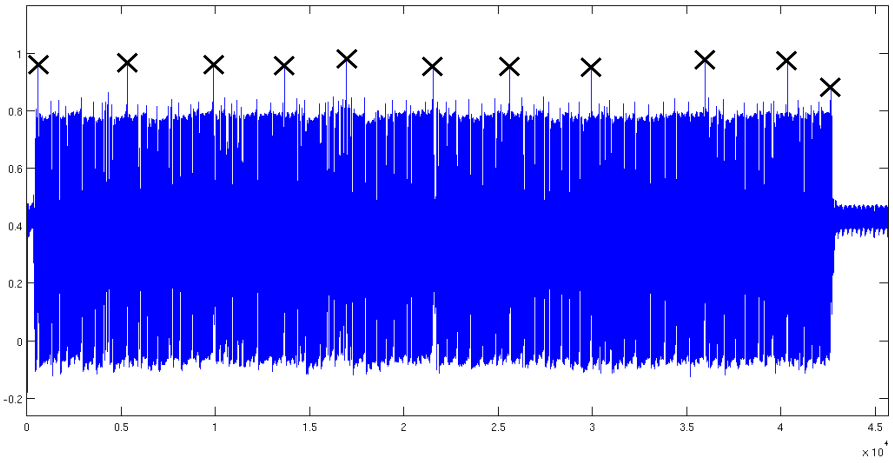

Fig. 3. Cross-correlation between an AddRoundKey pattern and one protected trace.

1. On improving the countermeasure. In the first place, the previous figures admittedly target the direct application of the CHES 2009/2010 countermeasures. However, while the choice of dummy operations to execute has little or no impact on "integrating" attacks, it is critical when playing with pattern matching as we undertake in this paper. In particular, two simple improvements could be implemented. First, one could use AES operations in the dummy cycles. Second, the hardware interrupts of the Atmel microcontroller could be exploited, in order for the RDIs to occur at less predictable places (e.g. the guarantee that AddRoundKey is executed as a single block would vanish in this case).

2. On the heuristic nature of the correlation-based approach. Second, it is worth emphasizing that the previous exploitation of cross-correlation is essentially heuristic. While it is intuitively useful to put forward a risk of attacks, it is also limited and hardly generic. In particular, if the aforementioned improvements were implemented, cross-correlation based attacks would become ineffective.

3. On the meaning of Kerckhoffs' principles in implementation attacks. Eventually, the analysis of RDIs raises the question of what the adversary exactly knows about his target implementation. In general, cryptographers like to consider that most possible information (e.g. about the algorithms) is public and that security only relies on the secrecy of a key. Straightforwardly translating this principle in the physical world would imply that source codes are given to the adversary, a condition that may not always be found in the field though. Such a question directly relates to the question of profiled vs. non-profiled attacks as well. For example, in the previous discussion of correlation-based attacks, is it realistic that the adversary can build an approximate pattern for the dummy operations or AES operations? In the following, we will essentially investigate the case where the answer is yes and justify this choice with three main reasons. Besides, we also provide a non-profiled solution to our problem in Section 4.6.

1. In practice, the gap between profiled and non-profiled attacks, and the lack of knowledge about the underlying hardware and implementation, can usually be overcome in the long term. Examples of solutions to reduce this gap include the use of non-profiled stochastic models [9, 21], or techniques inspired from side-channel attack reverse engineering [8, 12, 20].
2. In general in cryptography, a security evaluation has to look for worst cases, and this also applies to implementation attacks [24]. Hence, regardless the practical relevance of certain adversarial scenarios, it is essential to consider them, in order to have a fair understanding of the actual security level provided by cryptographic implementations. Practical security can of course be higher than what is lower-bounded by worst-case attacks.
3. Sound countermeasures against side-channel attacks should provide additional security even in the worst cases. For example, if an adversary is given a masked implementation together with an accurate description of its design

and source code (including the exact time instants when the shares are manipulated), the analysis of Chari et al. [3] still holds and the data complexity of an attack against this implementation can still be high enough[4].

4 Hidden Markov model cryptanalysis

Having justified the need of optimal evaluation tools, this section investigates a new cryptanalysis of RDIs based on HMMs. We argue that it constitutes an interesting generic tool to capture our problem. In particular, and contrary to the correlation-based techniques, it can easily deal with any type of dummy operations and interrupt processes. As mentioned in introduction, this approach follows previous works in the field of reverse engineering and cryptanalysis of randomized implementations, where similar principles have been used [10, 15]. In particular, the work of Karlov and Wagner is very similar to ours as it exploits HMMs to break the randomized exponentiation algorithms. To a certain extent, removing random delays in side-channel traces can also be seen as a simplified reverse engineering problem. That is, Eisenbarth et al. intend to build a disassembler, in order to extract an exact sequence of unknown instructions being executed by a device. We follow the same goal in the case where the instructions are known but the number of dummy loops that are executed for each delay is unknown. In the following, we first explain how to translate the RDI detection as a HMM problem. Next, we describe how to actually remove the delays from side-channel leakage traces and present results of experimental attacks against our randomized implementation. Eventually, we discuss possible improvements of the countermeasure as well as a non-profiled variant of the attack.

4.1 Building the HMM

A Markov model is a (memoryless) system with a finite number of states, for which the probabilities of transition to the next state only depend on the current state. It is thus constituted of a set of states π_i's and a transition probability matrix T. $T(i, j)$ is the (*a priori*) probability that the next state is π_j if the current state is π_i. If we denote with s_t the current state of the system at time t, $T(i, j) = \Pr(s_{t+1} = \pi_j | s_t = \pi_i)$. In the case of a Hidden Markov model, the sequence $\mathbf{s} = (s_0, s_1, ..., s_n)$ of the states occupied by the system is not known. However, the adversary has access to (at least) one observable that gives partial information about this sequence. Namely, at each time step t, a random vector $\mathbf{l}_t$ is observed by the adversary. In addition to the transition probability matrix, the HMM is then characterized by the emission probability functions associated to each state π_i, namely: $e_i(\mathbf{l}_t) = \Pr(\mathbf{l}_t | s_t = \pi_i)$ (see Figure 4).

[4] By contrast, the time complexity of the attack, that essentially depends on the number of time samples to test in the leakage traces, can be very small in such cases. This confirms that the (theoretical) soundness of masking first comes from the need to estimate higher-order moments in the leakage distributions, not from the increased time complexity. Again, this does not prevent the need to test large number of samples to be the most difficult to deal with in practice [25].

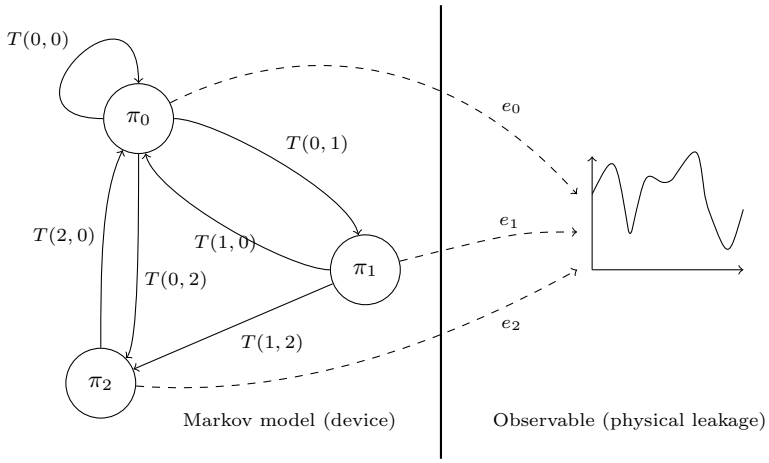

Fig. 4. Hidden Markov Model. The Markov model represents the device executing a sequence of instructions. The observable is the physical leakage.

In the case of our protected AES implementation, the Markov model describes the encryption process, with each state π_i associated to an instruction (e.g. `NOP`, `RET`, ...). Some instructions take only one clock cycle, but others require several clock cycles to be completed. As each state should correspond to the same number of clock cycles, longer instructions are split in different states, e.g. `RCALL` taking 3 cycles is split in three states associated to `RCALL` 0, `RCALL` 1 and `RCALL` 2. We call *instruction cycle* the (part of an) instruction associated with a state π_i. The list of all the 26 instruction cycles[5] appearing in the code of the protected AES can be found in Table 1. Note that the same instruction cycle can be used at multiple places in the AES code, corresponding to different running states, e.g. $\pi_0 \leftrightarrow$ `MOV` 0, $\pi_1 \leftrightarrow$ `EOR` 0, $\pi_2 \leftrightarrow$ `MOV` 0, ...

More precisely, the protected AES code can be divided in two types of instruction sequences: the deterministic instruction sequences and the dummy loops. In the deterministic sequences, the instruction cycle associated to a state is fully determined by its position in the sequence. In order to model these deterministic sequences, we can simply use one different state per successive cycle in the sequence, with deterministic transition probabilities: $T(i, i+1) = 1$ (see the top of Figure 5). By contrast, the non-deterministic dummy loops are constituted of a branching (`BRNE`) and a decrement (`DEC`) instruction. This translates into 4 instruction cycles: `BRNE T` 0, `BRNE T` 1 and `DEC` 0 in the loop (i.e. while the counter is decremented), and eventually `BRNE F` 0 to end the loop. There are two main ways to encode these loops in a Markov model. The simplest one consists in using 4 states, as presented in the middle part of Figure 5. The transition

[5] For conditional instructions, the number of cycles is different whether the condition is true or false. We consider thus these two cases as separate instructions.

Index	Cycle	Index	Cycle	Index	Cycle
0	NOP 0	9	RET 2	18	BRNE T 1
1	RCALL 0	10	RET 3	19	MOV 0
2	RCALL 1	11	TST 0	20	EOR 0
3	RCALL 2	12	BREQ F 0	21	LPM 0
4	LDI 0	13	BREQ T 0	22	LPM 1
5	LD 0	14	BREQ T 1	23	LPM 2
6	LD 1	15	DEC 0	24	RJMP 0
7	RET 0	16	BRNE F 0	25	RJMP 1
8	RET 1	17	BRNE T 0		

Table 1. List of instruction cycles occurring in the protected AES implementation.

probabilities at the output of the DEC state are p to restart another loop, and $1-p$ to exit the loop. This representation uses a fixed probability p, that cannot be changed based on the number of loops previously executed. It is thus not possible to render the details of the probability distribution of the delay lengths. Another way is to "unfold" the dummy loops by using $255 \times 3 + 1$ states, as in the lower part of Figure 5 (255 being the maximum number of dummy loops in one delay). It is then possible to fine tune the probabilities p_1, p_2, ..., in order to match the probability distribution of the delay lengths as closely as possible.

In terms of performances, the second representation is more precise and allows capturing the AES implementation more accurately, potentially leading to a better robustness against noisy measurements. However, its larger number of states also implies a more complex resolution phase (in time and memory). As experimental results showed that the compact representation was sufficient to obtain successful attacks with low data complexity, we focus on this one in the rest of the paper. Eventually, our Markov model for the protected AES implementation has approximately 6000 states, each of them associated to one of the 26 different instruction cycles from Table 1. The corresponding transition matrix T is very sparse, as the sequence of instructions is deterministic most of the time (in which case $T(i, i+1) = 1$ is the only non-zero value of the i^{th} line).

4.2 Building the templates

Once the Markov model corresponding to the protected AES is determined, we still have to estimate the emission probability functions $e_i(\mathbf{l}_t) = \Pr(\mathbf{l}_t | s_t = \pi_i)$ corresponding to each state π_i. We make two assumptions for this purpose:

1. The power trace $\mathbf{l}$ can be cut cycle in cycles: $\mathbf{l} = (\mathbf{l}_0, \mathbf{l}_1, ...)$. This is possible using the FFT-based technique of Section 3.1. We denote with *trace cycle* each part $\mathbf{l}_i$ of the power trace corresponding to one clock cycle. Trace cycles are measured vectors of 25 values (obtained with the setup of Section 3.1).
2. The emission at time t, denoted as $\mathbf{l}_t$, only depends on the type of the instruction executed at time t. This assumption is admittedly not entirely accurate (because data dependencies, pipeline effects, ... are neglected). But it again turned out to be sufficiently respected for our attacks to succeed.

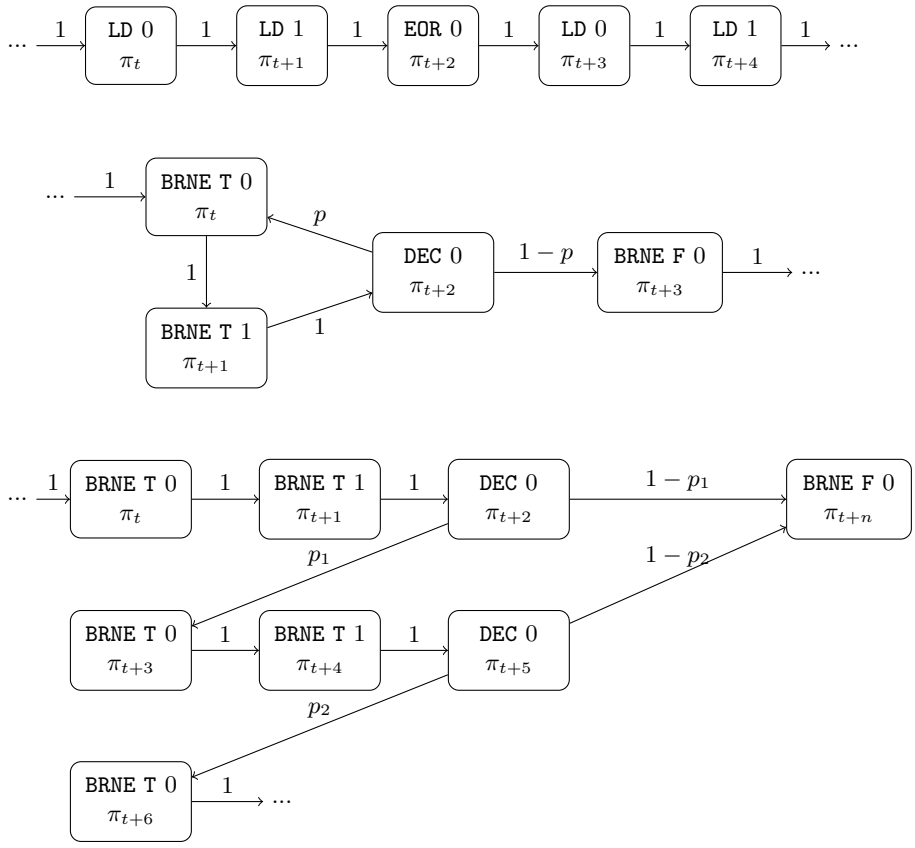

Fig. 5. Parts of the Markov model. Above: beginning of the `addRoundKey` operation (deterministic sequence). Middle and below: dummy loop. Each node is a different state, each edge is a non-zero transition probability, specified above the arrow.

Estimating the emission probability functions is equivalent to building a template for each state π_i. But contrarily to standard side-channel template attacks, where templates are built from a single instruction in order to distinguish the data being processed (e.g. [4]), we build templates for the different instruction cycles in order to tell them apart. That is, as our Makov Model contains 6000 states, we could build up to 6000 templates. But thanks to our second assumption, the same template can be used for every state π_i associated to the same instruction cycle from Table 1. Hence only 26 templates are required for the 6000 different states. Next, the emission probability functions are directly determined by these templates: for an instruction cycle i, we have the function $e_i(\mathbf{l}_t) = \mathcal{N}(\mathbf{l}_t|\hat{\boldsymbol{\mu}}_i, \hat{\boldsymbol{\Sigma}}_i)$. For each template, the mean vector $\hat{\boldsymbol{\mu}}_i$ and correlation matrix $\hat{\boldsymbol{\Sigma}}_i$ are estimated from a set of trace cycles $\mathbf{l}_t$ corresponding to the same instruction cycle i.

In order to build these 26 templates, we thus need to find sets of trace cycles $\mathbf{l}_t$ corresponding to each possible instruction cycle. However, due to the unknown lengths of the delays in a protected implementation, it is impossible to directly match trace cycles to their corresponding instruction cycles. As a result, we used a *profiling phase* for this purpose. That is, we built the templates from a training device for which the length of the delays was fixed. In addition, we used this profiling phase in order to efficiently reduce the dimensionality of our traces cycles, using the Principal Component Analysis (PCA) technique described in [23]. PCA consists in linearly projecting the data (i.e. the trace cycles) on a lower dimensional subspace, in such a way that the variance between the different instruction cycles is maximized. In practice, we kept 3 dimensions per template, which appeared to be a good compromise between the amount of variability we retain and the complexity of the parameters we need to evaluate.

4.3 Removing the delays

Given properly profiled templates, all the parameters of our HMM are fixed: the state vector π, the transition probability matrix T and the emission probability functions e_i. Hence, it only remains to identify the state sequence $\mathbf{s} = (s_0, s_1, ...)$ that is the best match for a given observation sequence $\mathbf{l} = (\mathbf{l}_0, \mathbf{l}_1, ...)$. The Viterbi algorithm can be launched for this purpose, as described with Algorithm 2 and briefly explained as follows. Let us consider a HMM with N_s states, and a sequence of N_o observations. A probability table $V \in \mathbb{R}^{N_s \times N_o}$ is first built, such that each value $V(i,j)$ is the probability of the most probable sequence of states ending in state π_i, given the sequence of observations $(\mathbf{l}_0, ..., \mathbf{l}_j)$. The first column of V is initialized with the probabilities for the different states to match the first observation $\mathbf{l}_0$: $V(i,0) = p_i \cdot e_i(\mathbf{l}_0)$ (where the p_i's are the initial a priori probabilities that the system starts in state π_i). Next, for each additional observation $\mathbf{l}_t$ in the sequence, the probabilities for the different states are determined by $V(i,t) = e_i(\mathbf{l}_t) \cdot \max_j(V(j,t-1) \cdot T(j,i))$. These probabilities take into account the emission probability of the current observation (i.e. $e_i(\mathbf{l}_t)$), but also the most probable sequences of length $t-1$ (i.e. $V(j,t-1)$), weighted by the transition probabilities $T(j,i)$'s. A matrix $I \in \mathbb{N}^{N_s \times (N_o-1)}$ is finally used to store, for each

step $t \geq 1$ and each state i, the index j of the most probable state sequence of length $t-1$ that can lead to state i: $I(i,t-1) = \text{argmax}_j(V(j,t-1) \cdot T(j,i))$. Once the full sequence of observation is processed, the algorithm selects the most probable ending state, and then backtracks to the first observation using the matrix I to select at each step the previous state in the most probable sequence.

Algorithm 2 Viterbi algorithm

input: a HMM characterized by a set of N_s states π_i, initial probabilities p_i, a transition probability matrix T and emission probability functions e_i.
input: a sequence of N_o observations $\mathbf{l} = (\mathbf{l}_0, \mathbf{l}_1, ...)$.
output: the most probable sequence of states corresponding to the observations.
Define V a new matrix in $\mathbb{R}^{N_s \times N_o}$.
Define I a new matrix in $\mathbb{N}^{N_s \times (N_o - 1)}$.
//Initial probabilities.
for $i = 0$ to $i = N_s - 1$ **do**
  $V(i,0) \leftarrow p_i \cdot e_i(\mathbf{l}_0)$
end for
//Computing the probabilities.
for $t = 1$ to $t = N_o - 1$ **do**
  for $i = 0$ to $i = N_s - 1$ **do**
    $V(i,t) \leftarrow e_i(\mathbf{l}_t) \cdot \max_j(V(j,t-1) \cdot T(j,i))$
    $I(i,t-1) \leftarrow \text{argmax}_j(V(j,t-1) \cdot T(j,i))$
  end for
end for
//Backtracking to find the most probable path.
Define $\mathbf{s}$ a new vector of size N_o.
$\mathbf{s}(N_o - 1) \leftarrow \text{argmax}_j(V(j, N_o))$
for $t = n_o - 2$ to $t = 0$ **do**
  $\mathbf{s}(t) \leftarrow I(\mathbf{s}(t+1), t)$
end for
return $\mathbf{s}$

4.4 Results of the HMM method and impact of the noise

We evaluated the efficiency of the HMM method by using it against power traces measured from a protected implementation of the AES on our 8-bit Atmel microcontroller. For comparison purposes, we also investigated the security of a non-protected implementation running on the same platform, in order to evaluate the security improvement offered by RDIs. Since we are considering first-order (standard) DPA attacks (as defined in [18]), we chose to run CPA to illustrate the unsecurity of these implementations. As shown in Figure 6, the success rate of CPA against the protected implementation with actual noise level (corresponding to the curves with no additional simulated noise in the figure, i.e. $\sigma^* = 0$) is nearly the same as for the CPA against the unprotected implementation. Note that for each curve, we estimated the first-order success rate defined in [24],

from a set of 100 independent experiments. In fact, the slight difference between protected and unprotected implementations can be explained by the postprocessing of the traces after removing the RDIs, because of slight imperfections of the cycles cut. Otherwise, the actual removing of the delays was close to 100% successful. As a result, both implementations can be broken after approximately 100 measurements. For comparison, the integration attack on the protected implementation in [6] requires approximately 45 000 power traces to succeed. We conclude that (1) the HMM method is much more efficient than the integration method to attack the RDI countermeasure, and (2) with the actual noise level of an Atmel 8-bit microcontroller, the RDI countermeasure is completely reversible.

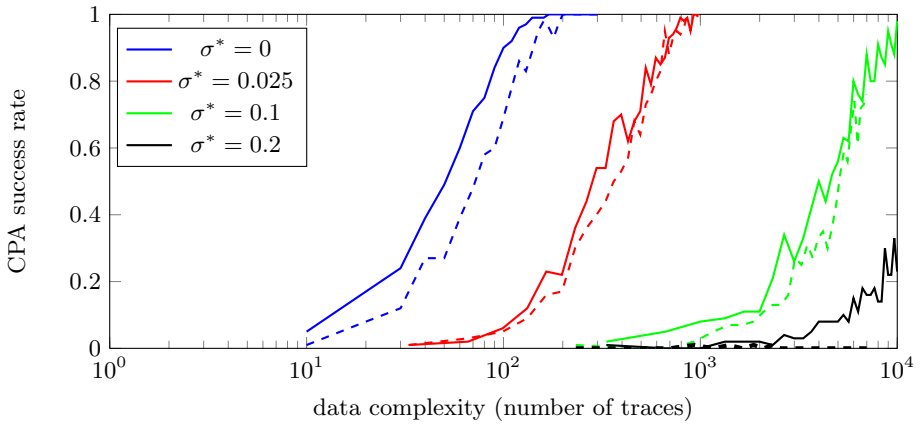

Fig. 6. Success rate of a CPA attack in different scenarios. The plain lines correspond to the attack on an unprotected implementation, the dashed lines correspond to the attack on a protected implementation with RDI removed by the HMM method. σ^* is the standard deviation of the simulated Gaussian noise added to the traces, $\sigma^* = 0$ corresponds to the real noise level. The delays are efficiently removed up to $\sigma^* = 0.1$

In view of the lack of security improvement of our RDI-protected implementation, we additionally investigated the impact of noise on the efficiency of the HMM-based random delay removal. For this purpose, we added to the (already noisy) traces a simulated random Gaussian noise with standard deviation σ^* and applied the HMM method to the resulting traces. The corresponding signal-to-noise ratios are given in Table 2. It turns out that it is still possible to efficiently remove the delays with $\sigma^* = 0.1$, which is one order of magnitude higher than the actual noise we observed in our measurements (i.e. $\sigma = 8 \times 10^{-3}$). In other words, up to very high noise levels, the adversary has more trouble estimating the correlation coefficient of a CPA than removing the delays. More precisely, Figure 6 shows the success rates of unprotected and protected implementations with $\sigma^* = 0.025$, $\sigma^* = 0.1$ and $\sigma^* = 0.2$. It confirms that removing the delays is still pretty accurate at $\sigma^* = 0.1$ (the correct length is found for 95% of the delays

in this case). However, when σ^* goes beyond 0.1, it becomes increasingly harder to correctly detect the delay lengths. And by $\sigma^* = 0.2$, the HMM method does not correctly identify the delay lengths anymore. In conclusion, the RDI countermeasure has an impact on security only when the noise level is high enough for a successful CPA attack against the corresponding unprotected implementation to require more than 10 000 traces. This unfortunately goes against the goal of using RDIs to emulate noise for small embedded (e.g. 8-bit) devices.

	$\sigma^* = 0$	$\sigma^* = 0.025$	$\sigma^* = 0.1$	$\sigma^* = 0.2$
SNR	0.2	0.014	1.34×10^{-3}	2.44×10^{-4}

Table 2. SNR of the unprotected implementation with additive simulated noise σ^*.

4.5 RDI improvements

The HMM method is very efficient against the RDI countermeasure presented in [6, 7], in part due to the very regular pattern of the delays. It is thus natural to think about possible methods to make the delays and the actual AES computations harder to tell apart. In the following of this section, we discuss some proposals that could be used to achieve this goal, and their limitations.

One straightforward idea is to use delays with no regular pattern, e.g. delays made of random instructions. However, implementing the RDI countermeasure in this case will still require (1) a specific deterministic header containing the instructions needed to determine the delay length, and (2) a loop ensuring that the delay stops after a given counter has reached 0. Using the hardware interrupts feature of the Atmel microcontrollers would not help in this respect, as dealing with these interrupts also gives rise to (4 or 5) very distinguishable cycles. In addition, even if the delays had a perfectly random pattern, the AES operations would still have to be processed, and could be identified with the Viterbi algorithm. For illustration, we ran experiments with hypothetical simulated traces, where the delays only consisted in random instructions instead of dummy loops. That is, these simulated traces do not contain any identifiable header or tail. We recall that this context is unrealistic as such headers and tails are needed to implement the countermeasure and generally provide useful information to the adversary. But even this minimum leakage could be efficiently exploited. Namely, while the Viterbi algorithm was not always able to accurately predict the instructions processed during the delays, it was still able to correctly identify the position of the AES instruction sequences. Hence, as illustrated in Appendix, Figure 8, such an idea is not sufficient to prevent successful attacks. Yet, one can notice that it reduces the noise robustness of the HMM cryptanalysis.

Another proposal is to use delays that look like the surrounding AES instructions. But this solution again faces the issue that the headers and tails are still distinguishable, leading to similarly efficient attacks. Besides, an extreme solution is to duplicate every AES computation n times, with one execution on

real data and the rest on dummy data. This way, the HMM method provides no information on where are the delays, as the delays are totally indistinguishable from the AES computations. Unfortunately, this countermeasure provides little security at high cost: instead of attacking one time sample, the adversary will simply have to integrate over n. Eventually, we conjecture that solutions to improve the time randomization of cryptographic implementations should combine RDIs with other ideas, e.g. shuffling [13] or clock jitter. We leave the analysis of these combined scenarios as an interesting scope for further research.

4.6 A non-profiled version of the HMM method

The previous sections assumed the adversary can perform a profiling phase on a test device. Although justified in a security evaluation context, this assumption may not always be verified in practice. In the following, we finally show that even in a non-profiled context, it is possible to perform efficient HMM-based attacks against an RDI-protected implementation, by exploiting clever "on-the-fly" characterization of the the target device, with a single leakage trace.

The key observation is that the encryption process cannot start with random instructions. Even if a delay is inserted at the beginning of the code, there are always some deterministic instructions in the delay header, where the length of the delay is determined. In our case, the AES encryption starts with 40 deterministic cycles before the first dummy loops take place. These deterministic cycles do not include every 26 instruction cycles for which we need a template, but only the first 12 of them (in Table 1). Moreover, these 40 cycles are not enough to estimate very accurate templates for all these 12 instruction cycles. Nevertheless, we can build 12 templates from these first 40 cycles and artificially increase their (noise) variance, as we know that they are not perfectly accurate. For the remaining 14 templates, we use a unique default template, built from the average of the cycles after the first 40. As a result, we have 26 emission probability functions e_i that we can plug into a first HMM: HMM_0. Using the Viterbi algorithm on the leakage trace $\mathbf{l}$ with the model HMM_0 gives the most probable sequence of states, according to our (admittedly bad estimations of the) emission probability functions. For the actual noise level we measured experimentally, between 50% and 70% of the observations are associated to the correct instruction cycle after this first iteration of the Viterbi algorithm. Next, from this predicted sequence of states, we can estimate new, more accurate templates (and emission probability functions) for the 26 instruction cycles. Plugging these new templates into a second HMM (i.e. HMM_1), we can process the same power trace $\mathbf{l}$ again, hence obtaining a better classification of the trace cyles. By iterating this procedure at most 10 times, we get a correct estimation of the templates and a perfect identification of the delay lengths. It is thus possible to perform the profiling phase "on-the-fly", by simply using iterations of the Viterbi algorithm on the same power trace, provided that we have at least some information to start with (e.g. the 12 not so accurate templates used in our experiment). As illustrated in Appendix, Figure 8, this non-profiled version of the attack is still robust to noise addition, up to quite low signal-to-noise ratios.

5 Conclusions

The investigations in this paper confirm the discussion in [22] that protections against side-channel attacks based on time randomizations are challenging to evaluate, as they may easily provide a false sense of security, whenever they are reversible with the appropriate signal processing / statistical / modeling tools. The case of RDIs is a good illustration of this concern: we show that their implementation in a low-cost microcontroller can be completely reversed and that making them effective is a non-trivial task (e.g. software-based improvements are unlikely to be sufficient). Hence, the study of other time randomization techniques such as shuffling [13], or the combination of RDIs with other countermeasures against side-channel attacks, are interesting research problems. Besides, RDIs are also considered as a solution to prevent fault attacks. In this setting, it is an interesting question to determine whether the HMM-based cryptanalysis could be applied "on-the-fly" during a fault attack, in order to help adversaries to insert faults in the sensitive computations of randomized implementations.

References

1. http://point-at-infinity.org/avraes/.
2. Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In Marc Joye and Jean-Jacques Quisquater, editors, *CHES*, volume 3156 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2004.
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 *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.
4. Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *CHES*, volume 2523 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2002.
5. Christophe Clavier, Jean-Sébastien Coron, and Nora Dabbous. Differential power analysis in the presence of hardware countermeasures. In Çetin Kaya Koç and Christof Paar, editors, *CHES*, volume 1965 of *Lecture Notes in Computer Science*, pages 252–263. Springer, 2000.
6. Jean-Sébastien Coron and Ilya Kizhvatov. An efficient method for random delay generation in embedded software. In Christophe Clavier and Kris Gaj, editors, *CHES*, volume 5747 of *Lecture Notes in Computer Science*, pages 156–170. Springer, 2009.
7. Jean-Sébastien Coron and Ilya Kizhvatov. Analysis and improvement of the random delay countermeasure of ches 2009. In Stefan Mangard and François-Xavier Standaert, editors, *CHES*, volume 6225 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2010.
8. Rémy Daudigny, Hervé Ledig, Frédéric Muller, and Frédéric Valette. Scare of the des. In John Ioannidis, Angelos D. Keromytis, and Moti Yung, editors, *ACNS*, volume 3531 of *Lecture Notes in Computer Science*, pages 393–406, 2005.
9. Julien Doget, Emmanuel Prouff, Matthieu Rivain, and François-Xavier Standaert. Univariate side channel attacks and leakage modeling. *J. Cryptographic Engineering*, 1(2):123–144, 2011.

10. Thomas Eisenbarth, Christof Paar, and Björn Weghenkel. Building a side channel based disassembler. *Transactions on Computational Science*, 10:78–99, 2010.
11. Sylvain Guilley, Karim Khalfallah, Victor Lomné, and Jean-Luc Danger. Formal framework for the evaluation of waveform resynchronization algorithms. In Claudio Agostino Ardagna and Jianying Zhou, editors, *WISTP*, volume 6633 of *Lecture Notes in Computer Science*, pages 100–115. Springer, 2011.
12. Sylvain Guilley, Laurent Sauvage, Julien Micolod, Denis Réal, and Frédéric Valette. Defeating any secret cryptography with scare attacks. In Michel Abdalla and Paulo S. L. M. Barreto, editors, *LATINCRYPT*, volume 6212 of *Lecture Notes in Computer Science*, pages 273–293. Springer, 2010.
13. 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.
14. James Irwin, Dan Page, and Nigel P. Smart. Instruction stream mutation for non-deterministic processors. In *ASAP*, pages 286–295. IEEE Computer Society, 2002.
15. Chris Karlof and David Wagner. Hidden markov model cryptoanalysis. In Walter et al. [30], pages 17–34.
16. Stefan Mangard. Hardware countermeasures against dpa ? a statistical analysis of their effectiveness. In Tatsuaki Okamoto, editor, *CT-RSA*, volume 2964 of *Lecture Notes in Computer Science*, pages 222–235. Springer, 2004.
17. Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.
18. Stefan Mangard, Elisabeth Oswald, and François-Xavier Standaert. One for all - all for one: Unifying standard dpa attacks. *IET Information Security*, 5 (2), 2011:100–110.
19. Sei Nagashima, Naofumi Homma, Yuichi Imai, Takafumi Aoki, and Akashi Satoh. Dpa using phase-based waveform matching against random-delay countermeasure. In *ISCAS*, pages 1807–1810. IEEE, 2007.
20. Denis Réal, Vivien Dubois, Anne-Marie Guilloux, Frédéric Valette, and M'hamed Drissi. Scare of an unknown hardware feistel implementation. In Gilles Grimaud and François-Xavier Standaert, editors, *CARDIS*, volume 5189 of *Lecture Notes in Computer Science*, pages 218–227. Springer, 2008.
21. Werner Schindler, Kerstin Lemke, and Christof Paar. A stochastic model for differential side channel cryptanalysis. In Josyula R. Rao and Berk Sunar, editors, *CHES*, volume 3659 of *Lecture Notes in Computer Science*, pages 30–46. Springer, 2005.
22. François-Xavier Standaert. Some hints on the evaluation metrics and tools for side-channel attacks. Non-Invasive Attacks Testing Workshop, Nara, Japan, September 2011, http://perso.uclouvain.be/fstandae/PUBLIS/107_slides.pdf.
23. François-Xavier Standaert and Cédric Archambeau. Using subspace-based template attacks to compare and combine power and electromagnetic information leakages. In Elisabeth Oswald and Pankaj Rohatgi, editors, *CHES*, volume 5154 of *Lecture Notes in Computer Science*, pages 411–425. Springer, 2008.
24. François-Xavier Standaert, Tal Malkin, and Moti Yung. A unified framework for the analysis of side-channel key recovery attacks. In Antoine Joux, editor, *EUROCRYPT*, volume 5479 of *Lecture Notes in Computer Science*, pages 443–461. Springer, 2009.

25. 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 *Lecture Notes in Computer Science*, pages 112–129. Springer, 2010.
26. Daehyun Strobel and Christof Paar. An efficient method for eliminating random delays in power traces of embedded software. In Howon Kim, editor, *ICISC*, volume xxxx of *Lecture Notes in Computer Science*, pages yyy–zzz. Springer, 2011.
27. Kris Tiri and Ingrid Verbauwhede. Securing encryption algorithms against dpa at the logic level: Next generation smart card technology. In Walter et al. [30], pages 125–136.
28. Michael Tunstall and Olivier Benoît. Efficient use of random delays in embedded software. In Damien Sauveron, Constantinos Markantonakis, Angelos Bilas, and Jean-Jacques Quisquater, editors, *WISTP*, volume 4462 of *Lecture Notes in Computer Science*, pages 27–38. Springer, 2007.
29. Jasper G. J. van Woudenberg, Marc F. Witteman, and Bram Bakker. Improving differential power analysis by elastic alignment. In Aggelos Kiayias, editor, *CT-RSA*, volume 6558 of *Lecture Notes in Computer Science*, pages 104–119. Springer, 2011.
30. Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors. *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, volume 2779 of *Lecture Notes in Computer Science*. Springer, 2003.

A Additional figures

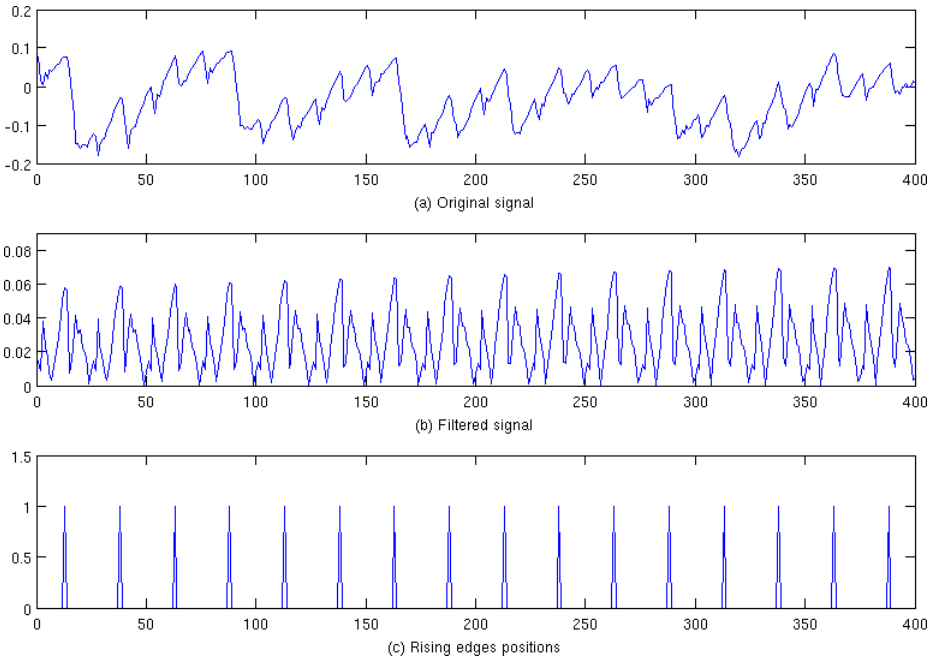

Fig. 7. CPU's clock rising edges detection on power traces.

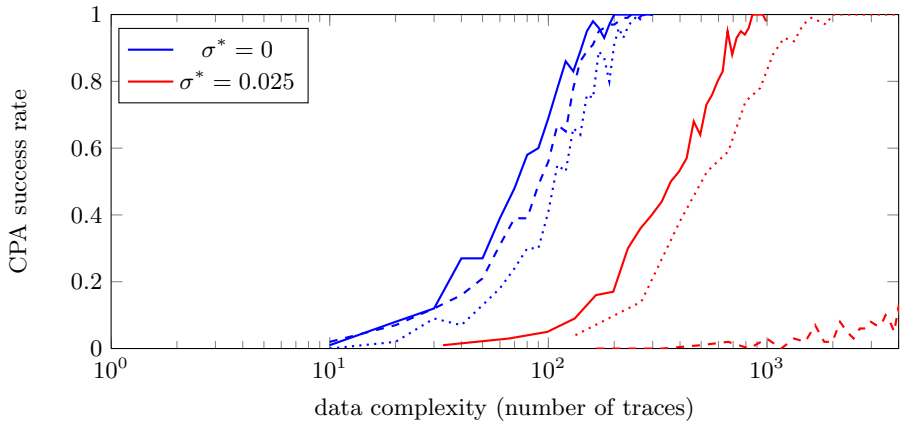

Fig. 8. Success rate of a CPA attack against traces processed using the HMM method in different scenarios. The plain lines correspond to the attack on a protected implementation with the CHES version of the RDI. The dashed lines correspond to the attack against the hypothetical (simulated) implementation of Section 4.5. The dotted lines correspond to the non-profiled version of the attack described in Section 4.6.