Shuffling Against Side-Channel Attacks: a Comprehensive Study with Cautionary Note

Nicolas Veyrat-Charvillon, Marcel Medwed,
Stéphanie Kerckhof, François-Xavier Standaert
Université catholique de Louvain, UCL Crypto Group,
B-1348 Louvain-la-Neuve, Belgium.

Abstract. Together with masking, shuffling is one of the most frequently considered solutions to improve the security of small embedded devices against side-channel attacks. In this paper, we provide a comprehensive study of this countermeasure, including improved implementations and a careful information theoretic and security analysis of its different variants. Our analyses lead to important conclusions as they moderate the strong security improvements claimed in previous works. They suggest that simplified versions of shuffling (e.g. using random start indexes) can be significantly weaker than their counterpart using full permutations. We further show with an experimental case study that such simplified versions can be as easy to attack as unprotected implementations. We finally exhibit the existence of "indirect leakages" in shuffled implementations that can be exploited due to the different leakage models of the different resources used in cryptographic implementations. This suggests the design of fully shuffled (and efficient) implementations, were both the execution order of the instructions and the physical resources used are randomized, as an interesting scope for further research.

1 Introduction

Already in the first Differential Power Analysis (DPA) paper, Kocher et al. mentioned time randomization as a possible solution to improve security against side-channel attacks [15]. Following, different countermeasures have been proposed to exploit this idea, e.g. relying on the addition of random delays [7, 31], shuffling the execution order of independent operations [13, 26], or more generally, trying to build a non-deterministic processor [4, 19]. As usual in side-channel attacks, the main question regarding these solutions is: "to what extent do they improve security and at which cost?". In this paper, we propose a comprehensive treatment of this question in the case of the shuffling countermeasure.

For this purpose, we start with the efficiency issue. In general, shuffling can be applied to any set of independent operations. The SubBytes layer of 16 S-boxes in the AES Rijndael is a typical example. Randomizing such operations can be done in different ways. Taking the extreme cases, either the S-boxes are executed according to a Random Permutation (RP) among 16! possible ones, or they are executed from a Random Start Index (RSI) among 16 possible ones, that is then incremented. This difference is nicely illustrated with previous works on shuffled implementations of the AES. In a first paper from 2006 [13], the authors use partial shuffling and first-order masking based on S-box pre-computation. Whereas

masking is applied to the whole cipher, shuffling is only applied to the first and last rounds. Furthermore, the RSI approach is pursued, for performance reasons. In a second work from Rivain et al., higher-order masking is implemented and shuffling is mainly based on the RP approach [26]. Yet, for the MixColumns operations, only the (first-order masked) columns are shuffled, accounting for 8! possible permutations. That is, thanks to the first-order masking, they have 8 positions that can be shuffled, vs. 4 if MixColumns was not masked, and 16 for the other AES transforms. Implementation details are not given in [26], but we assume that this choice is again motivated by performance reasons, with a MixColumn operation implemented with `xtime` tables [10]. Apart from those works, shuffling was also applied to hardware implementations with 8- or 32-bit datapaths, where RSI is usually preferred as it nearly comes for free [11, 20, 22].

Following this state-of-the art, our first contribution is to improve the performances of software implementations using the RP approach. In this respect, we start from the observation that in an unprotected block cipher implementation, one usually keeps as much data as possible in the processor registers, in order to minimize RAM access. By contrast, once random access to these registers is required (as in shuffled implementations), RAM usage is inevitable. This implies that any register access becomes a serial of load and store operations, resulting in major performance overheads. We mitigate these overheads by exploiting a different technique, which consists in manipulating the program flow. It allows us to operate on registers while at the same time randomizing the sequence of operations. In practice, we present two approaches: the first one changes the program flow "on-the-fly", while the other one re-writes the program memory prior to execution. The latter approach can be viewed as an adaptation of the self-modifying codes used in software engineering [2], also applied to counteract side-channel attacks in [1]. Our new solutions come with contrasted performance results. Namely, the on-the-fly proposal minimizes the overall cycle count, while the program memory manipulations allow very efficient online encryption at the cost of long (possibly offline) pre-computations. For illustration, we apply these proposals (and previously published ones) to the AES Furious implementation available from [23]. Besides, we also investigate the efficient generation of (small) random permutations in low-cost microcontrollers. That is, we take a well known optimal algorithm for permutation generation and modify it slightly, in order to improve its performances. As a result, we are able to generate close-to-uniform permutations, and obtain an efficient alternative to proposals such as [8].

Next, we investigate the security of shuffling against side-channel attacks. Here, we start from the observation that the existing literature usually evaluates the impact of shuffling based on a so-called "integrated DPA" (aka windowing attack), introduced in [7] and applied, e.g. in [26, 30]. Intuitively, if the manipulation of a sensitive variable is spread over t time samples, its correlation with the actual leakages will be reduced by a factor $\sqrt{t}$ using such an attack, instead of t without integration. Integrating is a convenient tool for evaluation as it can be directly used to estimate the data complexity of a DPA using Mangard's formulas [17]. Yet, a possible limitation of this technique is that it treats the RSI

and RP cases in the same way. Hence, a natural question is to determine whether these two approaches are indeed equivalent in general, or if advanced evaluation tools can be used to put forward additional weaknesses for RSI implementations. Our results regarding this question are summarized as follows.

First, we specialize the information theoretic and security analysis from [28] to the context of shuffled implementations. It allows us to confirm that integrated DPA is indeed suboptimal compared to a Bayesian exploitation of the leakages. While our worst-case evaluations rely on profiled attacks [6, 27], we believe they are important to moderate claims of strong security improvements provided by shuffling (e.g. the data complexity increases by a factor 360 in [4]). In particular, these results complement the previous work of Asiacrypt 2010 [29], in which such an information theoretic and security analysis was performed for masking. As a result and for the first time, we obtain lower bounds for the data complexity of standard side-channel attacks against shuffled implementations.

Second, we notice that security evaluations for masking always combine the leakage corresponding to the masked data and its masks, e.g. [21, 24]. Quite surprisingly, and to the best of our knowledge, the impact of such a scenario has not been investigated in the case of shuffling. Therefore, we include the possibility of a leakage on the permutation (or start index) manipulated when shuffling. We show that as soon as some information is leaked about them, attacks against RSI- and RP-based implementations become significantly different, the RSI case being much easier to attack, for computational reasons.

Finally, we observe that direct leakages about the start index or permutations naturally arise in practice and can be exploited. More surprisingly, we also show the existence of "indirect leakages", coming from the different power consumption models of the hardware resources manipulating the key bytes. For example, since the 16 registers used in our shuffled Furious implementations have (slightly) different models, marginalizing the distribution of the observed leakage over the 16 AES key bytes provides information about which S-box is computed.

Summarizing, we observe that *all* previous works on shuffling reduced the size of the permutation set for some of the operations in the protected algorithm. Hence, our results bring the important cautionary note that time complexity is critical in the security evaluation of this countermeasure, as permutations with a small size can be enumerated which leads to exploitable weaknesses. In this respect, an implementation protected with RSI-based shuffling can sometimes be as weak as an unprotected one. As for the RP-based solution, we recall that it can be used as a noise amplifier for leaking devices, but never as a noise generator.

2 Efficient implementations

This section explores the software design space for shuffling the AES on an Atmel ATMega644P microcontroller [3]. We first describe an efficient way to obtain close-to-uniform permutations in this device. Next, we show how to obtain an AES implementation for which every transform can be shuffled according to such permutations, including MixColumns (and the key scheduling algorithm if

needed). Afterwards, we describe different implementations: a basic one relying on a previously proposed "double indexing" method, and two optimized ones relying on randomized execution path and program memory. We finally provide precise performance evaluations and a comparison with previous works.

Permutation generation. The first building block of a shuffled implementation is a permutation generator. From a sequence $S := \{1, \ldots, n\}$, a uniform permutation can be produced in linear time [14]. The original algorithm iterates over every element S_i (with i from 0 to $n-2$), and swaps it with a random element from the remaining tail, i.e. $\{S_i, \ldots, S_{n-1}\}$. However, sampling from $\{i, \ldots, n-1\}$ needs either a modulo operation and a random number greater than n to start from, or an approach with probabilistic run-time. We avoided this performance drawbacks by sampling from $\{0, \ldots, n-1\}$. Permuting a sequence of 16 entries following this algorithm takes 362 cycles on our device, using 8 bytes of randomness. It still allows to generate all permutations, but with a slight bias that decreases with the size of the permuted set. To estimate the impact of this bias for different sizes of the permutation set N, we systematically sampled 10^8 permutations generated with this method, and built histograms with $N!$ bins. We then estimated the Euclidean distance between these biased histograms and a uniform distribution. In addition, we compared this situation with the one obtained with a quite minimum side-channel leakage. Namely, we assumed that the Hamming weight of the first entry of a (uniformly generated) permutation is known to be the least informative one (i.e. with half of the bits set to one). As can be observed in Table 2, the bias due to this small side-channel information is already significantly larger than the one due to the permutation generation algorithm. Furthermore, actual leakages in Sections 3 and 4 affect all the permutation entries, which further reduces the bias of the permutation generation algorithm compared to the one caused by physical information. Eventually, we will show in the next sections that exploiting these biases in a side-channel attack where we shuffle among 16! possible permutations is computationally hard. Therefore, we conclude that our performance optimized algorithm should not lead to a significant security reduction of the shuffling countermeasure.

N	3	4	5	6	7	8	9
Perm. generation	0.04535	0.03522	0.02034	0.00993	0.00430	0.00170	0.00063
Small SCA Leak.	0.28868	0.20412	0.07454	0.03726	0.01627	0.00643	0.00234

Table 1. Bias of the optimized permutation algorithm vs. bias of a small SCA leakage.

Obtaining independent operations. Applying shuffling to an implementation requires finding sets of independent operations. In the AES case, sets of 16 independent operations naturally arise from the AddRoundKey and SubBytes transforms. By contrast, the situation is a little bit trickier for ShiftRows and MixColumns. For example, implementing ShiftRows requires one extra byte of storage in an unprotected implementation, and two in the case of RSI-based shuffling (i.e. when the permutation is "monotonous", which restricts the number of

permutations to 16). But if 16 independent operations are desired, 16 bytes of temporary storage are required. As for MixColumns, four additional registers are sufficient if the state is processed column-wise, but this would then account for 4! permutations. Hence, having 16 independent operations again requires 16 bytes of temporary storage. Since our device has only 32 registers, some of which being already occupied, RAM usage becomes inevitable for shuffling. Besides, the key schedule has only four independent operations by default. This is because within one key schedule round, there are only four S-box executions. Thus, the smallest number of indistinguishable operations is four. Yet, applications requiring on-the-fly key expansion also need an appropriate SPA protection to prevent attacks such as [16]. In these cases, we interleaved the real key schedule with three dummy key schedules, in order to obtain 16 shuffleable operations.

Basic implementation with double indexing. Direct shuffling requires an indirect indexing of the operands. That is, a counter is used to index a permutation vector, and the result is used to index the operand vector. Thus, instead of operating on registers directly, two RAM accesses are required for each (read or write) access to operands. This naturally leads to quite large cycle counts, as in AVR devices, load and store operations take two cycles (compared to one cycle for arithmetic and logic operations). Implementing a fully shuffled AES this way results in an execution time of 30 202 cycles, excluding the key schedule. In the following we propose two different strategies in order to improve on these figures. In both cases, instructions are shuffled rather than data location, in order to allow register usage. Precisely, we are still limited by the number of available registers when performing certain transforms. But contrary to the double indexing proposal we do not always access RAM when operating on intermediate data. The first solution changes the execution path on-the-fly while the second actually rewrites the program memory (i.e. assuming that this re-writing is pre-computed, this solution can be seen as a simplified one-time program [12]).

Optimized implementation with randomized execution path. For this implementation, the assembly code of every (compound of) round transform(s) is split into 16 independent blocks of instructions. Each of the 16 blocks is augmented with a label. This allows us to identify its address in ROM. Furthermore, every transform is associated with an array of 17 16-bit words, where the first 16 words hold the addresses of the 16 blocks, and the 17th holds the address of the return instruction. The array content is initialized with the addresses of the labels at compile time. Finally, we append a flow-control macro to each of the 16 blocks. This macro performs three things: fetch an address from the array, advance the pointer to the next array entry, and jump to the fetched address.

During the execution of the cipher, we first re-order the first 16 addresses in the array, according to a previously generated permutation. Then, when we enter a transform, we set a pointer to the beginning of the array and execute the flow-control macro. This causes the execution of the first block and sets a pointer to the address of the next block. The flow-control macro is executed 16 times, until it finally looks up the address of the return instruction. In practice, we defined

several sets of transforms and therefore need an address array for each of them. The first one is the compound of AddRoundKey, SubBytes, and ShiftRows. This transform reads the state from RAM and stores the result into the register file. The next one performs MixColumns and stores the result back to RAM. Afterwards, we perform one iteration of the key schedule. Similar to ShiftRows and MixColumns, this implies additional memory requirements, because of the RotWord operation. We finally need a standalone AddRoundKey layer for the last round. As each of the address arrays need 17x2 bytes, we have an additional RAM use of 170 bytes (for technical reasons, the key schedule uses two 17x2-byte address arrays). Permuting each of these arrays takes 205 cycles, implying an overhead of 1225 cycles. Eventually, for every set of transforms, we need to load an address and jump to this address 17 times, each of which takes 6 cycles. Together with the preamble to set up the array pointer, it leads to an additional overhead of 108 cycles for each of these compounds of transforms.

Optimized implementation with randomized program memory. For this implementation, we used the self-programming capabilities of the ATMega644p microcontrollers. As the shuffling applies to independent operations, and as for each operation, the state bytes are always stored in the same registers or RAM locations, the execution order of the operations can be permuted by modifying the data corresponding to these locations in program memory. In our target controller, the program memory has to be modified one page (i.e. 256 bytes) at a time. Hence, the shuffling can be prepared in five steps. First, the page is transferred from program memory to the RAM. Afterwards, the bytes of code corresponding to state-byte locations are modified according to the permutation vector. Then, the previous version of the page is erased from program memory, and the new page is loaded into a page buffer. Finally, this page buffer is written in program memory. This process is executed before each AES execution. The main advantage of this solution is that after pre-processing of a shuffled program memory, the execution time of the AES is nearly the same as for the unprotected implementation. Minor differences come from the fact that we need to have independent operations, which implies to use RAM for the storage of some intermediate results. Its main drawback is the long pre-computation time, which accounts for approximately 18 milliseconds independently of the clock frequency. This comes from the time-consuming instructions used to erase program memory and write page buffer in memory (4.5 millisecond per page writen or erased [9]), and the low granularity of these instructions (i.e. working at the page level) in the Atmel controllers. More flexible devices (e.g. devices with ARM architectures) would allow to improve this limitation. Note also that our target Atmel's EEPROM allows only for 10 000 re-write cycles, which could possibly lead to DoS attacks. If this is an issue, and depending on the actual available ROM, different areas can be used randomly and increase the number of possible encryptions by some factor. Again, alternative devices could be considered to relax this limitation. For example, the ARM LPC214x series allows already for 100 000 cycles. Note finally that, as this implementation mainly makes sense if pre-processing is allowed, it is naturally executed with a pre-computed key scheduling.

Implementation results. The performance results of our implementations are compared with previous works in Table 2. Namely, we use the AES Furious as reference. As for protected implementations, we considered the basic one based on double indexing and the ones of Herbst et al. and Rivain et al. However, as mentioned above, they do not allow direct comparison. Herbst et al. only protect the outer rounds (one and ten) with RSI-based shuffling, but implement masking for all the rounds and the key schedule. Rivain et al. implement higher-order masking and use a "simplified" shuffling for the MixColumns operation (they also work on a different 8051-based architecture). The implementation for which we give cycle numbers is not masked except for MixColumns. By contrast, our implementations use log-table based polynomial multiplication, and are able to shuffle all bytes during MixColumns. Not surprisingly, our implementation based on double indexing is the slowest. Its performance is comparable to the one of Rivain et al. Manipulating the program counter allows us to get a performance improvement of almost a factor five and, excluding the key scheduling, leads to encryption time only twice as slow as Rijndael Furious. As previously mentioned, the larger overheads when executing the key scheduling come from the need to execute additional dummy schedulings, in order to keep a permutation among 16! for this part of the implementation. Finally, the randomized program memory allows the fastest online encryption (i.e. excluding program re-writing).

Table 2. Implementation result comparison

Implementation	Clock cycles	RAM [byte]
Furious [23]	2 739	176
Furious with KS [23]	3 546	176
Herbst et al. [13]	11 845	-
Rivain et al. [26]	29 400	-
Dbl. ind.	30 202	240
Dbl. ind. with KS	46 395	132
Rand. exec. path	6 934	394
Rand. exec. path with KS	14 834	302
Rand. prog. mem.	3299 (+$\simeq$18 msec)	480

3 Evaluation framework

We now move to the security analysis of the shuffling countermeasures and its variants. For this purpose, we rely on the evaluation framework from [28] and adapt it to capture the specificities of shuffled implementations. In order to have a fair understanding of the strengths and weaknesses of the countermeasure, we pay a particular attention to worst-case (profiled) attacks. But for completeness, we also compare them with the integrated DPA used in previous works.

Notations. Variables are denoted with capital letters, sampled values with lowercase letters and functions with sans serif fonts. We consider the standard DPA attacks described in [18] and illustrate our notations with the case of the

AES Rijndael. In this context, the adversary tries to recover a 16-byte master key $\mathbf{k} = \{k_0, k_1, \ldots, k_{15}\}$, from a leakage corresponding to the first key addition and S-box layers. In attacks against unprotected implementations, each S-box is executed at a well defined time instant, giving rise to key leakages defined as:

$$\begin{array}{lll} L_0 & \leftsquigarrow & \mathsf{Sbox}(k_0 \oplus X_0), \\ L_2 & \leftsquigarrow & \mathsf{Sbox}(k_2 \oplus X_2) \end{array} \qquad \begin{array}{lll} L_1 & \leftsquigarrow & \mathsf{Sbox}(k_1 \oplus X_1), \\ & \ldots & \end{array}$$

That is, we have 16 leakage points (or cycles) L_c (where c is the cycle index) and 16 subkeys k_s. If we denote the part of the master key that is manipulated at time c with a variable S_c, we straightforwardly have $S_c = c$ in this unprotected case. Note that the variable nature of the leakages comes both from possible noise in the measurements and the variable (known) inputs X_i. By contrast, in the case of a shuffled implementation, the execution order of the S-box computations is randomized according to a permutation P, leading to key leakages of the form:

$$\begin{array}{lll} L_0 & \leftsquigarrow & \mathsf{Sbox}(k_{\mathsf{P}(0)} \oplus X_{\mathsf{P}(0)}), \\ L_2 & \leftsquigarrow & \mathsf{Sbox}(k_{\mathsf{P}(2)} \oplus X_{\mathsf{P}(2)}) \end{array} \qquad \begin{array}{lll} L_1 & \leftsquigarrow & \mathsf{Sbox}(k_{\mathsf{P}(1)} \oplus X_{\mathsf{P}(1)}), \\ & \ldots & \end{array}$$

That is, we have $S_c = \mathsf{P}(c)$ with P the secret permutation that is re-generated for every new input block, e.g. with the algorithm in Section 2. In this protected case, not only leakage about the S-box execution may be obtained, but also leakage on the permutation used in the shuffled implementation. In theory, an attack could exploit sixteen "direct" permutation leakages denoted as $L'_c \leftsquigarrow S_c$. Such notations allow us to reflect both the RSI- and RP-based shuffling methods. In the first case, we have $\mathsf{P}(c) = c + \tau \pmod{16}$, with $\tau \xleftarrow{R} [0:15]$. In the second case, P is directly picked up among the set of all 16! permutations, i.e. $\mathsf{P} \xleftarrow{R} \mathcal{P}_{16}$.

Information theoretic analysis. As a first step in our evaluation, we perform an information theoretic analysis that is aimed to capture the worst-case security of an implementation. In general, and for a fixed key byte K_s, we assume that the adversary can observe a leakage vector $\mathbf{L} = \{L_0, L_1, \ldots, L_{15}\}$. The goal of this evaluation is to obtain an accurate estimation of the mutual information[1]:

$$\mathrm{MI}(K_s; \mathbf{L}, X) = \mathrm{H}[K_s] - \sum_k \Pr[K_s = k] \sum_x \Pr[X = x] \cdot \int_l \Pr[\mathbf{L} = \mathbf{l} | K_s = k, X = x] \cdot \log_2 \Pr[K_s = k | \mathbf{L} = \mathbf{l}, X = x] \; dl.$$

In this equation, the term $\Pr[K_s = k | \mathbf{L} = \mathbf{l}, X = x]$ is directly obtained from $\Pr[\mathbf{L} = \mathbf{l} | K_s = k, X = x]$ using Bayes' theorem. Hence, it is this last conditional leakage probability that is most critical to evaluate. For convenience, we will ignore the variable X in the rest of the paper, as it is assumed to be known for all computations. Next, we will consider two main evaluation scenarios.

[1] As discussed in [25], this mutual information can only be perfectly estimated when the evaluator knows the exact leakage model of his target device. This only happens in simulated analyses (e.g. as will be performed in the next section). Whenever a practical evaluation is carried out, it is formally a "perceived information" that is evaluated, with the goal to be as close as possible to the mutual information.

1. No permutation leakage, i.e. the adversary gets 16 leakage cycles, and each of them could correspond to the target subkey with probability 1/16. That is:

$$\Pr[\mathbf{L} = \mathbf{l} | K_s = k] = \sum_c \frac{1}{16} \Pr[L_c = l_c | K_s = k].$$

We will refer to this attack as case $(1.a)$. Besides, and as mentioned in introduction, a usual trick to attack shuffled implementations is to integrate over the leakage cycles. In this case, the adversary defines a variable $\overline{L} = \sum_c L_c$, and performs the attack against this variable. It boils down to consider 15 cycles out of 16 as "algorithmic noise". We will refer to this attack as case $(1.b)$.

2. Leakage on the permutation. In the same way as all the shares are assumed to leak in a masked implementation, it is natural to assume that the manipulation of a permutation may leak in a shuffled implementation. In practice, such leakages usually appear each time the permutation is manipulated in the microcode, e.g. when fetching the S_c'th part of the key, or when jumping to the S_c'th piece of code computing an S-box. We now show how to perform an information theoretic evaluation in these cases. As previously, the impact of different implementations of the countermeasure affects the term $\Pr[\mathbf{L} = \mathbf{l} | K_s = k]$. For this purpose, we start with the following general formulation:

$$\Pr[\mathbf{L} = \mathbf{l} | K_s = k] = \sum_c \frac{\mathsf{f}(c, s, \mathbf{l}')}{\sum_{c'} \mathsf{f}(c', s, \mathbf{l}')} \Pr[L_c = l_c | K_s = k],$$

with $\mathbf{l}'$ the vector of 16 leakages on the previously defined variable S_c (indicating the part of the master key used at time c). The function f essentially indicates how the knowledge available about this variable can be exploited by the adversary, as witnessed by the five examples that we now describe.

$2.a$. *Unprotected implementation.* In this case, we have $\mathsf{f}(c, s, \mathbf{l}') = 1$ if $c = s$ and 0 otherwise (i.e. the adversary knows exactly where each key byte is manipulated).

$2.b$. *Direct template attack.* In this case, we just add the permutation leakage in the conditional probabilities, yet without making any difference between the RSI and RP cases, by computing $\mathsf{f}(c, s, \mathbf{l}') = \Pr[L'_c = l'_c | S_c = s]$. Note that the case with no permutation leakage corresponds to $\mathsf{f}(c, s, \mathbf{l}') = 1/16$.

$2.c$. *Taking advantage of* RSI. Here, the the adversary exploits the fact that only 16 permutations are possible (out of the 16! ones), which can be enumerated. Hence, he can compute: $\mathsf{f}(c, s, \mathbf{l}') = \prod_{i=0}^{15} \Pr[L'_i = l'_i | S_i = (s - c + i) \bmod 16]$.

Contrary to the RSI case, using a RP implies that the permutation is picked up randomly among the 16! $\simeq 2^{44}$, which is significantly harder to enumerate. Hence, our following experiments will additionally consider two heuristic solutions that can be used to mitigate this issue and attack more efficiently.

$2.d$. *Restricted enumeration against* RP. In this case, the function f is identical to the exhaustive one, i.e. $\mathsf{f}(c, s, \mathbf{l}') = \sum_{\mathsf{p}} \prod_{i=0}^{15} \Pr[L'_i = l'_i | S_i = \mathsf{p}(i)]$, but the

sum only goes over an enumerable subset of most probable p's. A *beam search* is used for this purpose [32]. This is a breadth-first search that limits the number of nodes (i.e. permutations in the sum) by pruning the least probable ones, which is done by weighting permutations p's with $\prod_{i=0}^{15} \Pr[L'_i = l'_i | S_i = \mathsf{p}(i)]$.

2.*e. Excluding heuristic.* One alternative option to simplify the enumeration is to consider that whenever $S_c = s$, we have that $c \neq c'$ implies $S_{c'} \neq S_c$. This can be reflected with: $\mathsf{f}(c, s, \mathbf{l}') = \Pr[L'_c = l'_c | S_c = s] \cdot \prod_{c' \neq c} (1 - \Pr[L'_{c'} = l'_c | S_{c'} = s])$, which, up to normalization, is equivalent to:

$$\mathsf{f}(c, s, \mathbf{l}') = \frac{\Pr[L'_c = l'_c | S_c = s]}{1 - \Pr[L'_c = l'_c | S_c = s]}.$$

Overall, an intuition on the security of different implementations is obtained by quantifying the number of possible execution orders considered by the adversary (which may be more than the actual number of permutations, if attacks do not fully exploit their structure). In the unprotected case, only one order can occur. For the direct template attack, the adversary does not combine the different S_c informations and we implicitly have 16^{16} possible execution orders. In the RSI case, we exploit the fact that only 16 permutations are possible. The attack enumerating all possible permutations lists all 16! hypotheses. Finally, the excluding heuristic implicitly allows 16×15^{15} ones. This situation can be seen as an error correcting problem where 16 noisy values are transmitted, that can be integers from 0 to 15. The security of the countermeasure relies on a large probability of decoding error. In the RSI case, we only have 16 possible codewords, which gives us a very resilient code, lowering the probability of errors and thereby the strength of the countermeasure. For a RP, we have 16! codewords over a space of 16^{16} possible transmissions, hence increasing the probability of decoding errors.

As far as performing these attacks/evaluations in practice is concerned, case (a) is a classical template attack for which the computational complexity is usually neglected. Carrying out attacks/evaluations where L' is exploited naturally requires to build additional templates. Yet, the computational complexity of cases (b), (c) and (e) can also be neglected, as they only imply a few additional arithmetic operations. In fact, only case (d) may require intensive computations, if all permutations with non-negligible likelihood (with respect to L') are taken into account by the beam search. As will be shown in the next section, increasing the noise gradually implies that all permutation candidates have more similar likelihoods. Hence, this last attack is only applicable for low noise levels.

Security analysis. The second step of our evaluation is to perform a security analysis. It allows measuring the extent to which the different strategies listed have a strong impact on the data complexity of successful side-channel attacks. For this purpose, we apply template attacks with the key selected as:

$$\tilde{k} = \operatorname*{argmax}_{k^*} \prod_{j=1}^{q} \Pr[\mathbf{L}^j = \mathbf{l}^j, \mathbf{L}'^j = \mathbf{l}'^j | K_s = k^*],$$

and we compute their success rate, in function of the data complexity q.

4 Simulated experiments

In order to gauge the impact of the proposed formulations and attacks, we first lead various experiments against simulated AES implementations. For this purpose, we re-use the notations introduced in the previous section and assume that the adversary is provided with a key leakage vector $\mathbf{L}$ with elements $L_c = \mathsf{HW}(\mathsf{Sbox}(k_{\mathsf{P}(c)} \oplus X_{\mathsf{P}(c)})) + \mathcal{N}(0, \sigma^2)$, and possibly a permutation leakage vector $\mathbf{L}'$ with elements of the form $L'_c = \mathsf{HW}(S_c) + \mathcal{N}(0, \sigma^2)$. In both cases, the second term is a Gaussian distributed random noise, with variance σ^2 that we will use as a parameter of our evaluations. Using these notations, there are various contexts that could be investigated. As illustrated in Table 3, we classify them among two axes: the target device and the adversary's means.

		Target devices		
		Unp.	RSI shuf.	RP shuf.
adversary's means	**L**	UNP-TA $(2.a)$	INT-TA $(1.b)$	
			UNI-TA $(1.a)$	
	L,**L**′		DPLEAK-TA $(2.b)$	
	L,**L**′ + comp.		RSIENUM-TA $(2.c)$	RESENUM-TA $(2.d)$
				EXCLUDING-TA $(2.e)$

Table 3. Classification of the attacks

As far as the target device is concerned, we considered the case of an unprotected implementation for reference, an RSI-based shuffled implementation and a RP-based shuffled implementation. As far as the adversary's means are concerned, we first analyzed attacks where only the key leakage vector $\mathbf{L}$ is available. Next we evaluated attacks where the permutation leakage vector $\mathbf{L}'$ is additionally provided. Finally, we quantified the efficiency gains obtained when exploiting computational power, in order to enumerate (i.e. sum over) the possible permutations. Overall, this gives rise to seven attacks:

1. Template attack against the unprotected implementation (UNP-TA), i.e. the straightforward case where S-boxes are executed in deterministic order.
2. Template attack against integrated leakages (INT-TA), i.e. the attack against shuffled implementations previously used, e.g. in [7, 26, 30].

In these two first cases, template attacks and correlation DPA are essentially equivalent given that they exploit the same leakage model [18]. For coherence, we will keep on using template attacks everywhere. But as the experiments in Section 5 target a microcontroller with strong Hamming weight leakage dependencies, simpler (non-profiled) attacks would naturally apply as well. By contrast, the following attacks explicitly take advantage of a Bayesian description.

3. Template attack with uniform S_c (UNI-TA). In this case, the adversary follows the Bayesian strategy but does not exploit any information on the permutation (i.e. he assumes a uniform prior on the leakage cycles). Hence, the attacks still have identical efficiencies in the RSI and RP cases.

4. Template attack with direct permutation leakage (DPLEAK-TA). It corresponds to the attack $(2.b)$ described in the previous section. Here, the leakage vector $\mathbf{L}'$ is simply added in the adversary's conditional probabilities. But again, it does not distinguish between the RSI and RP cases.
5. Template attack with permutation leakage enumerating the RSI permutations (RSIENUM-TA). It corresponds to the attack $(2.c)$ in the previous section, where the adversary takes advantage of the 16 permutations that a RSI-based shuffling tolerates to combine its permutation leakages.
6. Template attacks with restricted enumeration (RESENUM-TA). It corresponds to the attack $(2.d)$ described in the previous section. A beam search [32] is performed to enumerate the most likely permutations.
7. Template attacks with excluding heuristic (EXCLUDING-TA). It corresponds to the attack $(2.e)$ in the previous section, where the likelihood of the permutations is weighted by simply excluding duplicates.

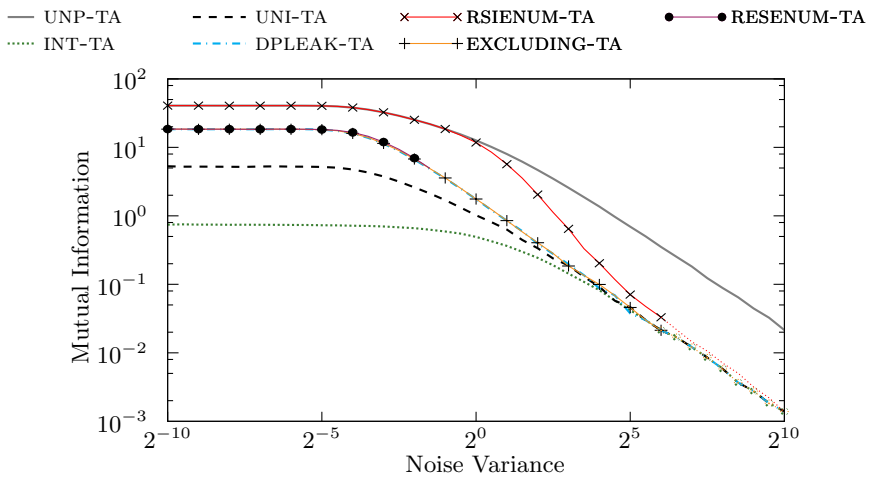

Fig. 1. Mutual information versus noise variance

The result of a simulated information theoretic analysis for these different attacks is given in Figure 1, in function of the noise variance. Several observations can be highlighted. First, and as usual in such worst-case evaluations, the asymptotic trend only appears for large noise levels. In this respect, the main conclusion is that (unlike masking [29]), the slope of the MI curves is the same for both the unprotected and all the shuffled implementations. Intuitively, it suggests that shuffling can (at best) be used to amplify the noise existing in side-channel measurements (i.e. imply a shift of the IT curves). Besides, one can observe that for lower noise levels, significant differences arise between the different scenarios of Table 3. For example, it is interesting to note that even without exploiting permutation leakage, the integrated attack is less efficient than the template attack with uniform prior. It confirms that this integrated attack is suboptimal in a profiled case, and is not suited to evaluate the worst-case

security of an implementation in low-noise scenarios. Quite naturally, the distance between integrated and stronger attacks increases as permutation leakage becomes available. In this setting, the amount of information extracted is quite dependent on countermeasure implemented. If the RSI approach is chosen (and this information is exploited computationally), the implementation turns out to be as weak as an unprotected one until noise levels beyond $\sigma^2 = 2^0$. By contrast, in the RP case, the noise amplification happens earlier. In this respect, it is worth to notice the limited difference between the DPLEAK-, EXCLUDING-, and RESENUM-TAs for RP-shuffled implementations, the latter ones only bringing a small advantage. We also observe that as expected, the RESENUM-TA could only be launched until noise levels of approximately $\sigma^2 = 2^{-2}$: beyond this threshold, the large amount of permutations to enumerate with the beam search turned out to be hardly tractable. This last fact confirms the expectation in Section 2 that the small bias resulting from our efficient permutation generation algorithm should not lead to significantly improved side-channel attacks.

Note that the insecurity of RSI-based shuffling (and, to a lower extent, RP-based shuffling) for low noise levels has to be interpreted with care. What our analysis shows is *not* that the start index or permutation is trivially revealed with a template-based SPA (as the number of permutation candidates in the beam search already explodes when $\sigma^2 = 2^{-2}$). It is really the fact that the 16 leakage samples of the permutation can be exploited jointly that make these countermeasures weak. In other words, what these results show is the importance of computational power in the evaluation of shuffling: summing over 16 cases is easy, summing of 16! ones is harder, as highlighted by the different curves of the RSIENUM-TA and EXCLUDING-/RESENUM-TA information theoretic evaluations.

As a complement of information theoretic analyzes, we performed a security analysis, and computed the success rates of our different attacks, in function of the number of plaintexts measured by the adversary. This allows translating the IT curves of Figure 1 into data complexities. For illustration, we selected three different noise variances, corresponding to low (i.e. $\sigma^2 = 2^{-3}$), middle (i.e. $\sigma^2 = 2^0$) an large (i.e. $\sigma^2 = 2^3$) noise levels (where large refers to the fact that the IT curves are merging at this stage). The results of these simulated experiments are given in Figure 2 and confirm the previous observations. We again observe the weakness of the RSI-based shuffling in the low noise level case, and the lower efficiency of the integrated attack. The success rate curves also exhibit the slight advantage of the heuristic enumeration when exploiting the leakage of a RP for the smallest noise level, as well as the better behavior of the (computationally cheap) excluding heuristic when the noise increases (again, the RESENUM-TA evaluation could only be performed in the low noise case, i.e. upper figure).

5 Practical experiments

The previous simulated attacks naturally raise the question whether our attacks similarly apply to real world implementations. In order to validate our conclusions, we also performed these attacks against shuffled implementations of the AES, based on the randomized execution path technique of Section 2.

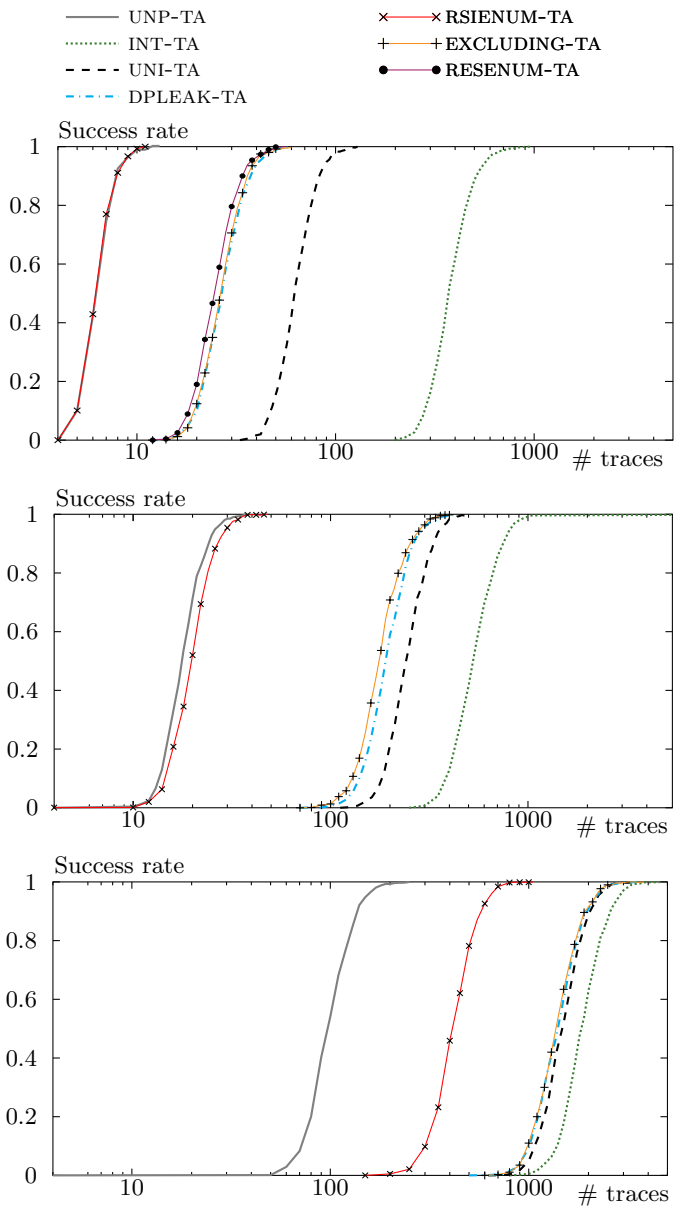

Fig. 2. Success rates of simulated attacks, $\sigma^2 = 2^{-3}$ (top), 2^0 (middle), 2^{+3} (bottom).

Our target device is an 8-bit Atmel microcontroller, and our measurement setup was monitoring the voltage variations of this target device over a small resistor inserted in our supply circuit, with a digital oscilloscope. Based on this setup, we profiled our implementation and built the probability distributions of the vectors $\mathbf{L}$ and $\mathbf{L}'$. That is, we first estimated 16 templates corresponding to the leakages of the permutation indexes c, i.e. $\Pr[L'_c|S_c = s]$. Next, we constructed 16×16 templates for the key leakages at the output of the S-box, i.e. $\Pr[L_c|K_s = k]$, for each value of c and s. The reason for having the 16×16 sets of key leakage templates is that these leakages behave differently when, at a given point in time, different subkeys are used. That is, we have $\Pr[L_c|K_{s_1}] \neq \Pr[L_c|K_{s_2}]$ if $s_1 \neq s_2$, and $\Pr[L_{c_1}|K_s] \neq \Pr[L_{c_2}|K_s]$ if $c_1 \neq c_2$. This fact is due to the slightly different power consumptions of different registers and memory accesses of the Furious implementation in our target device.

In order to limit the profiling efforts, our templates were kept univariate and constructed with the stochastic approach from [27], using the Hamming weight of the S-box outputs as base vectors. Interestingly, the fact that different key bytes give rise to different templates leads to indirect leakages on the permutation. That is, we have $\Pr[L_c|K_{s_1}] \neq \Pr[L_c|K_{s_2}]$ for a fixed cycle c. By summing over the 256 key candidates, we can then obtain marginal probabilities $\Pr[L_c = l_c|K_s]$ for all key byte indexes s. This directly leads to useful information of the type:

$$\Pr[S_c = s|L_c = l_c] = \frac{\Pr[L_c = l_c|K_s]}{\sum_{s'} \Pr[L_c = l_c|K_{s'}]}.$$

Furthermore, this information is directly reflected in all the Bayesian attacks, without any modification of the descriptions in Section 3 (including UNI-TA for which direct permutation leakages are ignored). That is, just the fact that we built 16×16 templates for different s and c values allows to exploit it.

The success rates of our experimental attacks are illustrated in Figure 3, where the noise level corresponds to $\sigma^2 = 3.25$. We observe that in this real case study, the RSI-based shuffled implementation remains as easy to attack as an unprotected one, in our worst-case evaluation setting. Besides, we note that the indirect leakage is quite useful for the template attack with uniform prior. One important consequence of this indirect leakage is that the UNI-TA could also apply to our countermeasure with randomized program memory, even if the precomputation was performed in a perfectly secure (i.e. leakage-free) environment. Interestingly, we also remark that the integrated attack is less efficient than in our simulated experiments, and is stuck to very low success rate for the data complexities we considered (yet, it eventually succeeded for larger number of measurements). This can be explained by two main reasons. First, the leakages on the permutation extracted with our templates (including the indirect ones) was larger than in our simulations, which naturally increases the gap between the integrating attack and the others. Second, the fact that different Atmel resources leak according to different models creates an additional noise for the integrating attack, due to a modeling error (i.e. these differences are lost after integration).

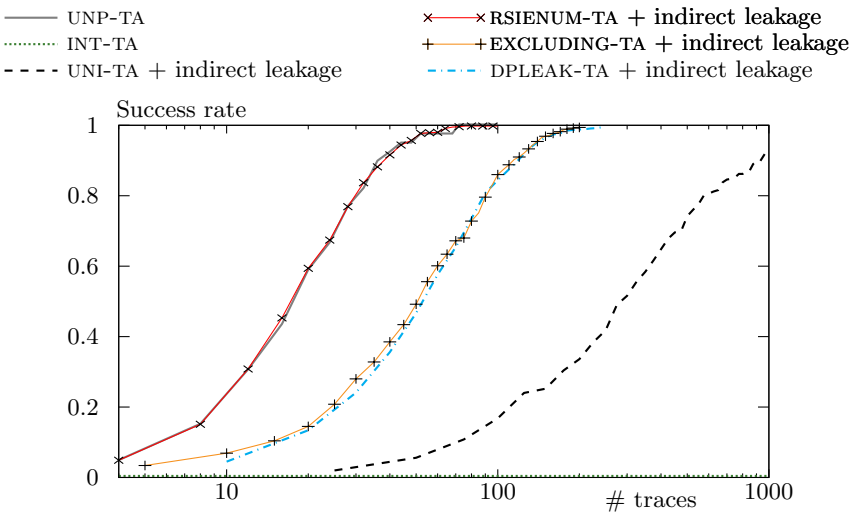

Fig. 3. Success rate of actual attacks on an ATMEL AVR implementation.

6 Conclusions

In this paper, we first proposed two new implementations of the shuffling countermeasure in small (e.g. 8-bit) microcontrollers. They respectively allow improved performances in terms of overall cycle count and online cycle count. Next, we provided the first comprehensive evaluation of the shuffling countermeasure, including worst-case Bayesian attacks. For this purpose, we described intuitive formulas capturing the different variants of shuffling, and integrated them in a general evaluation framework from Eurocrypt 2009. These evaluation tools allowed us to show that previously used integrated attacks may not be enough for assessing the security of a shuffled implementations. We put forward that simplifying the permutation generation (e.g. by using RSI rather than RP) can lead to a complete breakdown of the countermeasure if not too noisy measurements are available (which turned out to be verified in a practical case study). We also explained the computational origin of these weaknesses (i.e. their relation with the total amount of permutations that are considered in the countermeasure). Finally, we exhibited that indirect leakages may be available in shuffled implementations, due to the different leakage models of different resources. This suggest an interesting scope of further research. Namely, since our results show that randomizing the order of instructions in cryptographic implementations is not always sufficient, can we design efficient ways to randomize both the execution order and the physical resources used in a cryptographic implementation?

Acknowledgements. This work has been funded in parts by the ERC project 280141 (acronym CRASH) and the 7th framework European project TAMPRES. S. Kerckhof is a PhD student funded by a FRIA grant. F.-X. Standaert is a Research Associate of the Belgian Fund for Scientific Research (FNRS-F.R.S).

References

1. Antoine Amarilli, Sascha Müller, David Naccache, Dan Page, Pablo Rauzy, and Michael Tunstall. Can code polymorphism limit information leakage? In Claudio Agostino Ardagna and Jianying Zhou, editors, *WISTP*, volume 6633 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2011.
2. Bertrand Anckaert, Matias Madou, and Koen De Bosschere. A model for self-modifying code. In Jan Camenisch, Christian S. Collberg, Neil F. Johnson, and Phil Sallee, editors, *Information Hiding*, volume 4437 of *Lecture Notes in Computer Science*, pages 232–248. Springer, 2006.
3. Atmel. http://www.atmel.com/products/microcontrollers/avr/.
4. Ali Galip Bayrak, Nikola Velickovic, Paolo Ienne, and Wayne Burleson. An architecture-independent instruction shuffler to protect against side-channel attacks. *TACO*, 8(4):20, 2012.
5. Çetin Kaya Koç and Christof Paar, editors. *Cryptographic Hardware and Embedded Systems - CHES 2000, Worcester, MA, USA, August 17-18, 2000, Proceedings*, volume 1965 of *Lecture Notes in Computer Science*. Springer, 2000.
6. 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.
7. Christophe Clavier, Jean-Sébastien Coron, and Nora Dabbous. Differential power analysis in the presence of hardware countermeasures. In Çetin Kaya Koç and Paar [5], pages 252–263.
8. Jean-Sébastien Coron. A new dpa countermeasure based on permutation tables. In Rafail Ostrovsky, Roberto De Prisco, and Ivan Visconti, editors, *SCN*, volume 5229 of *Lecture Notes in Computer Science*, pages 278–292. Springer, 2008.
9. Atmel Corporation. *8-bit Microcontroller with 16K/32K/64K Bytes In-System Programmable Flash - ATmega164P/V ATmega324P/V ATmega644P/V*, 2010. Rev. 8011O- 07/10, `http://www.atmel.com/images/8011s.pdf`.
10. Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.
11. Martin Feldhofer and Thomas Popp. Power Analysis Resistant AES Implementation for Passive RFID Tags. In Christiopher Lackner, Timm Ostermann, Michael Sams, and Ronal Spilka, editors, *Austrochip 2008*, pages 1–6, 2008.
12. 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.
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. Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Publishing, Boston, MA, USA, 1997.
15. Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.
16. Stefan Mangard. A Simple Power-Analysis (SPA) Attack on Implementations of the AES Key Expansion. In Pil Joong Lee and Chae Hoon Lim, editors, *ICISC*, volume 2587 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2002.

17. 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.
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. David May, Henk L. Muller, and Nigel P. Smart. Non-deterministic processors. In Vijay Varadharajan and Yi Mu, editors, *ACISP*, volume 2119 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2001.
20. 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.
21. Thomas S. Messerges. Using second-order power analysis to attack dpa resistant software. In Çetin Kaya Koç and Paar [5], pages 238–251.
22. Amir Moradi, Oliver Mischke, and Christof Paar. Practical evaluation of dpa countermeasures on reconfigurable hardware. In *HOST*, pages 154–160. IEEE Computer Society, 2011.
23. Bertram Poettering. Rijndael Furious, http://point-at-infinity.org/avraes/.
24. Emmanuel Prouff, Matthieu Rivain, and Régis Bevan. Statistical analysis of second order differential power analysis. *IEEE Trans. Computers*, 58(6):799–811, 2009.
25. Mathieu Renauld, François-Xavier Standaert, Nicolas Veyrat-Charvillon, Dina Kamel, and Denis Flandre. A formal study of power variability issues and side-channel attacks for nanoscale devices. In Kenneth G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 109–128. Springer, 2011.
26. 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.
27. 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 *LNCS*, pages 30–46. Springer, 2005.
28. 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.
29. 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.
30. Stefan Tillich and Christoph Herbst. Attacking state-of-the-art software countermeasures-a case study for aes. In Elisabeth Oswald and Pankaj Rohatgi, editors, *CHES*, volume 5154 of *Lecture Notes in Computer Science*, pages 228–243. Springer, 2008.
31. 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.
32. Weixiong Zhang. *State-space search - algorithms, complexity, extensions, and applications.* Springer, 1999.