On the Cost of Lazy Engineering for Masked Software Implementations

Josep Balasch[1], Benedikt Gierlichs[1], Vincent Grosso[2], Oscar Reparaz[1], François-Xavier Standaert[2].

[1] KU Leuven Dept. Electrical Engineering-ESAT/COSIC and iMinds Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium.
[2] ICTEAM/ELEN/Crypto Group, Université catholique de Louvain, Belgium.

Abstract. Masking is one of the most popular countermeasures to mitigate side-channel analysis. Yet, its deployment in actual cryptographic devices is well known to be challenging, since designers have to ensure that the leakage corresponding to different shares is independent. Several works have shown that such an independent leakage assumption may be contradicted in practice, because of physical effects such as "glitches" or "transition-based" leakages. As a result, implementing masking securely can be a time-consuming engineering problem. This is in strong contrast with recent and promising approaches for the automatic insertion of countermeasures exploiting compilers, that aim to limit the development time of side-channel resistant software. Motivated by this contrast, we question what can be hoped for these approaches – or more generally for masked software implementations based on careless assembly generation. For this purpose, our first contribution is a simple reduction from security proofs obtained in a (usual but not always realistic) model where leakages depend on the intermediate variables manipulated by the target device, to security proofs in a (more realistic) model where the transitions between these intermediate variables are leaked. We show that the cost of moving from one context to the other implies a division of the security order by two for masking schemes. Next, our second and main contribution is to provide an exhaustive empirical validation of this reduction, based on two microcontrollers, several (handwritten and compiler-based) ways of generating assembly codes, with and without "recycling" the randomness used for sharing. These experiments confirm the relevance of our analysis, and therefore quantify the cost of lazy engineering for masking.

1 Introduction

Masking is a widely deployed countermeasure to protect block cipher implementations against side-channel attacks. It works by splitting all the sensitive variables occurring during the computations into $d+1$ shares. Its security proofs (such as given, e.g. for the CHES 2010 scheme of Rivain and Prouff [24]) ensure the so-called dth-order property, which requires that *every tuple of at most d intermediate variables in the implementation is independent of any sensitive variable.* Ensuring this property ($\underline{\text{ideally}}$) guarantees that the smallest key-dependent statistical moment in the leakage distribution is $d+1$. It has been shown (in different, more or less specialized settings [5, 9, 20, 27]) that the data complexity of side-channel attacks against such implementations increases exponentially with the number of shares. More precisely, in the usual context of (close to) Gaussian noise, this data complexity is proportional to $(\sigma_n^2)^d$, with σ_n^2 the noise variance. In practice though, security proofs for masking heavily rely on an independence assumption. Namely, the ($\underline{\text{ideal}}$) hope is that the leakage function manipulates the shared intermediate variables independently. Whenever this assumption is not fulfilled, all bets are off regarding the security of the implementation. For example, a leakage function that would recombine the different shares would directly lead to an implementation that is as easy to attack as an unprotected one. As a result, the main question for the proofs in [5, 9, 20] to provide concrete security improvements is whether this assumption is respected in practice.

Unfortunately, experiments have shown that the independent leakage assumption does not always hold in actual hardware and software. Many physical defaults can be the cause of this issue. For hardware implementations, glitches are a well-identified candidate [14]. For software implementations, the problem more frequently comes from memory transitions (e.g. captured by a Hamming distance model) [7]. From this empirical observation, different strategies could be followed. One can naturally try to enforce independent leakages at the hardware or software level, but current research rather concludes negatively in both cases [7, 16]. A more promising approach is to deal with the problem at the algorithmic level. For example, threshold implementations and solutions based on multi-party computations can provide "glitch-freeness" [19, 25]. But the first solution is rather specialized to hardware devices (see, e.g. [4, 17] for applications to the AES), while the second one implies strong performance overheads [11]. In the following, we pursue a third direction for the software case, and investigate the security guarantees that can be obtained if we simply ignore the problem.

For this purpose, we start by formalizing the types of leakage functions that can be encountered in practice (namely value-based vs. transition based, generic vs. specific). As any formalization effort, we do not claim that it perfectly corresponds to actual measurements. Yet, we will show that it captures some important physical defaults to a sufficient extent for our conclusions to be supported by practical experiments. Next, our first contribution is to provide a couple of reductions from security claims obtained for one type of leakage functions to security claims for another type. Our most interesting result shows that a dth-order security proof obtained against value-based leakages leads to a $\lfloor \frac{d}{2} \rfloor$th-order security proof against transition-based ones. As the main question for such reductions to be relevant is whether they can be confirmed by actual implementations, our second and main contribution is to provide a comprehensive analysis of two case-studies of masked software (namely, in an Atmel AVR and an 8051 microcontrollers). More precisely, we show empirical evidence that implementations masked with two shares and proved first-order secure against value-based leakages are insecure in our devices with transition-based leakages, while three-share ones are indeed first-order secure in the same context. Furthermore, we show that our conclusions hold both for handwritten assembly codes and for C code compiled with various flags. We also study the impact of recycled randomness in these case studies. We finally combine these security analyses with an evaluation of the performance overheads due to the increased number of shares needed to reach a given masking order, and to sub-optimally compiled codes.

Besides their theoretical interest, we believe these conclusions are important for security engineers, since they answer a long standing open question regarding the automated insertion of countermeasures against side-channel attacks. Our proofs and experiments suggest that a single C code of a masked block cipher can indeed provide concrete security on two different devices, at the cost of an artificially increased number of shares. The overheads caused by this increased order correspond to the "cost of lazy engineering" suggested by our title, which is to balance with the significant gains in terms of development time that automation allows. As a result and maybe most importantly, these results validate an important line of research trying to exploit compilers to replace the manual insertion of countermeasures by expert developers [3, 18, 23]. Our findings suggest that such an approach is well founded for masking.

2 Definitions

Following previous works on masking, we denote any key-dependent intermediate variable appearing in an unprotected implementation as a *sensitive variable.* Taking the example of the secure multiplication of two shared secrets in Algorithm 1, x and y are sensitive variables. We further denote as *intermediate variables* the set of all the variables appearing in a masked

Algorithm 1 SecMult: secure multiplication of two shared secrets x and y [24].

```
Require: Shares $(x_i)_i$ and $(y_i)_i$ satisfying $\oplus_i x_i = x$ and $\oplus_i y_i = y$
Ensure: Shares $(w_i)_i$ satisfying $\oplus_i w_i = x \times y$
 1: for $i$ from 0 to $d$ do
 2:     for $j$ from $i+1$ to $d$ do
 3:         $r_{i,j} \in_R GF(2^n)$
 4:         $r_{j,i} \leftarrow (r_{i,j} \oplus x_i \times y_j) \oplus x_j \times y_i$
 5:     end for
 6: end for
 7: for $i$ from 0 to $d$ do
 8:     $w_i \leftarrow x_i \times y_i$
 9:     for $j$ from 0 to $d$, $j \neq i$ do
10:         $w_i \leftarrow w_i \oplus r_{i,j}$
11:     end for
12: end for
```

implementation. These intermediate variables should not be sensitive if masking is well implemented, since each share should be independent of the key in this case. For example, the set of intermediate variables in Algorithm 1 is given by:

$$\begin{aligned}\mathcal{V} = &\{x_i\} \cup \{y_i\} \cup \{r_{i,j}\} \cup \{x_i \times y_j\} \cup \{r_{i,j} \oplus x_i \times y_j\} \\ &\cup \{x_j \times y_i\} \cup \{(r_{i,j} \oplus x_i \times y_j) \oplus x_j \times y_i\} \cup \{x_i \oplus y_i\} \\ &\cup \{x_i \times y_i \oplus_{j=0}^{i-1} [(r_{i,j} \oplus x_i \times y_j) \oplus x_j \times y_i] \oplus_{j=i+1}^{d} r_{i,j}\}.\end{aligned} \tag{1}$$

The security proof of the masking scheme in [24] (and following works) was typically obtained for value-based leakage functions that we define as follows:

Definition 1 (Value-based leakage functions). *Let $\mathcal{V}$ be a set of intermediate variables and $\mathsf{L}(.) = \mathsf{L_d}(.) + N$ be a leakage function made of a deterministic part $\mathsf{L_d}(.)$ and an (additive) random noise N. This leakage function is value-based if its deterministic part can only take values $v \in \mathcal{V}$ as argument.*

By contrast, the flaws in [7] come from the fact that the software implementation considered by the authors was leaking according to a Hamming-distance model. The following transition-based leakage functions aim at formalizing this issue:

Definition 2 (Transition-based leakage functions). *Let $\mathcal{V}$ be a set of intermediate variables and $\mathcal{T} := \{v \oplus v' \mid \forall v, v' \in \mathcal{V}\} \cup \mathcal{V}$ be the set of all the transitions between these intermediate variables. A leakage function $\mathsf{L}(.)$ is transition-based if its deterministic part $\mathsf{L_d}(.)$ can only take values $t \in \mathcal{T}$ as argument.*

Note that this type of transitions, based on the bitwise XOR between the values v and v', is motivated by practical considerations (since it generalizes the Hamming distance model).

Yet, even more general types of transitions, e.g. the concatenation $v||v'$, would not change our following conclusions – it would only make the bound of Theorem 1 more tight in certain cases (see next).

We further define generic vs. specific leakage functions as follows:

Definition 3 (Generic leakage functions). *A value-based (resp. transition-based) leakage function associated with an intermediate variable $v \in \mathcal{V}$ (resp. transition $t \in \mathcal{T}$) is generic if its deterministic part is a nominal mapping from this variable to a leakage variable $l_{\mathrm{d}} \in \mathcal{L}_{\mathrm{d}}$, such that the set of deterministic leakages $\mathcal{L}_{\mathrm{d}}$ has the same cardinality as the set of values $\mathcal{V}$ (resp. transitions $\mathcal{T}$).*

The identity mapping is a typical example of generic leakage function[1].

Definition 4 (Specific leakage functions). *A value-based (resp. transition-based) leakage function associated with an intermediate variable $v \in \mathcal{V}$ (resp. transition $t \in \mathcal{T}$) is specific if its deterministic part is a mapping from this variable to a leakage variable $l_{\mathrm{d}} \in \mathcal{L}_{\mathrm{d}}$, such that the set of deterministic leakages $\mathcal{L}_{\mathrm{d}}$ has smaller cardinality than the set of values $\mathcal{V}$ (resp. transitions $\mathcal{T}$).*

The frequently considered Hamming weight and distance functions are typical examples of specific (value-based and transition-based) leakage functions.

3 Reductions

From these definitions, a natural question is whether a proof of security obtained within one model translates into a proof in another model. As we now detail, three out of the four possible propositions are trivial (we recall them for completeness). The last one is more intriguing and practically relevant.

Lemma 1. *A proof of dth-order side-channel security obtained within a generic model implies a proof of dth-order security in a specific model.*

Proof. This directly derives from Definitions 3 and 4. By moving from one to the other, we only reduce the amount of information provided to the adversary (since we reduce the cardinality of the set of possible deterministic leakages).

Lemma 2. *A proof of dth-order security obtained within a specific model does not imply a proof of dth-order security in a generic model.*

Proof. A counterexample can be found in [12] for low-entropy masking schemes.

Lemma 3. *A proof of dth-order side-channel security obtained within a transition-based model implies a proof of dth-order security in a value-based model.*

Proof. Similarly to Lemma 1, this directly derives from Definitions 2 and 1. By moving from one to the other, we only reduce the amount of information provided to the adversary (since we reduce the input range of the leakage function).

We will need the following lemma to prove our last result.

[1] This definition differs from the one of "generic power model" in [2] since it relates to the leakage function, while the latter one relates to the adversary's model.

Lemma 4. *The information obtained from any subset of at most $\lfloor \frac{d}{2} \rfloor$ elements in a set $\mathcal{T}$ can be obtained from a subset of d elements in a set $\mathcal{V}$.*

Proof. Let $\mathcal{S_T} \subset \mathcal{T}$ such that $\#(\mathcal{S_T}) < \lfloor \frac{d}{2} \rfloor$. We show that $\exists\ \mathcal{S_V} \subset \mathcal{V}$ such that $\#(\mathcal{S_V}) < d$, and $\mathcal{S_T}$ can be built from $\mathcal{S_V}$ as follows (with $\#(.)$ the cardinality of a set). $\forall t \in \mathcal{S_T}$, if $t \in \mathcal{V}$, then $\mathcal{S_V} = \mathcal{S_V} \cup \{t\}$, else $\exists\ v, v' \in \mathcal{V}$ such that $t = v \oplus v'$ and $\mathcal{S_V} = \mathcal{S_V} \cup \{v, v'\}$. Since $\#(\mathcal{S_T}) < \lfloor \frac{d}{2} \rfloor$, and we add at most 2 elements in $\mathcal{S_V}$ per element in $\mathcal{S_T}$, we directly have that $\#(\mathcal{S_V}) < d$.

It directly leads to the following theorem:

Theorem 1. *An dth-order secure implementation against value-based leakage functions is $\lfloor \frac{d}{2} \rfloor$th-order secure against transition-based leakage functions.*

Proof. If there existed a subset of transitions $\mathcal{S_T}$ with less than $\lfloor \frac{d}{2} \rfloor$ elements which can be used to mount a successful side-channel attack, then there would exist a subset $\mathcal{S_V}$ with less than d elements that can be used to mount a successful side-channel attack as well. As this second attack is impossible by hypothesis, such a set $\mathcal{S_T}$ cannot exist and the implementation is at least $\lfloor \frac{d}{2} \rfloor$th-order secure.

This bound is tight for Boolean masking. If $x = v_0 \oplus v_1 \oplus \dots v_{d-1} \oplus v_d$, we can see that $x = t_0 \oplus \dots \oplus t_{\lfloor \frac{d}{2} \rfloor}$, with $t_i = v_{2i} \oplus v_{2i+1}$ for $0 \leq i < \lfloor \frac{d}{2} \rfloor$ and $t_{\lfloor \frac{d}{2} \rfloor} = v_d$ if d even, and $t_{\lfloor \frac{d}{2} \rfloor} = v_{d-1} \oplus v_d$ if d is odd. By contrast, it is not tight for other types of masking schemes such as inner product or polynomial [1, 22]. However, it would be tight even for those masking schemes in the context of concatenation-based transitions (i.e. if using $v||v'$ rather than $v \oplus v'$ in Definition 2).

4 Experiments

In view of the simplicity of Theorem 1, one can naturally wonder whether it captures real-world situations. That is, is it sufficient for a careless designer to double the security-order to obtain some guarantees for his masked implementations. In the rest of the paper, we investigate this question in various practically-relevant scenarios. For this purpose, we will focus on secure S-box computations. As explained in [24], this is usually the most challenging part of a masked block cipher. In the case of AES that we will consider next, the method exploits a representation of the S-box with power functions in $\mathrm{GF}(2^8) \equiv \mathrm{GF}(2)[x]/x^8+x^4+x^3+x+1$ (see Algorithm 2). We will implement it for two key additions followed by two inversions (see Algorithm 3). Note that we are aware that the masked inversion scheme proposed by Rivain and Prouff exhibits a small bias as presented by Coron et. al. in [8]. As will be discussed later in the paper, the existence of this issue does not affect our results and conclusions.

Concretely, we made several implementations of Algorithm 3, which is complex enough to exercise registers, ALU, RAM and ROM. Note that we provide input plaintext and key bytes to the implementations in $d+1$ shares each. This ensures that the device does not process unmasked variables, unless the shares are explicitly combined by the implementation, which is highly relevant for our testing procedure. We investigate the impact of the following parameters.

- Programming language: we contrast handwritten assembly (ASM) and compiled C code. For both ASM and C we implemented straightforwardly with little attention to secure the implementations.
- Device architecture: we provide results for an Atmel AVR and for an 8051 compatible microcontroller.

Algorithm 2 SecInv: secure inversion of a shared secret x in GF(2^8).

```
Require: Shares $(x_i)_i$ satisfying $\oplus_i x_i = x$
Ensure: Shares $(y_i)_i$ satisfying $\oplus_i y_i = x^{-1}$
 1: for $i$ from 0 to $d$ do $z_i \leftarrow x_i^2$
 2: end for
 3: RefreshMasks$(z_0, z_1, \ldots, z_d)$
 4: $(y_o, y_1, \ldots, y_d) \leftarrow$ SecMult$((z_0, z_1, \ldots, z_d), (x_0, x_1, \ldots, x_d))$
 5: for $i$ from 0 to $d$ do $w_i \leftarrow y_i^4$
 6: end for
 7: RefreshMasks$(w_0, w_1, \ldots, w_d)$
 8: $(y_o, y_1, \ldots, y_d) \leftarrow$ SecMult$((y_0, y_1, \ldots, y_d), (w_0, w_1, \ldots, w_d))$
 9: for $i$ from 0 to $d$ do $y_i \leftarrow y_i^{16}$
10: end for
11: $(y_o, y_1, \ldots, y_d) \leftarrow$ SecMult$((y_0, y_1, \ldots, y_d), (w_0, w_1, \ldots, w_d))$
12: $(y_o, y_1, \ldots, y_d) \leftarrow$ SecMult$((y_0, y_1, \ldots, y_d), (z_0, z_1, \ldots, z_d))$
```

Algorithm 3 Masked key addition and inversion.

```
Require: Shares $(p_i^0)_i, (p_i^1)_i, (k_i^0)_i, (k_i^1)_i$ satisfying $\oplus_i p_i^0 = p^0, \oplus_i p_i^1 = p^1, \oplus_i k_i^0 = k^0, \oplus_i k_i^1 = k^1$; with
    $k^0$ fixed and $k^1 \neq k^0$ fixed
Ensure: Shares $(c_i^0), (c_i^1)$ satisfying $\oplus_i c_i^0 = (p^0 \oplus k^0)^{-1}, \oplus_i c_i^1 = (p^1 \oplus k^1)^{-1}$
 1: for $i$ from 0 to 1 do
 2:     for $j$ from 0 to $d$ do
 3:         $x_j \leftarrow p_j^i \oplus k_j^i$
 4:     end for
 5:     $(c_0^i, \ldots, c_d^i) \leftarrow$ SecInv$(x_0, \ldots, x_d)$
 6: end for
```

– Compiler flags: we assess the impact of compiler flags. We compiled the C code with default options and with several combinations of flags that influence the degree of optimization as well as the order in which registers are assigned.
– Masking order: we implemented everything for $d = 1$ (2 shares) and for $d = 2$ (3 shares).
– Mask re-use: since randomness is expensive on low cost microcontrollers an implementer might decide to re-use random masks. We contrast implementations that use *fresh* randomness for the processing of each input byte (initial masking, SecMult, RefreshMasks) and implementations that *recycle* the randomness from the processing of the first byte for the processing of the second byte. Since our microcontrollers do not have an internal source of randomness, we provide uniformly distributed random numbers from the measurement PC.

4.1 Implementation details

Our main target platform is an AVR ATmega163 microcontroller in a smart card body. It internally provides 16 kBytes of flash memory and 1 kByte of data memory. Implementations are processed by `avr-gcc` (ver. 4.3.3) from the WinAVR tools (ver. 20100110).

The implementation of the secure inversion of Algorithm 2 requires support for arithmetic in the finite field GF(2^8). Field addition/subtraction can be straightforwardly performed via the bitwise exclusive-or (XOR) operator. Multiplication over GF(2^8) is however more complex to implement. We choose to develop this operation by using an approach based on the so-called `log` and `alog` tables [28]. At the cost of storing $2 \times 256 = 512$ bytes, this technique allows to compute the product of two *non-zero* field elements with mainly three table lookups. If either of the inputs is zero, the result is simply zero. Despite being very efficient for software platforms, this technique has a major drawback: checking whether any of the inputs is zero can lead to exploitable timing or power leakages. To overcome this issue we select an SPA-resistant variant of the `log` and `alog` technique proposed in CHES 2011 by Kim et al. [13]. As illustrated in Algorithm 4, the security of this algorithm relies on avoiding the use of `if / else` statements. Instead, the return value of the routine is given by the product of r (the outcome of the `log` and `alog` lookups) and the result of a logical evaluation ($a \&\& b$) dependent on both inputs.

Algorithm 4 SPA-resistant multiplication over GF(2^8) [13].

```
Require: Field elements $a$, $b \in \mathrm{GF}(2^8)$, log and alog tables
Ensure: Field element $a \times b \in \mathrm{GF}(2^8)$
 1: $(c, s) = \texttt{log}[a] + \texttt{log}[b]$        /* $c$ holds carry bit, $s$ the lower 8 bits */
 2: $r = \texttt{alog}[c + s]$
 3: return $(a \&\& b) \cdot r$                         /* && indicates logical AND condition */
```

Assembly. Our assembly implementations are tailored to the target AVR architecture and optimized for speed. We have developed codes for each of the tested masking orders, i.e. one for $d = 1$ and one for $d = 2$. Our routine for field multiplication takes 22 cycles. More than a third of this time is devoted to achieve a constant flow of operations to securely implement line 3 in Algorithm 4. Both `log` and `alog` tables are stored in program memory. All raisings to the power of two are implemented as lookup tables in program memory. While this requires the storage of $3 \times 256 = 768$ bytes, it results in a significant performance increase. Further speed-ups are achieved by aligning all tables on a 256 byte boundary (0x100). This ensures all addresses of the cells differ only in the lower byte and allows for more efficient handling of pointers.

C language. One of the goals of our experiments is to devise and evaluate platform-independent C code. Declaring and accessing program memory arrays in AVR requires the use of special attributes in `avr-gcc`[2]. Consequently, we cannot take advantage of storing lookup tables in program memory and the implementation becomes more restricted in terms of storage than its ASM counterpart. Our C routine for multiplication over GF(2^8) follows the code given in Algorithm 4. The two `log` and `alog` tables take half of the available space in RAM. Because of this we opt to perform field squarings as field multiplications, i.e. without using lookup tables. This saves 768 bytes of memory arrays with respect to the assembly implementations, but results in larger execution times and more randomness requirements.

4.2 Testing procedure

The security evaluation of cryptographic implementations with respect to side-channel attacks is a topic of ongoing discussions and an open problem. Since long, implementations are evaluated (in academia) by testing their resistance to state-of-the-art attacks. However, it is well known that this is a time-consuming task with potentially high data and computational complexity. In addition, an implementation that resists known attacks may still have vulnerabilities that can be exploited by new attacks. Hence, this style of evaluation can lead to a false sense of security, but it also stimulates improvements of the state-of-the-art. In 2009, Standaert et al. [26] proposed a framework for the evaluation of cryptographic implementations w.r.t. side-channel attacks. For univariate analysis (i.e. analysis of each time sample separately), their information-theoretic metric shows how much information is available to an attacker in a worst-case scenario. It directly

[2] See http://www.nongnu.org/avr-libc/user-manual/pgmspace.html

corresponds to the success rate of a (univariate) template attack adversary and captures information present in *any* statistical moment of the leakage distributions. For multivariate analysis (i.e. joint analysis of time samples) the technique relies on heuristics regarding the selection of time samples, just as well as all state-of-the-art attacks. The technique has strong requirements w.r.t. data and computational complexity. For our evaluations, computing the metric is beyond feasible, but it would also be inappropriate as we are interested in testing specific statistical moments of the measured distributions for evidence of leakage (while a worst-case evaluation typically exploits all the statistical moments jointly). We therefore adopt the relatively novel approach to evaluation called *leakage detection*. Contrary to the classical approach of testing whether a given attack is successful, this approach decouples the detection of leakage from its exploitation. And contrary to the IT metric, this approach can be tuned in order to evaluate specific statistical moments of the measured distributions.

For our purpose we use the non-specific t-test based fixed versus random leakage detection methodology of [6, 10]. It has two main ingredients: first, chosen inputs allow to generate two sets of measurements for which intermediate values in the implementation have a certain difference. Without making an assumption about how the implementation leaks, a safe choice is to keep the intermediate values fixed for one set of measurements, while they take random values for the second set. The test is *specific*, if particular intermediate values or transitions in the implementation are targeted (e.g. S-box input, S-box output, Hamming distance in a round register, etc.). This type of testing requires knowledge of the device key and carefully chosen inputs. On the other hand, the test is *non-specific* if *all* intermediate values and transitions are targeted at the same time. This type of testing only requires to keep all inputs to the implementation fixed for one set of measurements, and to choose them randomly for the second set. Obviously, the non-specific test is extremely powerful. The second ingredient is a simple, robust and efficiently computable statistical test to determine if the two sets of measurements are significantly different (to be made precise below).

In our experiments, all implementations receive as input $4(d+1)$ shares $(p_i^0)_i$, $(p_i^1)_i$, $(k_i^0)_i$, $(k_i^1)_i$ of the plaintext and key bytes. The (unshared) key bytes (k_0, k_1) are fixed with $k_0 \neq k_1$. We obtain two sets of measurements from each implementation. For the first set, we fix the values $p^0 = k^0$ and $p^1 = k^1$ such that, without masking, the input of the inversion function would be zero, which is likely to be a "special" case. Indeed, all the intermediate results through the exponentation to the power of 254 would be zero. We denote this set $\mathcal{S}_{fixed}$. For the second set, the values of p_0 and p_1 are drawn at random from uniform. We denote this set $\mathcal{S}_{random}$. Note that we obtain the measurements for both sets interleaved (one fixed, one random, one fixed, on random, etc.) to avoid time-dependent external and internal influences on the test result. A power trace covers the execution of steps 1 to 6 in Algorithm 3.

We then compute Welch's (two-tailed) t-test:

$$t = \frac{\mu(\mathcal{S}_{fixed}) - \mu(\mathcal{S}_{random})}{\sqrt{\frac{\sigma^2(\mathcal{S}_{fixed})}{\#\mathcal{S}_{fixed}} + \frac{\sigma^2(\mathcal{S}_{random})}{\#\mathcal{S}_{random}}}}, \tag{2}$$

(where μ is the sample mean, σ^2 is the sample variance and $\#$ denotes the sample size) to determine if the samples in both sets were drawn from the same population (or from populations with the same mean). The *null hypothesis* is that the samples in both sets were drawn from populations with the same mean. In our context, this means that the masking is effective. The alternative hypothesis is that the samples in both sets were drawn from populations with different means. In our context, this means that the masking is not effective.

At each point in time, the test statistic t together with the degrees of freedom ν, computed with the Welch-Satterthwaite equation:

$$\nu = \frac{(\sigma^2(\mathcal{S}_{fixed})/\#\mathcal{S}_{fixed} + \sigma^2(\mathcal{S}_{random})/\#\mathcal{S}_{random})^2}{(\sigma^2(\mathcal{S}_{fixed})/\#\mathcal{S}_{fixed})^2/(\#\mathcal{S}_{fixed}-1) + (\sigma^2(\mathcal{S}_{random})/\#\mathcal{S}_{random})^2/(\#\mathcal{S}_{random}-1)}\,, \quad (3)$$

allow to compute a p value to determine if there is sufficient evidence to reject the null hypothesis at a particular significance level $(1-\alpha)$. The p value expresses the probability of observing the measured difference (or a greater difference) by chance if the null hypothesis was true. In other words, small p values give evidence to reject the null hypothesis.

As in any evaluation, one is left with choosing a threshold to decide if an observed difference is significant or not. Further, also this type of evaluation is limited by the number of measurements at hand. In case the test does not show sufficient evidence of leakage, repeating the same evaluation with more measurements might do. For all experiments performed in this work, we select a threshold of ± 5 for the t-statistic. More details on how we derive this value are provided in Appendix A.

4.3 Security results

We measure the power consumption of the AVR platform as the voltage drop over a 50 Ohm shunt resistor placed in the GND path. For all code evaluations we set the device's clock at 3.57 MHz and the oscilloscope's sampling rate at 250 MS/s. Results are presented in form of plots of t-values on the y-axis and time on the x-axis. Recall that the t-test is applied to each time sample individually. Superposed, we plot a threshold of ± 5 for the t-statistic. For clarity, an auxiliary trigger signal is inserted on the upper part of the figure to indicate the beginning and the end of each byte's processing, i.e. masked key addition followed by masked field inversion.

Assembly. We begin by evaluating the AVR assembly implementation corresponding to the masking order $d = 1$ (two shares). The results are shown in Figure 1. The first input byte is processed until time sample $\approx 3 \times 10^4$, while processing of the second byte starts at time sample $\approx 4 \times 10^4$. The left plot corresponds to the implementation with fresh randomness. The right plot is the result for recycled randomness. Both experiments are performed using a set of 1 000 measurements: 500 corresponding to $\mathcal{S}_{fixed}$ and 500 corresponding to $\mathcal{S}_{random}$.

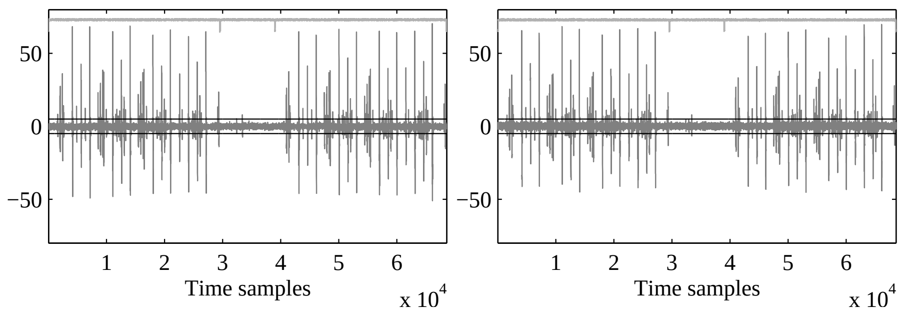

Fig. 1. T-test evaluation. Assembly, $d = 1$. Left: fresh randomness, 1k traces. Right: recycled randomness, 1k traces. Clear evidence of first-order leakage.

Figure 1 shows clear excursions of the t-test statistic beyond the defined thresholds, rejecting the null hypothesis. This indicates the existence of obvious univariate first-order leakage, in the form of identical patterns in each byte processing. There is no appreciable difference between using fresh versus recycled randomness. The outcome of this first experiment is however not surprising: as our platform is known to leak transitions, a (straightforward) implementation with masking order $d = 1$ is likely to be vulnerable to univariate attacks (see, e.g. [7] for similar findings). Perhaps more important, the results of the evaluation serve to validate the soundness of our testing methodology.

The situation changes when we evaluate the case $d = 2$ (three shares), as illustrated in Figure 2. Even by increasing the number of measurements to 10 000, the t-test fails to reject the null hypothesis for both scenarios. This indicates that any attack exploiting univariate first-order information (i.e., mean traces for each unshared value) is expected to fail, since there is no information about intermediate values in the first statistical moment. Interestingly, this result shows a first constructive application of Theorem 1. Starting with an implementation with second-order security in a value-based leakage model, we are able to achieve first-order security on a device with a transition-based leakage behavior. Finally, note that all our claims regarding the evaluation for $d = 2$ are restricted to first-order scenarios. In fact, attacks exploiting second or higher statistical moments are expected to succeed in breaking the implementation. We addressed this issue in more detail in Appendix B (together with the previously mentioned flaw exhibited at FSE 2013). Besides, and as already mentioned, all evaluations are inherently limited to the number of measurements at hand. In this respect, one may imagine that more measurements would allow detecting a first-order leakage. Yet, we note that in all our following experiments, whenever we claim no evidence of first-order leakages, second-order leakages were identified with confidence. This suggests that even if first-order leakages could be detected, their informativeness would be limited compared to second-order ones.

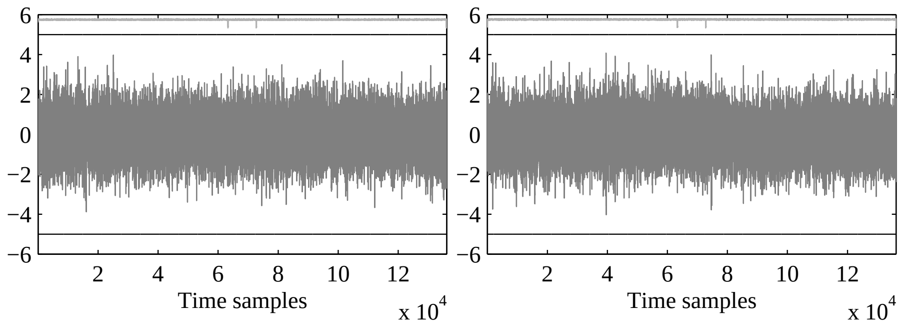

Fig. 2. T-test evaluation. Assembly, $d = 2$. Left: fresh randomness, 10k traces. Right: recycled randomness, 10k traces. No evidence of first-order leakage.

C language. A natural follow-up question is whether the results obtained so far hold for the case of C implementations. In the following we evaluate the results of our platform-independent C code. For the first set of tests we initially switch off the `avr-gcc` compiler flags for optimization, i.e. we use the option `-O0`.

Figure 3 shows the results obtained for the case $d = 1$ (two shares). As expected, the evaluation of the $d = 1$ implementation on our AVR platform indicates univariate first-order leakage. This result is consistent with its assembly counterpart. The main difference is that the absolute magnitude of the t-test at time samples beyond the ± 5 threshold is smaller, probably due to the leakage being more scattered. After all, code resulting from compiling C is expected to be more sparse code than concise, hand-crafted assembler. Illustrative of this effect is also the considerable increase in length of our measurements, from 70 000 samples to 1 200 000 samples.

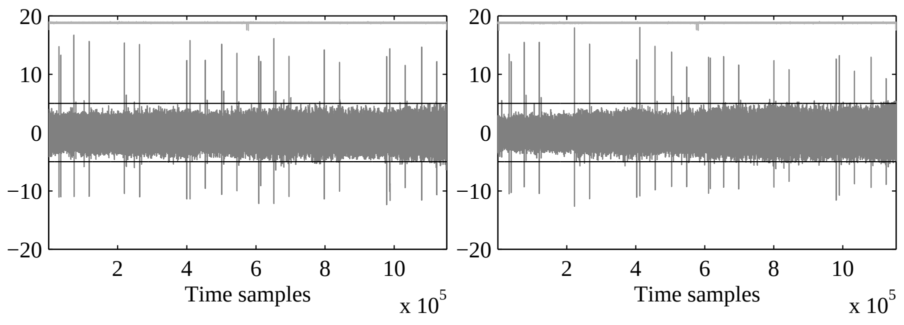

Fig. 3. T-test evaluation. C, no flags, $d = 1$. Left: fresh randomness, 1k traces. Right: recycled randomness, 1k traces. Clear evidence of first-order leakage.

The results obtained for $d = 2$ (three shares) are given in Figure 4. Here we observe a substantial difference between the fresh randomness and recycled randomness scenarios. While the left plot does not exhibit excursions beyond the threshold, the right plot does unexpectedly suggest clear univariate leakage. In fact, the t-test trace shows a particular pattern not bound to a few time samples. Rather differently, it gradually increases over time and it only appears during the second half of the trace, i.e. during the processing of the second input byte with recycled randomness.

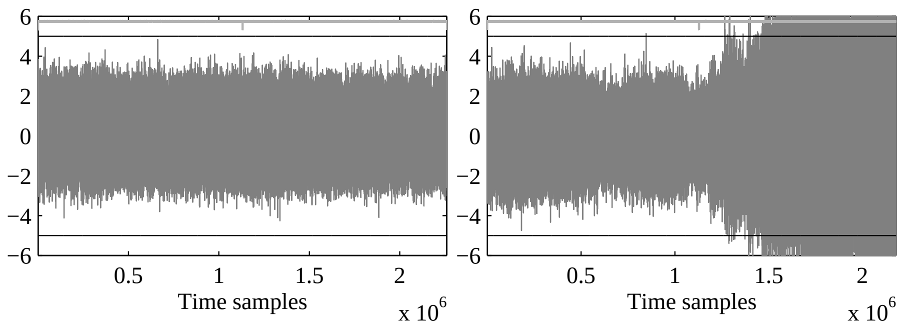

Fig. 4. T-test evaluation. C, $d = 2$. Left: fresh randomness, 10k traces. Right: recycled randomness, all C, 10k traces. Evidence of first order leakage.

We have verified that these results are caused by a non-constant time behavior of our compiled code. Although our C routines are written following seemingly constant-time and SPA-resistant algorithms [13], the `avr-gcc` compiler generates code with conditional execution paths. More specifically, the compiler transforms the boolean evaluation $a\&\&b$ into a series of `TST` (test for zero and minus) and `BREQ` (branch if equal) commands in assembly, regardless of the choice of compiler flags. This results in variable execution times (and flow) depending on the values of the input(s). From this, we conclude that the pseudo-code given in Algorithm 4 is equivalent to the original use of `if / else` statements, and therefore fails in providing resistance against SPA.

Note that it is unclear whether this leakage due to time variations can be exploited by univariate first-order attacks. While any practically exploitable first-order leakage will show up in the t-test curve, the contrary is not true, i.e. not all leakage identified by the t-test may be practically exploitable. In order to confirm the identified origin of the leakage, we implement a new C routine for multiplication in $GF(2^8)$ that does not directly evaluate the boolean condition $a\&\&b$. Instead, our code follows a series of time-constant operations which are equivalent to the boolean statement. The results obtained from evaluating this code are depicted in Figure 5. No obvious leakage is observed in either of the two scenarios, verifying that the shapes in Figure 4 are indeed caused by misalignments due to timing differences. As a downside, note that the performance of our platform-independent SPA-resistant code degrades significantly. The number of samples per measurement increases from 2 500 000 to 8 500 000, which in turn makes our analyses more difficult to carry out.

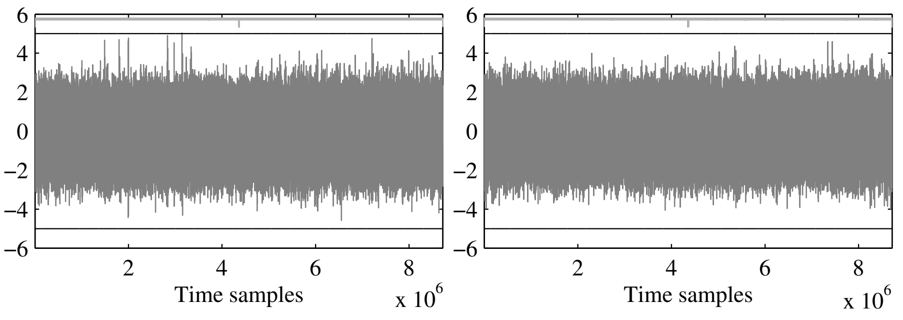

Fig. 5. T-test evaluation. C, $d = 2$, secure routine for multiplication in $GF(2^8)$. Left: fresh randomness, 10k traces. Right: recycled randomness, 10k traces. No evidence of first-order leakage.

These results are interesting since they moderate the applicability of Theorem 1 for compiled codes. That is, while this theorem nicely predicts the impact of transition-based leakages on the the security order of our implementations, it does not prevent the existence of other flaws due to a careless implementation leading to data-dependent execution times. That is, whenever taking advantage of compilers, designers should still pay attention to avoid such SPA flaws, e.g. by ensuring constant-time implementations. Note that this remains arguably easier task than ensuring DPA security, which therefore maintains the interest of our theorem even in this case.

Compiler options. A relevant scenario for the security evaluation of C code is to determine the impact of compiler flags. To this end, we provide in Appendix C the security evaluation for different compilation processes with `avr-gcc`. In particular, we analyze the effects for different degrees of optimization (flag `-O`) and for different assignment of registers (flag `-morder`). As can be seen in Figures 8, 9 and 10, these changes do not significantly affect our security conclusions. They do have however quite strong impact on the performance, in terms of both code size and cycle count. A detailed summary of the performance figures for each of the 30 combinations of compiler flags and masking orders is provided in Appendix D.

Other platforms. A final question of interest is to confirm whether the previous results hold for different devices than AVR. To this end, we perform a second set of experiments for the C implementations on an 8051 processor. Our results confirm that this is indeed the case, albeit with certain differences regarding the t-statistic shape and the number of traces required to achieve sound results. Because of space restrictions, this analysis is provided in Appendix E.

5 Concluding remarks

Confirmed by numerous experiments, the results in this paper first suggest a simple and natural way to convert security proofs obtained against value-based leakage models into security guarantees of lower order against transition-based ones. As a result, they bring a theoretical foundation to recent approaches to side-channel security, trying to automatically insert countermeasures such as masking in software codes. From a pragmatic point of view though, this positive conclusion should be moderated. On the one hand, just looking at the security order, we see that compiled codes can bring similar guarantees as handwritten assembly. On the other hand, reaching such a positive result still requires paying attention to SPA leakages (e.g. data-dependent execution times). Furthermore, compiled codes generally imply significant performance overheads. Yet, we hope that our results can stimulate more research in the direction of design automation for side-channel resistance, combining low development time and limited implementation overheads.

Acknowledgements. F.-X. Standaert is a research associate of the Belgian Fund for Scientific Research (FNRS-F.R.S.). Oscar Reparaz is funded by a PhD fellowship of the Fund for Scientific Research - Flanders (FWO). Benedikt Gierlichs is a Postdoctoral Fellow of the Fund for Scientific Research - Flanders (FWO). This work has been funded in parts by the European Commission through the ERC project 280141 (CRASH).

References

1. J. Balasch, S. Faust, B. Gierlichs, and I. Verbauwhede. Theory and practice of a leakage resilient masking scheme. In X. Wang and K. Sako, editors, *ASIACRYPT*, volume 7658 of *LNCS*, pages 758–775. Springer, 2012.
2. L. Batina, B. Gierlichs, E. Prouff, M. Rivain, F.-X. Standaert, and N. Veyrat-Charvillon. Mutual information analysis: a comprehensive study. *J. Cryptology*, 24(2):269–291, 2011.
3. A. G. Bayrak, F. Regazzoni, D. N. Bruna, P. Brisk, F.-X. Standaert, and P. Ienne. Automatic application of power analysis countermeasures. *IEEE Transactions on Computers*, 99(PrePrints):1, 2013.
4. B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen. A more efficient AES threshold implementation. *IACR Cryptology ePrint Archive*, 2013:697, 2013.
5. S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards sound approaches to counteract power-analysis attacks. In M. J. Wiener, editor, *CRYPTO*, volume 1666 of *LNCS*, pages 398–412. Springer, 1999.
6. J. Cooper, E. DeMulder, G. Goodwill, J. Jaffe, G. Kenworthy, and P. Rohatgi. Test Vector Leakage Assessment (TVLA) methodology in practice. International Cryptographic Module Conference, 2013. http://icmc-2013.org/wp/wp-content/uploads/2013/09/goodwillkenworthtestvector.pdf.
7. J.-S. Coron, C. Giraud, E. Prouff, S. Renner, M. Rivain, and P. K. Vadnala. Conversion of security proofs from one leakage model to another: A new issue. In W. Schindler and S. A. Huss, editors, *COSADE*, volume 7275 of *LNCS*, pages 69–81. Springer, 2012.
8. J.-S. Coron, E. Prouff, M. Rivain, and T. Roche. Higher-order side channel security and mask refreshing. To appear in the proceedings of FSE 2013.
9. A. Duc, S. Dziembowski, and S. Faust. Unifying leakage models: from probing attacks to noisy leakage. Cryptology ePrint Archive, Report 2014/079, 2014.
10. G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi. A testing methodology for side channel resistance validation. NIST non-invasive attack testing workshop, 2011. http://csrc.nist.gov/news_events/non-invasive-attack-testing-workshop/papers/08_Goodwill.pdf.

11. V. Grosso, F.-X. Standaert, and S. Faust. Masking vs. multiparty computation: How large is the gap for AES? In G. Bertoni and J.-S. Coron, editors, *CHES*, volume 8086 of *LNCS*, pages 400–416. Springer, 2013.

12. V. Grosso, F.-X. Standaert, and E. Prouff. Low entropy masking schemes, revisited. To appear in the proceedings of CARDIS 2013.

13. H. Kim, S. Hong, and J. Lim. A Fast and Provably Secure Higher-Order Masking of AES S-Box. In B. Preneel and T. Takagi, editors, *CHES*, volume 6917 of *LNCS*, pages 95–107. Springer, 2011.

14. S. Mangard, T. Popp, and B. M. Gammel. Side-Channel Leakage of Masked CMOS Gates. In A. Menezes, editor, *CT-RSA*, volume 3376 of *LNCS*, pages 351–365. Springer, 2005.

15. L. Mather, E. Oswald, J. Bandenburg, and M. Wójcik. Does My Device Leak Information? An a priori Statistical Power Analysis of Leakage Detection Tests. In K. Sako and P. Sarkar, editors, *ASIACRYPT*, volume 8269 of *LNCS*, pages 486–505. Springer, 2013.

16. A. Moradi and O. Mischke. Glitch-free implementation of masking in modern FPGAs. In *HOST*, pages 89–95. IEEE, 2012.

17. A. Moradi, A. Poschmann, S. Ling, C. Paar, and H. Wang. Pushing the limits: A very compact and a threshold implementation of AES. In K. G. Paterson, editor, *EUROCRYPT*, volume 6632 of *LNCS*, pages 69–88. Springer, 2011.

18. A. Moss, E. Oswald, D. Page, and M. Tunstall. Compiler assisted masking. In E. Prouff and P. Schaumont, editors, *CHES*, volume 7428 of *LNCS*, pages 58–75. Springer, 2012.

19. S. Nikova, V. Rijmen, and M. Schläffer. Secure hardware implementation of nonlinear functions in the presence of glitches. *J. Cryptology*, 24(2):292–321, 2011.

20. E. Prouff and M. Rivain. Masking against side-channel attacks: A formal security proof. In T. Johansson and P. Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *LNCS*, pages 142–159. Springer, 2013.

21. E. Prouff, M. Rivain, and R. Bevan. Statistical analysis of second order differential power analysis. *IEEE Trans. Computers*, 58(6):799–811, 2009.

22. E. Prouff and T. Roche. Higher-order glitches free implementation of the AES using secure multiparty computation protocols. In B. Preneel and T. Takagi, editors, *CHES*, volume 6917 of *LNCS*, pages 63–78. Springer, 2011.

23. F. Regazzoni, A. Cevrero, F.-X. Standaert, S. Badel, T. Kluter, P. Brisk, Y. Leblebici, and P. Ienne. A design flow and evaluation framework for DPA-Resistant instruction set extensions. In C. Clavier and K. Gaj, editors, *CHES*, volume 5747 of *LNCS*, pages 205–219. Springer, 2009.

24. M. Rivain and E. Prouff. Provably secure higher-order masking of AES. In S. Mangard and F.-X. Standaert, editors, *CHES*, volume 6225 of *LNCS*, pages 413–427. Springer, 2010.

25. T. Roche and E. Prouff. Higher-order glitch free implementation of the AES using secure multi-party computation protocols - extended version. *J. Cryptographic Engineering*, 2(2):111–127, 2012.

26. F.-X. Standaert, T. Malkin, and M. Yung. A unified framework for the analysis of side-channel key recovery attacks. In A. Joux, editor, *EUROCRYPT*, volume 5479 of *LNCS*, pages 443–461. Springer, 2009.

27. F.-X. Standaert, N. Veyrat-Charvillon, E. Oswald, B. Gierlichs, M. Medwed, M. Kasper, and S. Mangard. The World Is Not Enough: Another Look on Second-Order DPA. In M. Abe, editor, *ASIACRYPT*, volume 6477 of *LNCS*, pages 112–129. Springer, 2010.

28. E. D. Win, A. Bosselaers, S. Vandenberghe, P. D. Gersem, and J. Vandewalle. A Fast Software Implementation for Arithmetic Operations in GF(2^n). In K. Kim and T. Matsumoto, editors, *ASIACRYPT*, volume 1163 of *Lecture Notes in Computer Science*, pages 65–76. Springer, 1996.

A On the choice of the t-test threshold

Our testing methodology requires to set a threshold value to choose whether an observed difference in means of the two sets is significant or not. Typical significance levels in statistics are 0.05 and 0.00001 [15]. However, here we aim at choosing the threshold in a less arbitrary, data-driven way. To this end, we run a test "random-vs-random". In this test, measurements in both groups come from the same population (population of traces with random plaintext) so we know that the null hypothesis is true. We compute the t-statistic based on a random partition into two groups, keep its largest absolute value and repeat the experiment 200 times each iteration with a random partition into two sets. The highest absolute t-value we observed was 5.6088. The fact that this value is so large can be attributed to the fairly long duration of the traces (5 million of time samples each). In light of this, we selected a more conservative significance threshold of 5.

For orientation, note that for large sample sizes observing a single t-value with absolute value greater than 4.5 approximately corresponds to a 0.001% probability of the null hypothesis being true [10].

B Bivariate leakage

To test for bivariate second-order leakage, we note that the t-test methodology can be extended to test the population means of "pre-processed" traces. In particular, to test bivariate second-order leakage one can extend the traces by feeding all possible tuples of time samples (i.e. pairs for $d = 1$, triples for $d = 2$) from each trace through a suitable combination function (e.g., centered product [5, 21]), and then run the t-test on these extended traces. If the traces do not exhibit first-order leakage, and the extended traces do exhibit first-order leakage, this means that the traces exhibit second-order leakage.

In Figure 6 we depict the result of a bivariate second-order leakage detection test on the Assembly, $d = 2$ implementation with fresh randomness, run for the two time sample windows displayed in Figure 7. A t-statistic with absolute value less than 5 is a white pixel and non-white pixels mean a T-value greater than 5. The results clearly show evidence of second-order bivariate leakage, scattered over hundreds of time samples. The largest t-value from this plot is 10.3.

Note that at FSE 2013, Coron et. al. presented a flaw in the masking scheme of Rivain and Prouff presented at CHES 2010 implemented in this paper [8]. In the case of $d = 2$, they show that there exists a bias between two variables belonging to the SecMult and Refresh routine. However, the bias is so small that even in a simulated environment with 1 million traces and no noise it is not exploitable. The bias only becomes relevant when the SNR is poor ($\text{SNR} < 1/8$). In contrast, the bivariate second-order leakage detected in our implementation using 10k traces from Figure 6 is quite strong and can be detected in multiple different windows. Hence, we are not exploiting the (weak) bivariate leakage from [8] but a much stronger leakage arising from the specific leakage behavior of our platform, as we aim to analyze in this paper.

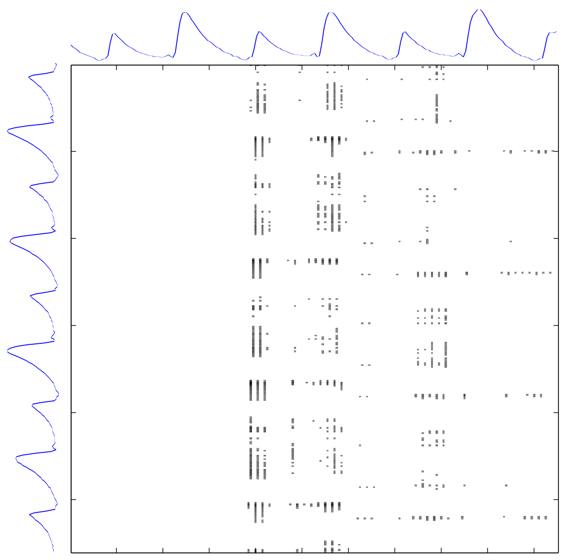

Fig. 6. T-test on the combination of two time samples from two time windows, as in Figure 7, shown onto the bi-dimensional region. Time flows from upper left corner to lower right.

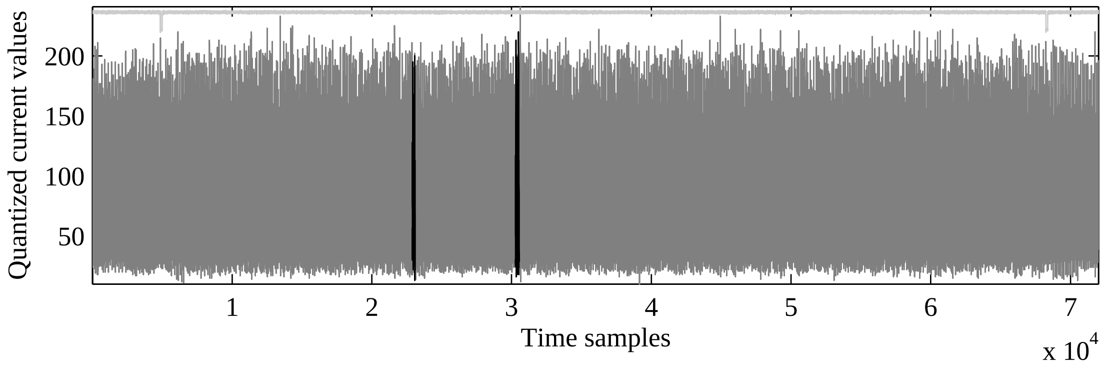

Fig. 7. Exemplary power trace for assembler, $d = 2$, fresh randomness, with two time windows highlighted in black. These two time windows exhibit bivariate leakage as Figure 6 shows.

C Compiler options

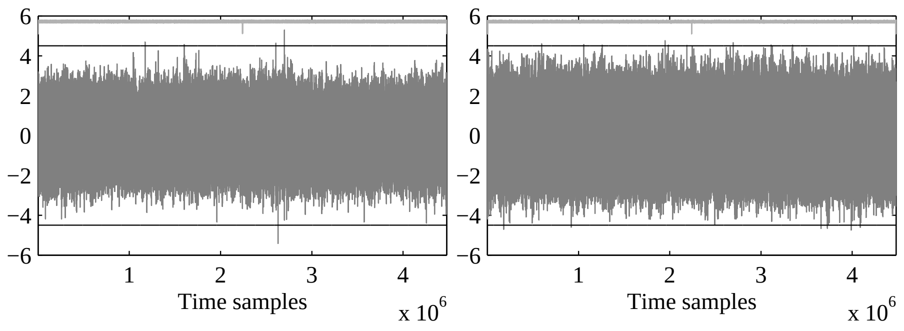

Fig. 8. T-test evaluation. C, $d = 2$. -O1. Left: fresh, right: recycled randomness

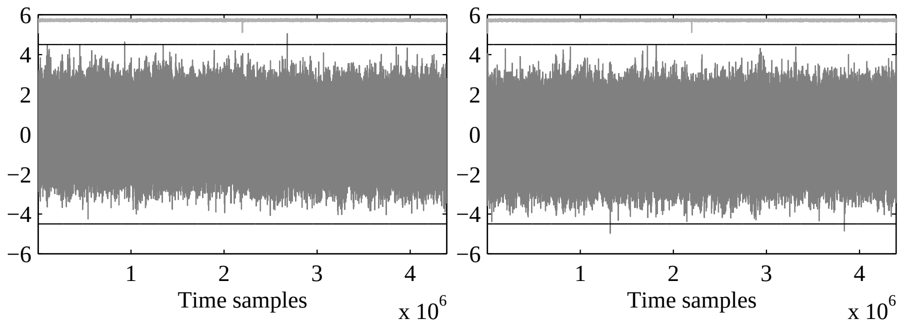

Fig. 9. T-test evaluation. C, $d = 2$. -O2. Left: fresh, right: recycled randomness

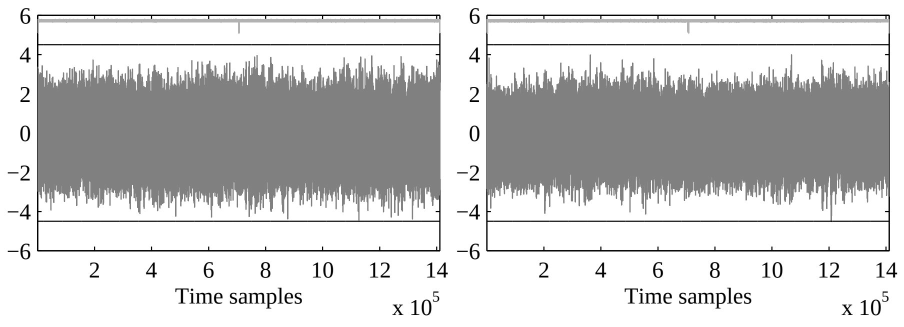

Fig. 10. T-test evaluation. C, $d = 2$. -O3. Left: fresh, right: recycled randomness

D Implementation results

In order to complement our security analyses, Table 1 shows the performance results of our various implementations. As one may expect, the implementations leading to a better speed vs. memory trade-off are programmed in assembly. The fastest C implementations (with flag `-O3`) are ten times slower than their assembly counterpart. Recall that due to data memory constraints, C implementations perform field squaring as field multiplication. In addition, achieving a time and flow constant implementation of Algorithm 1 in C is more complex than in assembly. In fact, while a multiplication over $GF(2^8)$ in assembly takes 22 cycles, the fastest one achieved in C (again with flag `-O3`) requires 150 cycles. This explains the great difference in performance numbers.

Table 1. Implementation results for masking order $d = 1$ (left) and $d = 2$ (right).

Language	Flags	ROM (bytes)	Speed (cycles)
ASM	n/a	2 820	627
C	`-O0`	2 806	38 005
C	`-O1`	1 776	18 611
C	`-O2`	1 626	17 677
C	`-O3`	3 866	5 017
C	`-Os`	1 606	17 722
C	`-morder1 -O0`	2 926	38 116
C	`-morder1 -O1`	1 770	18 341
C	`-morder1 -O2`	1 630	17 669
C	`-morder1 -O3`	3 874	5 017
C	`-morder1 -Os`	1 610	17 714
C	`-morder2 -O0`	2 818	38 487
C	`-morder2 -O1`	1 780	18 645
C	`-morder2 -O2`	1 634	17 939
C	`-morder2 -O3`	3 868	5 029
C	`-morder2 -Os`	1 614	17 984

Language	Flags	ROM (bytes)	Speed (cycles)
ASM	n/a	3 588	1 168
C	`-O0`	2 886	72 880
C	`-O1`	1 956	35 752
C	`-O2`	2 018	35 083
C	`-O3`	4 310	11 211
C	`-Os`	2 002	35 443
C	`-morder1 -O0`	3 006	73 018
C	`-morder1 -O1`	1 952	35 247
C	`-morder1 -O2`	2 024	35 071
C	`-morder1 -O3`	4 318	11 051
C	`-morder1 -Os`	2 010	35 443
C	`-morder2 -O0`	2 898	73 056
C	`-morder2 -O1`	1 958	35 809
C	`-morder2 -O2`	2 030	35 600
C	`-morder2 -O3`	4 312	11 139
C	`-morder2 -Os`	2 014	35 960

E Confirmation of our results on an 8051-compatible platform.

In this setup, both program and data memory are provided as external components. We process our C implementations using the Keil C51 toolchain (v9.02) and setting the compiler flags to speed optimization. The ASIC core is clocked at 7 MHz and the sampling rate of the oscilloscope is set at 250 MS/s. Power measurements are obtained by capturing the voltage drop over a 50 Ohm resistor in the V_{dd} path.

The evaluation results are illustrated in Figure 11 for the case of fresh randomness. The left plot depicts the outcome of the t-test for $d = 1$ (2 shares). The existence of univariate first-order leakage is confirmed by clear peaks appearing symmetrically along the processing of each byte. The shape of the excursions beyond the ± 5 threshold is different than the one obtained for the AVR. Also, we require to run the t-test evaluation with a larger number of measurements in order to clearly detect first-order leakage. As usual in the context of empirical evaluations, such a situation is hard to explain formally. Nevertheless, we believe two main reasons are the cause for this. First, the more noisy nature of the measurement setup. And second, the less leaky behavior of the targeted 8051 core. For the sake of completeness, we present the results for

$d = 2$ (3 shares) in the right plot of Figure 11. Similar to AVR, there is no evidence of first-order leakage after processing 10 000 traces. Although we expect second-order leakage to be present in these measurements, we have not attempted to detect it. The reason for this is the expensive computation and storage required to jointly process all possible samples pairs within such long traces (of millions of time samples).

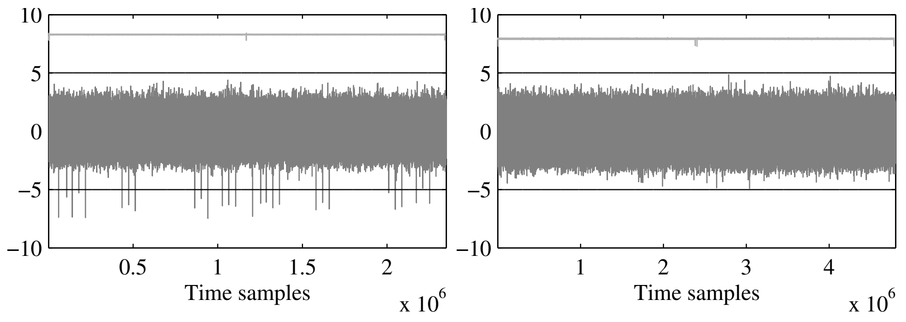

Fig. 11. T-test evaluation. C, 8051 platform, fresh randomness. Left: $d = 1$, 10k traces. Right: $d = 2$, 10k traces. First-order leakage visible only in the left plot.