Security evaluation against side-channel analysis at compilation time

Nicolas Bruneau[1,2,⋆], Charles Christen[3], Jean-Luc Danger[4,1], Adrien Facon[1,5], and Sylvain Guilley[1,4,5]

[1] Secure-IC S.A.S, Cesson-Sévigné, 35510, France
[2] STMicroelectronics, Rousset, 13790, France
[3] Direction Générale de l'Armement, Bruz, 35170, France
[4] Télécom-Paris, Institut Polytechnique de Paris, 91400 Saclay, France
[5] Département d'informatique de l'ENS, CNRS, PSL University, 75005 Paris, France

Abstract. Masking countermeasure is implemented to thwart side-channel attacks. The maturity of high-order masking schemes has reached the level where the concepts are sound and proven. For instance, Rivain and Prouff proposed a full-fledged AES at CHES 2010. Some non-trivial fixes regarding refresh functions were needed though. Now, industry is adopting such solutions, and for the sake of both quality and certification requirements, masked cryptographic code shall be checked for correctness using the same model as that of the the theoretical protection rationale (for instance the probing leakage model).
Seminal work has been initiated by Barthe et al. at EUROCRYPT 2015 for automated verification at higher orders on concrete implementations. In this paper, we build on this work to actually perform verification from within a compiler, so as to enable timely feedback to the developer. Precisely, our methodology enables to provide the actual security order of the code at the intermediate representation (IR) level, thereby identifying possible flaws (owing either to source code errors or to compiler optimizations). Second, our methodology allows for an exploitability analysis of the analysed IR code. In this respect, we formally handle all the symbolic expressions in the static single assignment (SSA) representation to build the optimal distinguisher function. This enables to evaluate the most powerful attack, which is not only function of the masking order d, but also on the number of leaking samples and of the expressions (e.g., linear vs non-linear leakages).
This scheme allows to evaluate the correctness of a masked cryptographic code, and also its actual security in terms of number of traces in a given deployment context.

Key words: Cryptographic code, compilation, intermediate representation (IR), static single assignment (SSA), side-channel analysis, masking protection, compositional countermeasure, formal analysis, optimal side-channel attacks, Taylor expansion of distinguishers.

⋆ Work done while at Secure-IC S.A.S.

1 Introduction

Context. With the massive deployment of Internet of Things (IoT), many devices are placed in-the-field which handle sensitive information. Typically, they must authenticate themselves, hence protect the integrity of public keys, and they handle private information, hence must ensure the confidentiality of secret encryption keys. The IoT devices are programmed in software, and their cryptographic stack deserves special attention.

The industry has put forward methodologies to ensure the protection of keys. A survey on this topic is freely available as a Technical Report from the ETSI [21]. For instance, cryptographic keys derivation, storage, and usage are confined in so called Trusted Execution Environments (TEEs). It is customary to study the security of digital assets according to their usage: data *at rest*, *in transit*, and *in computation*. The TEE takes care of keys at rest; cryptographic mechanisms, such as key establishment and key wrapping, allow to protect keys in transit; the hard problem is that of protecting keys in computation.

Indeed, software cryptographic code is vulnerable to side-channel attacks. They come in two flavors: first those which exploit conditional control-flow and/or table lookups (such as substitution boxes, or sboxes S), which fall into the class of so-called cache-timing attacks. Software techniques exist to make control-flow and table lookups uniform. In such situation, implementations are still vulnerable to a second kind of side-channel, which consists in physical spying of manipulated values, thanks to power or electromagnetic analysis.

State-of-the-art. Several approaches have been put forward. One of them consists in the use of whitebox cryptography (WBC). The idea is that even if an attacker has access to the code (including keys) in source code, there is no way for her to extract the key. The technique is based on data obfuscation in the code. WBC implementations aim an linearizing the control flow and hiding data into obscure (random-looking) tables. However, WBC has little (practical) theoretical foundation, and without surprise, several structural attacks have been demonstrated [10]. Other approaches revolve around "signal-to-noise" minimization. One approach is to cancel the signal leakage, through balancing. This suits leakage in the control-flow: typically, algorithms such as AES have a data-independent control flow, therefore this kind of leakage is easy to plug perfectly. Now, so-called vertical leakage (that is, leakage of values, cf. ISO/IEC 20085-1 [23]) is harder to cancel. Balancing approaches are possible [26], albeit result in unpredictable security level and complex coding style. Another approach

is the "masking" countermeasure, which aims at introducing some noise in the implementation. It consists in randomly sharing sensitive variables so that a side-channel attacker collect many meaningless leakages, since information is dissolved into several shares (conventionally, the number of shares is denoted by d). Therefore, such protections consume a lot of randomness (which must be uniform, independent, etc.). Today, many masking protections are constructively designed to be dth-order secure. The parameter $d > 0$ is a design security metric whereby each tuple of strictly less than d shares is independent from the clear sensitive variables.

For masking schemes such as Ishai-Sahai-Wagner (CRYPTO 2003 [22]) or Rivain-Prouff (CHES 2010 [27]) the security proof is based on composability: it is possible to design widgets for basic operations (e.g., field addition and multiplication) which form a universal set. Subsequently, combining them allows to build arbitrary computations. Reuse of variables shall be dealt with cautiously. In practice, reused variables usually benefit from refresh. This kind of masking is mature from a theoretical standpoint, and therefore is diffusing in the industry.

The problem is therefore to attest of the actual security, not of the principle of masking, but on the very implementation under consideration. There is one field of research which consists in checking a complete implementation (Barthe et al., EUROCRYPT 2015 [2]—known as MaskVerif). Since the combinatorics of verification is large, the proof employs heuristics taylored for the masking scheme.

Now, in practice, the attacker must perform a dth-order attack. But the attacker will maximize her advantage, and so she is not expected to be satisfied by one combination of d shares. Here we face a paradox: the larger the order d, the more possible combinations, hence it is relevant to study whether in practice, the security level in terms of data complexity (number of traces to recover the key) is still increasing with parameter d.

In this paper, we show how to compute optimal attacks, with tradeoffs regarding data and computational complexities (as in Bruneau et al., J. of Cryptology 2018 [16]). We aim at analyzing real-world implementations, irrespective of the source code language they are written in. Moreover, we want to consider optimized code. For this reason, we analyze the intermediate representation generated by a compiler, and generate the formula for the multivariate high-order distinguisher after having simplified the leaking terms.

Results show that monovariate high-order attacks are underestimating the security level by orders of magnitude, especially for high noise levels.

Contributions. The contributions in this paper are as follows:

- Application of dth-order masking correctness verification on an implementation extracted from within a compiler (at the IR level, after all optimization passes);
- Use of automated proof tools paradigm to generate optimal distinguishers for side-channel attacks;
- Trade-off regarding attack computation and attack efficiency, based on a truncated Taylor expansion of the side-channel distinguisher;
- Cautionary note that increasing d[1] does not increase exponentially the number of traces to recover the key.

Outline. The rest of this paper is structured as follows. An introduction on masking schemes and their need for formal verification has already be given in the present Sec. 1. Next the state-of-the-art automation method for the verification of a full-fledged implementations is recalled in Sec. 2. This methodology and its coding is leveraged to build automatically optimal distinguishers to attack at best the implementation. This allows to derive the most realistic security level (as in practice, the attacker will face more harsh attacking conditions, such as a degraded leakage model). This is the topic of Sec. 3, where we also highlight compromise between data and computational efficiency. Eventually, conclusion and perspectives are provided in Sec. 4.

2 Reminder about automated proof of masking schemes

2.1 Multi-variate and high-order side-channel attacks

State-of-the-art attacks against masked software consist in manual construction of leakage models. For instance, evaluation is done in [1, §5.2] on a first-order masking scheme, by using an absolute difference between two samples of the leakage. In [7], a combination is used albeit with unchosen coefficients, since those are coefficients from discrete Fourier or Hartley transforms. In [13], the dimensionality is reduced by an additive combination of samples, weighted by a profile obtained in a characterization phase.

It shall be noticed that software execution of software implementing masking countermeasure might leak a variable multiple times. Typically,

[1] In this paper, we use the same letter d for the number of shares necessary to recover information on sensitive variables (designer's perspective) and the smallest attack order (evaluator's perspective). Actually, those values match in practice, assuming that the implementation is not flawed.

the variable can be popped from the stack, then processed, and finally pushed back to the stack. It can also be copied at different places for different usages. Additionally, masking schemes themselves might involve multiple random variables. Typically, table recomputation countermeasure needs to address all entries of a lookup table, and process them with the same mask. Therefore, many samples in time leak the mask, and it is possible to combine them all in order to build the most efficient attack.

A methodology to build an attack of masked and shuffled table recomputation countermeasure is described in [16]. The method is empirical, because the *optimal distinguisher*, described in [14], is computationally too complex. The reason is that the optimal distinguisher shall be averaged over all masks m (if the countermeasure is of order d, m consists in a tuple of $(d-1)$ independent random variables). An approach is to develop the expression of the optimal distinguisher using a Taylor expansion, as described in [15]. This *systematic* approach allows for automation of attacks, by enabling a trade-off between accuracy of the distinguisher (Taylor expansion at high order) and computational complexity (Taylor expansion at low order). Namely, the mathematical formula for optimal attacks is explained below:

– Optimal attack consists in guess the key $\hat{k}$ according to maximum likelihood approach [14]:

$$\hat{k} = \underset{k}{\operatorname{argmax}} \sum_{q=1}^{Q} \log \sum_{m} \exp -\frac{1}{2\sigma^2} \big(x_q - f(t_q, k, m)\big)^2, \tag{1}$$

where:
- q is the traces index,
- t_q are the known texts, e.g., plaintexts,
- m are the masks (whose distribution does not depend on q),
- f is the leakage model, e.g.,

$$f(t_q, k, m) = w_H(S(t_q \oplus k) \oplus m) \tag{2}$$

  here assumed, but obtained by profiling in real attacks,
- x_q are the leakages $x_q = f(t_q, k^*, m) + n_q$, k^* being the correct key,
- σ^2 is the (centered) noise variance (n_q is a sample of this noise, that is $p_N(n_q) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp -\frac{n_q^2}{2\sigma^2}$).

– For attack simplification, the following Taylor expansion:

$$\log \mathbb{E} \exp(tX) = \sum_{n=1}^{\infty} \kappa_n \frac{t^n}{n!} \tag{3}$$

is leveraged

- for an expectation $\mathbb{E}$ over X, and
- where κ_n is the cumulant of order n of random variable X,

starting at order $n = d$ and stopping strictly before ∞ for tractability reasons[2].

In this article, we combine all the relevant tuples which leak information, up to a predetermined order chosen by the attacker. If this order is less than the protection order, then (provided the masking is perfect [11]), our algorithm finds no terms. This is consistent with the targeted security order: no attack shall be feasible hence the distinguisher is constant for all key hypotheses; the "argmax" in Eqn. 1 returns the full keyspace. Such result attests of countermeasure "correctness", meaning the formal verification that the countermeasure is "correctly" implemented (i.e., without flaws). But if we specify an order greater than that of the countermeasure, then our algorithm lists all the leaking terms, one shall consider to achieve the optimal attack rounded at a given order, as explicited theoretically in [15]. Besides, we assume that the code is constant-time (control flow does not depend on the data), hence traces are well aligned and can readily be used for performing a vertical high-order attack.

2.2 Analysis of code at Intermediate Representation

Applying countermeasures at the LLVM Intermediate Representation (IR) level, as produced by clang/LLVM's middle-end, has already been hinted in [6,5]. But this approach has limitations, as shown in [20], because [5] actually assumes as "masked data" any intermediate variable which even depends in a non-uniform way of masking material.

Like in "Side-Channel Robustness Analysis of Masked Assembly Codes using a Symbolic Approach" (PROOFS 2017, JCEN 2019 [8]), the control-flow of the program to analyse must be statically known; in particular, there must be no indirect jump and the number of loop iterations must be known at compile time. Fortunately, cryptographic algorithms usually fulfill this requirement, as well as their algorithms implementation. Indeed, otherwise, there is the possibility of a cache-timing attack (see for instance review in [17]).

[2] At infinite order, the expansion of Eqn. (3) is not considered, rather the original expression of Eqn. (1) is used.

2.3 Probing leakage model

In order to capture security correctness, the "probing leakage model" (stemming from initial work by Blömer et al. [11]) has been put forward. A design is dth-order secure if any tuple of strictly less than d intermediate variables carries no information on sensitive variables. Gilles Barthe and coauthors have been automating the verification of such property, recognized as that of "non-interference". It uses simplification heuristics, for instance to show that $M \oplus E$, where M is a randomly distributed mask and E an expression which does not depend on M, is simply distributed as another uniformly distributed random mask M'. M' cannot be distinguished from M, hence *sensitive* expression E is not exposed. It is able to attest of the soundness of a masking scheme, or to find explicit counter-examples. On top of these interesting results, we also provide the design of the best possible attack (namely the *optimal attack*).

3 Contribution: automated attack construction

3.1 Framework concept

The analyses we conduct on the cryptographic code are sketched in Fig. 1. The nominal compilation flow is represented in the leftmost column of the figure (on the running example of LLVM). The rest of the figure represents the security analysis. The column in the middle represents the symbolic analyses, based on expressions for each SSA. The rightmost column represents the concrete analysis, whereby variables are assigned values, as per a series of (say ≈ 100) attack simulations.

The compilation is conducted in a nominal way until intermediate representation is reached. Here, we also let optimization passes be executed. The outcome is a list of static single assignments (SSA). Altogether, these expressions are the inputs of our analysis.

(a) They are turned into symbolic expressions, named terms. They play the role of "intermediate variables" in side-channel papers such as [11].
  (a)-(1) Algorithm of MaskVerif is applied. The size d of the tuples is incremented (starting from $d = 1$) until we find a dependency into at least one tuple of expressions.
    * If this size is strictly inferior to the intended masking order, then a flaw has been found. Recall that as the analysis is at optimized IR level, the flaw can be structural (i.e., already present in the source code, as happened in the past for some countermeasures such as [29,18]) or caused by an optimization pass

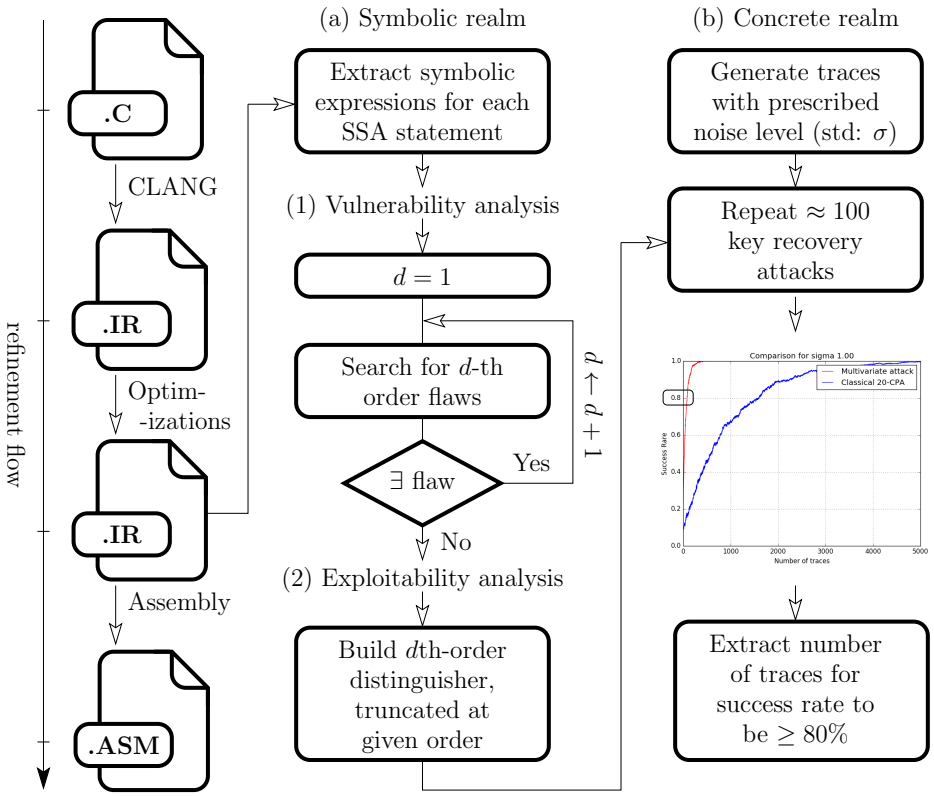

Fig. 1: Big picture of the two pipelined analyses conducted in the optimized intermediate representation code

which either removes some randomness or swaps operations (resulting in countermeasure break, see [28]).
    - Otherwise there is no flaw (result obtained by *formal proof*) and the masking order is exactly verified on the optimized IR representation. This however only indicates the absence of vulnerability from a design-for-security perspective. This means that the design matches its security specification.

  (a)-(2) The optimal distinguisher at order d (see generic formula in (1)) is derived. Its expression is rounded at a given order and simplified, for evaluation efficiency.

(b) Concrete security shall be attested by actual exploitation of the leaking terms. This complementary step is mandatory to have a concrete idea about the attacker effort to extract the key. Differences might show up owing to the multiplicity of leakages, the confusion in the terms (linear terms induce leakages harder to exploit, compared to sboxes), the variety of terms, etc. This is achieved based on simulated traces, computed by a leakage model (typically the classical Hamming weight model, cf. Eqn. (2)), and assuming a certain level of additive noise (i.i.d. for each expression and normally distributed as $\mathcal{N}(0, \sigma^2)$). The optimal distinguisher from step (a)-(2) is evaluated under q traces, and the indicator of attack success ($\hat{k} \stackrel{?}{=} k^*$) is computed. This process is reproduced ≈ 100 times, and averaged indicator yield the success rate. This curve is (globally–after smoothing) increasing. The number of traces for which success rate is equal to 80% is returned.

3.2 Framework implementation

Analysis parameters. The parameters of our analysis are:

- the order d of the masking in the input C files; they are immune to side-channel flaws if this order is equal to the one found in the vulnerability analysis (1) of Fig. 1.
- the optimal attack distinguisher Taylor expansion order ($\geq d$ if no flaw). In practice, an expansion at minimum degree (sufficient to distinguish) performs good results in terms of attack distinguishing power (meaning that the optimal attack extracts keys with similar number of traces). Higher degrees can sometimes help or sometimes not—this is difficult to estimate, and up to today, the exact degree where to cut the distinguisher off is considered an open problem.
- the leakage function associated with each share (i.e., each expression in the SSA extracted upon compilation), and

– the simulated measurement noise added to the leakage of each share (since masking works only as a countermeasure in the presence of some noise, hence evaluation must take place in this condition).

Source code to analyze. From a syntactic point of view, arbitrary source code can be inputted, since only optimized IR is analyzed. The countermeasures shall be applied at source-code level. Some constraints can nonetheless be enforced, such as `__attribute__((optnone))` in Listing 1.1 of Sec. A.1. The tool must nonetheless be informed about the name of the top-most function to analyze and the mask names. In the exemplar Listing 1.1, those are respectively "`sbox`" and "`r`" (short for `r`andom) at line 111.

Analysis toolset. The tools used in the analysis are described here-after:

(a) symbolic expressions are generated by an LLVM plugin called SAW, contributed by Galois Inc. on GitHub.
  (a)-(1) The expressions happen to be huge: they are simplified using SAGE. Then, the are parsed and loaded into Julia. A script into Julia performs the Non-Interference analysis for each tuple of d expressions, exactly as explained in MaskVerif [2].
  (a)-(2) The optimal distinguisher is computed (and simplified to the best extend, for instance by regrouping identical leakages).
(b) It is then applied on these traces. In practice, since the attack is computationally intensive, it is first translated in C language which is compiled with maximum amount of optimizations. Further, this C code itself contains many precomputations before the attack proper is launched.

3.3 Why check at the IR level?

The choice of the validation level (refer to vertical axis in Fig. 1) results from a compromise:

– it shall be as low as possible, for the validation to be as close as possible to the final product (notice that we do not expect to go as down as the concrete evaluation, since this will be the task of a third-party certification laboratory), while
– it shall be possible to conduct a formal analysis, so as to prove the security order (or to formally detect flaws) and to devise the best possible attack.

From this tradeoff, we position the analysis after the optimization passes, hence we analyze the IR code which is the closest to the actual assembly (i.e., machine code) which will be executed. This means that we detect all faults potentially caused by the compiler[3], which could break the countermeasure.

Moreover, from a practical point of view, the method consisting in outputting proof elements from the compiler allows to streamline the evaluation: the user codes the countermeasure in the language of its own choice, and then an automated verdict is provided[4].

3.4 Application

Codes applying a masking strategy as that put forward in [11] can be analyzed. A classical example is the Ishai-Sahai-Wagner computations [22], extending protection of bits (elements of $\mathbb{F}_2$) to words (elements of $\mathbb{F}_{2^l}$) [27]. A concrete code which can be analysed is listed in Appendix A.1.

In this article, we take as a running example the operations in a Galois field of characteristic two. Those are suitable for the computation of block ciphers, such as AES and even PRESENT [12]. AES fits naturally in $\mathbb{F}_{2^8}$ whereas PRESENT is nibble-oriented, hence can be represented in $\mathbb{F}_{2^4}$. In both AES and PRESENT, the only complex part regarding masking is the sbox, because it is non-linear. The expression of the PRESENT sbox (see table at page 453 of [12]) is obtained from Lagrange polynomial interpolation. We use MAGMA [31] to compute the extrapolation in $\mathbb{F}_{16}$ represented as $\mathbb{F}_2[x]/\langle x^4+x+1\rangle$. The element of this field $(\mathbb{F}_{16}, +, \cdot)$ are denoted according to the convention below:

- $[0, 1, 2, 3, 4, \ldots, 15]$ ..... (decimal)
- $[\texttt{0x0}, \texttt{0x1}, \texttt{0x2}, \texttt{0x3}, \texttt{0x4}, \ldots, \texttt{0xf}]$ ..... (hexadecimal)
- $[0, 1, x, x+1, x^2, \ldots, x^3+x^2+x+1]$ ..... (polynomial)

We get for the sbox of PRESENT the expression:

$$\begin{aligned} sbox(A) = {} & \texttt{0xc} + \texttt{0x7} \cdot A^2 + \texttt{0x7} \cdot A^3 + \texttt{0xe} \cdot A^4 + \texttt{0xa} \cdot A^5 + \texttt{0xc} \cdot A^6 \\ & + \texttt{0x4} \cdot A^7 + \texttt{0x7} \cdot A^8 + \texttt{0x9} \cdot A^9 + \texttt{0x9} \cdot A^{10} + \texttt{0xe} \cdot A^{11} \\ & + \texttt{0xc} \cdot A^{12} + \texttt{0xd} \cdot A^{13} + \texttt{0xd} \cdot A^{14}, \end{aligned}$$

[3] The LLVM IR is lowered to machine instructions, and some optimizations can still be performed on this representation. In particular, some memory accesses can be gathered, some peephole optimizations may remove some useless computations, selection of some instructions may disrupt the intended control flow, instruction scheduling may reorder computations and register allocation can introduce flaws.

[4] This is the way all Secure-IC pre-silicon tools, namely Virtualyzr® and Catalyzr®, work.

which is efficiently computed using Horner's method:

$$\begin{aligned} sbox(A) = \texttt{0xc} + A^2 \cdot (\texttt{0x7} + A \cdot (\texttt{0xe} + A \cdot (\texttt{0xa} + A \cdot (\texttt{0xc} + A \cdot (\quad \backslash \\ \texttt{0x4} + A \cdot (\texttt{0x7} + A \cdot (\texttt{0x9} + A \cdot (\texttt{0x9} + A \cdot (\texttt{0xe} + A \cdot (\quad \backslash \\ \texttt{0xc} + A \cdot (\texttt{0xd} + A \cdot (\texttt{0xd} + A)))))))))))). \end{aligned}$$

This expression is implemented in C language in function `sbox` in the Listing 1.1 of Appendix A.1. The attacked function is actually the composition of `AddRoundKey` and `sBoxLayer`, namely $A \mapsto \texttt{sbox}(A \oplus k^*)$, as done customarily in side-channel analysis.

Also, for the sake of tractability, we provide examples on masking of order $d = 1$. Indeed, our framework (described in previous Sec. 3.1) is, as of today, limited in the "concrete realm" of Fig. 1. The attack part is slow, though it has been translated from Julia to C for more efficiency. We leave the case $d > 1$ as a venue for further work, and as we shall see, results for $d = 1$ are already very rich.

One shall beware of the way the multiplication is performed, indeed, it is known that lookup-table based multiplications in characteristic two Galois fields do leak information [19]. In our implementation, the multiplication is constant-time (see function `mult` at line 19 of Listing 1.1).

Using the tool, we prove the correctness of code in Listing 1.1 of Appendix A.1. Notice that slight changes, such as asserting macro at line 50, would have the tool detect a first-order flaw. On the same code, we compare the traditional second-order (bi-variate) attack with our attack (multi-variate analysis, denoted MVA, extracted from the compiler). For these attacks, we use a Taylor expansion at order 2. The results for a few selected noise levels (in terms of noise standard deviation σ) are represented in Fig. 2. It clearly appears that the optimal attack, even though rounded at order 2 (i.e., the first order at which there is a leakage, since the implementation is first-order secure), is significantly more efficient than the customary 2O-CPA.

The multivariate attack is performing better and better relative to the classical bi-variate attack as the noise level increases. This was already remarked in some papers, such as those analysing substitution table masking with recomputation [30,16][5].

[5] Battistello et al. [4] also notice that great multiplicity helps attacks, albeit in the different context of low-noise implementations (e.g., software running on top of a CPU). Anyway, such results highlight well that high dimensionality significantly favors the attackers, and that this aspect is often overlooked when simply analysing the security of a masking scheme only in terms of its degree (i.e., number of shares).

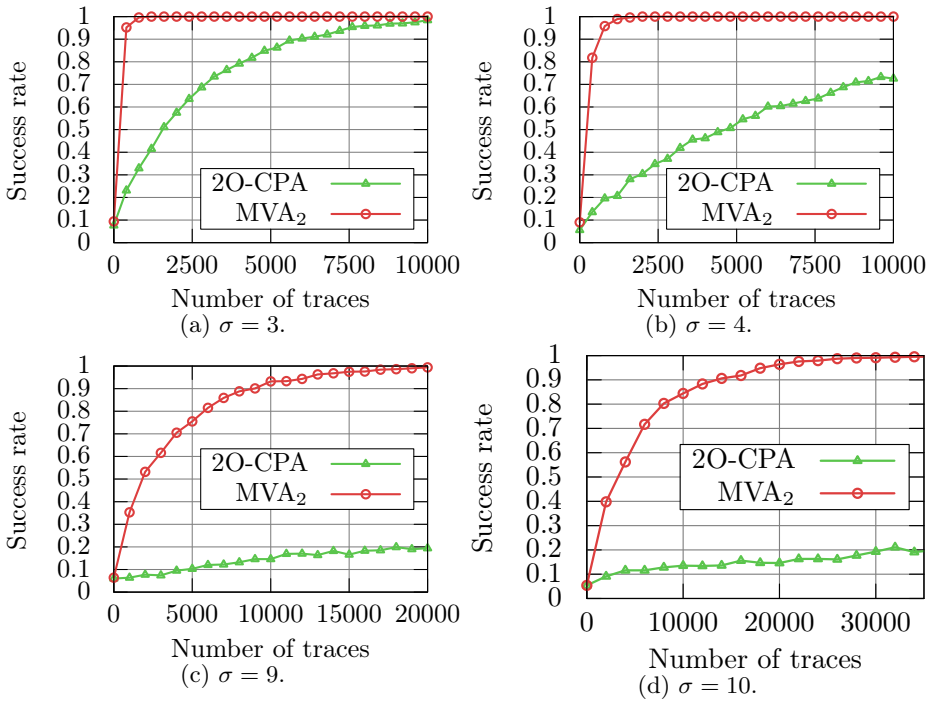

Fig. 2: Comparison of success rate for classical bi-variate attack and our multi-variate attack

The speed of the attack can be summarized as the number of traces to reach 80% of success rate. We represent the number of traces to reach 80% of success rate for bivariate and multivariate attacks. It can be seen that the number of traces is indeed increasing exponentially, although not at the same rate. We represent in Fig. 3 the number of traces to recover the key (with success probability of 80%) between the classical bi-variate attack and our optimal distinguisher (termed "multivariate attack"). Again, we see that the difference increases with the noise level (quantified by the noise standard deviation σ). Moreover, we see here that this difference increases faster than linearly with σ. Therefore, we really insist that the security of a masking scheme cannot only be considered by considering the number of shares, but also the dimensionality of the leakages.

We clearly see that our method allows to build really powerful attacks (compared with naïve state-of-the-art attack for which dimensionality = order). The combination of multiple leaking points makes the computation of the attack non-trivial, but the result is that, with equal amount

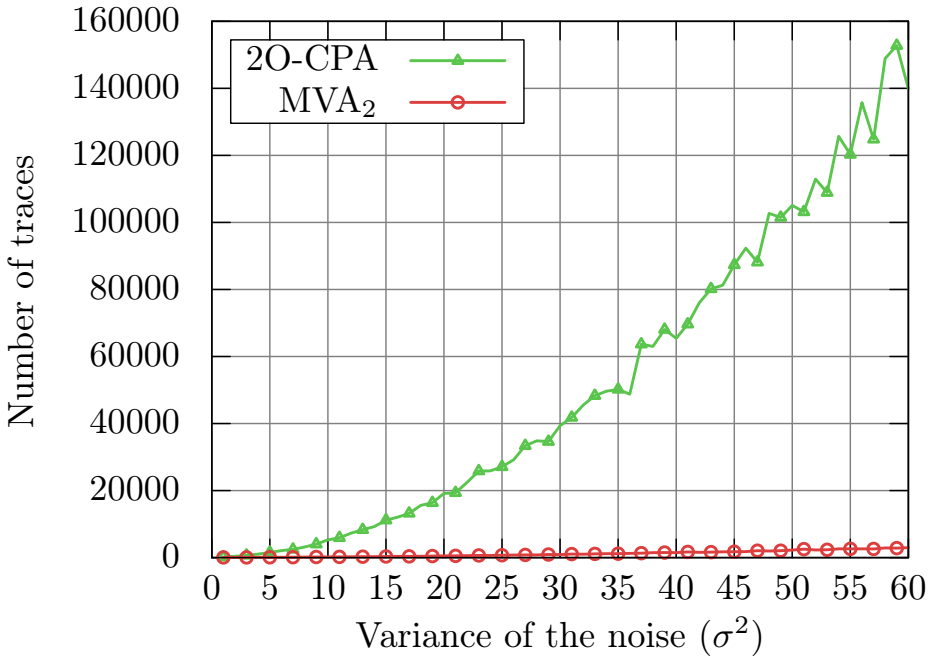

Fig. 3: Comparison between the number of traces to recover the key (with success probability of 80%) between the classical bi-variate attack and our multivariate attack (MVA_2, meaning that it is rounded at degree 2)

of traces, the multivariate attack (and actually not the best one, simply an approximation with a Taylor expansion) surpasses the schoolbook attack by orders of magnitudes. However, we underline that the goal of our study is not to design stronger attacks. Actually, we aim at providing after compilation the clearest possible image of the real security-level of the produced code. Therefore, the intent is to help the developer decide whether his implementation is secure enough vis-à-vis a security objective. In this respect, we had to devise a setup in which the strongest possible concrete attack is deployed.

The method of Taylor expansion is akin the soft analytical side-channel analysis. Actually, in the case of low-noise, such method is realistic, whereas our method is not. The table 1 compares the state-of-art-art approaches in terms of attack depending on the noise level σ (for a normalized leakage).

Table 1: Optimal attack methods in highly multivariate contexts

No noise	**Low noise**	**High noise**
Soft analytical [32]	SAT- or Pseudo-Boolean-solvers [25]	Taylor expansion [15]

The impact of the Taylor expansion order is negligible in our case-study, as illustrated in Fig. 4, which compares MVA rounded at order 2nd and 3rd. The difference is very limited, and in practice, the MVA_2 actually performs a little better than the MVA_3. Therefore, in order to save computation time, we recommend to apply the software verification scheme (Fig. 1) only at minimum order. The exact number of terms (which quantifies the complexity of the attack) is:

- MVA_2: 595 terms,
- MVA_3: 49011 terms.

In addition, the MVA_3 attack is more complex to perform, since the terms depend on σ, hence for the two values of noise in Fig. 4, two different distinguisher values shall be computed.

3.5 Limitations

Our tool checks the IR hence is unaware of registers and memory allocation. In particular, low-order leakage might arise owing to resources reuse, which, obviously, our tool cannot predict.

Besides, our method, like former methods [2,3], remains tailored for perfect sharings [11], but not custom maskings, e.g., first order masking schemes where masks are reused or so-called low-entropy masking schemes, e.g., masking schemes where the masks are not uniformly distributed [9,24]. For instance, the code segment in Appendix A.2 is perfectly masked at order one but cannot be analyzed by our method (which uses same heuristics as [2]).

4 Conclusion and perspectives

In this article, we tackled the issue of analyzing the security level of a software cryptographic code, during its compilation. For the analysis to be complete, we leverage both a vulnerability analysis (already put forward by Barthe et al. at EUROCRYPT 2015) and an exploitability analysis (based on a simulation of the optimal attack, computed within the compiler and tailored for the inputted code).

The first phase (vulnerability analysis) allows to verify that there is no flaw in the source code down to the optimized IR code. The second phase (exploitability analysis) investigates in which respect the implementation is robust in a given context (characterized by a noise level).

The two phases provide a clear view of the suitability of the compiled code for its execution in an operational environment.

As a byproduct of this work, we show that it is possible to automate crafted distinguishers which perform significantly better than simple theoretical attacks (e.g., d-valued d-th order CPA). The improvement results from the signal-to-noise increase by aggregating multiple high-order univariate analyses. The resulting multivariate high-order attacks can be extremely efficient. The reason is that there is an exponential way to combine leakages, and that this exponential advantage is comparable to the exponential complexity increase of the sharing order. We show concrete examples where security metrics (number of shares d) do not trivially reflect the practical security level in terms of attacks.

As a perspective, we intend to optimize the attacks (third column of Fig. 1). Indeed, using parallelism (vectorization) and precomputations, those could be drastically speeded-up. Moreover, we also intend to study the vulnerability and exploitability of attacks not only at the IR level, but also directly on assembly code. Eventually, it could be beneficial to extend the methodology (today geared from Barthe et al. method) to masking schemes which are not full-entropy or that reuse masks.

Acknowledgments

This work has been partly financed via TeamPlay, a project from European Union's Horizon20202 research and innovation program, under grand agreement N° 779882 (`https://teamplay-h2020.eu/`).

References

1. Josep Balasch, Benedikt Gierlichs, Oscar Reparaz, and Ingrid Verbauwhede. DPA, Bitslicing and Masking at 1 GHz. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, volume 9293 of *Lecture Notes in Computer Science*, pages 599–619. Springer, 2015.
2. Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified Proofs of Higher-Order Masking. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 457–485. Springer, 2015.
3. Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. Parallel Implementations of Masking Schemes and the Bounded Moment Leakage Model. In *Advances in Cryptology - EUROCRYPT 2017, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*, pages 535–566, 2017.
4. Alberto Battistello, Jean-Sébastien Coron, Emmanuel Prouff, and Rina Zeitoun. Horizontal Side-Channel Attacks and Countermeasures on the ISW Masking Scheme. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, volume 9813 of *Lecture Notes in Computer Science*, pages 23–39. Springer, 2016.
5. Ali Galip Bayrak, Francesco Regazzoni, David Novo, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. Automatic Application of Power Analysis Countermeasures. *IEEE Trans. Computers*, 64(2):329–341, 2015.
6. Ali Galip Bayrak, Francesco Regazzoni, David Novo, and Paolo Ienne. Sleuth: Automated Verification of Software Power Analysis Countermeasures. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings*, volume 8086 of *Lecture Notes in Computer Science*, pages 293–310. Springer, 2013.
7. Pierre Belgarric, Shivam Bhasin, Nicolas Bruneau, Jean-Luc Danger, Nicolas Debande, Sylvain Guilley, Annelie Heuser, Zakaria Najm, and Olivier Rioul. Time-Frequency Analysis for Second-Order Attacks. In Aurélien Francillon and Pankaj Rohatgi, editors, *CARDIS*, volume 8419 of *LNCS*, pages 108–122. Springer, 2013.
8. Inès Ben El Ouahma, Quentin Meunier, Karine Heydemann, and Emmanuelle Encrenaz. Side-Channel Robustness Analysis of Masked Assembly Codes using a Symbolic Approach. *Journal of Cryptographic Engineering*, pages 1–12, March 16 2019. DOI: 10.1007/s13389-019-00205-7.

9. Shivam Bhasin, Jean-Luc Danger, Sylvain Guilley, and Zakaria Najm. A low-entropy first-degree secure provable masking scheme for resource-constrained devices. In *Proceedings of the Workshop on Embedded Systems Security, WESS 2013, Montreal, Quebec, Canada, September 29 - October 4, 2013*, pages 7:1–7:10. ACM, 2013.
10. Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi. Cryptanalysis of a White Box AES Implementation. In *Selected Areas in Cryptography*, pages 227–240, 2004.
11. Johannes Blömer, Jorge Guajardo, and Volker Krummel. Provably Secure Masking of AES. In Helena Handschuh and M. Anwar Hasan, editors, *Selected Areas in Cryptography*, volume 3357 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2004.
12. Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In *CHES*, volume 4727 of *LNCS*, pages 450–466. Springer, September 10-13 2007. Vienna, Austria.
13. Nicolas Bruneau, Jean-Luc Danger, Sylvain Guilley, Annelie Heuser, and Yannick Teglia. Boosting Higher-Order Correlation Attacks by Dimensionality Reduction. In Rajat Subhra Chakraborty, Vashek Matyas, and Patrick Schaumont, editors, *Security, Privacy, and Applied Cryptography Engineering - 4th International Conference, SPACE 2014, Pune, India, October 18-22, 2014. Proceedings*, volume 8804 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2014.
14. Nicolas Bruneau, Sylvain Guilley, Annelie Heuser, and Olivier Rioul. Masks Will Fall Off – Higher-Order Optimal Distinguishers. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, volume 8874 of *Lecture Notes in Computer Science*, pages 344–365. Springer, 2014.
15. Nicolas Bruneau, Sylvain Guilley, Annelie Heuser, Olivier Rioul, François-Xavier Standaert, and Yannick Teglia. Taylor Expansion of Maximum Likelihood Attacks for Masked and Shuffled Implementations. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 573–601, 2016.
16. Nicolas Bruneau, Sylvain Guilley, Zakaria Najm, and Yannick Teglia. Multivariate High-Order Attacks of Shuffled Tables Recomputation. *Journal of Cryptology*, 31(2):351–393, Apr 2018.
17. Sébastien Carré, Adrien Facon, Sylvain Guilley, Sofiane Takarabt, Alexander Schaub, and Youssef Souissi. Cache-timing attack detection and prevention - application to crypto libs and PQC. In Ilia Polian and Marc Stöttinger, editors, *Constructive Side-Channel Analysis and Secure Design - 10th International Workshop, COSADE 2019, Darmstadt, Germany, April 3-5, 2019, Proceedings*, volume 11421 of *Lecture Notes in Computer Science*, pages 13–21. Springer, 2019.
18. Jean-Sébastien Coron, Emmanuel Prouff, and Matthieu Rivain. Side Channel Cryptanalysis of a Higher Order Masking Scheme. In Pascal Paillier and Ingrid Verbauwhede, editors, *CHES*, volume 4727 of *LNCS*, pages 28–44. Springer, 2007.
19. Jean-Luc Danger, Youssef El Housni, Adrien Facon, Cheikh T. Gueye, Sylvain Guilley, Sylvie Herbel, Ousmane Ndiaye, Edoardo Persichetti, and Alexander Schaub. On the Performance and Security of Multiplication in $GF(2^N)$. *Cryptography*, 2(3):25, 2018.

20. Hassan Eldib, Chao Wang, and Patrick Schaumont. Formal verification of software countermeasures against side-channel attacks. *ACM Trans. Softw. Eng. Methodol.*, 24(2):11:1–11:24, December 2014.

21. ETSI / TC CYBER. Security techniques for protecting software in a white box model, October 2018. ETSI TR 103 642 V1.1.1.

22. Yuval Ishai, Amit Sahai, and David Wagner. Private Circuits: Securing Hardware against Probing Attacks. In *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, August 17–21 2003. Santa Barbara, California, USA.

23. ISO/IEC JTC 1/SC 27/WG 3. ISO/IEC CD 20085-1:2017 (E). Information technology - Security techniques — Test tool requirements and test tool calibration methods for use in testing non-invasive attack mitigation techniques in cryptographic modules — Part 1: Test tools and techniques, January 25 2017.

24. Maxime Nassar, Youssef Souissi, Sylvain Guilley, and Jean-Luc Danger. RSM: a Small and Fast Countermeasure for AES, Secure against First- and Second-order Zero-Offset SCAs. In *DATE*, pages 1173–1178. IEEE Computer Society, March 12-16 2012. Dresden, Germany. (TRACK A: "Application Design", TOPIC A5: "Secure Systems").

25. Yossef Oren, Ofir Weisse, and Avishai Wool. A new framework for constraint-based probabilistic template side channel attacks. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 17–34. Springer, 2014.

26. Pablo Rauzy, Sylvain Guilley, and Zakaria Najm. Formally proved security of assembly code against power analysis - A case study on balanced logic. *J. Cryptographic Engineering*, 6(3):201–216, 2016.

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

28. Debapriya Basu Roy, Shivam Bhasin, Sylvain Guilley, Jean-Luc Danger, and Debdeep Mukhopadhyay. From theory to practice of private circuit: A cautionary note. In *33rd IEEE International Conference on Computer Design, ICCD 2015, New York City, NY, USA, October 18-21, 2015*, pages 296–303. IEEE Computer Society, 2015.

29. Kai Schramm and Christof Paar. Higher Order Masking of the AES. In David Pointcheval, editor, *CT-RSA*, volume 3860 of *LNCS*, pages 208–225. Springer, 2006.

30. Michael Tunstall, Carolyn Whitnall, and Elisabeth Oswald. Masking Tables – An Underestimated Security Risk. In Shiho Moriai, editor, *FSE*, volume 8424 of *Lecture Notes in Computer Science*, pages 425–444. Springer, 2013.

31. University of Sydney (Australia). Magma Computational Algebra System. `http://magma.maths.usyd.edu.au/magma/`, Accessed on 2014-08-22.

32. Nicolas Veyrat-Charvillon, Benoît Gérard, and François-Xavier Standaert. Soft analytical side-channel attacks. In *Advances in Cryptology - ASIACRYPT 2014 Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*, pages 282–296, 2014.

A Example of input codes for analysis

A.1 Codes which can be analyzed in our framework

Two examples of codes which can be analyzed are provided here-after in Listing 1.1. The selection between the two codes is achieved by defining macro `SBOX_TYPE` to either `cube` or `present` at line 117.

```
1  /*
2   * Regarding the refresh algorithm, there is no need at order 1, but beware of higher-order!
3   * https://eprint.iacr.org/2015/359.pdf
4  */
5
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <stdint.h>
9
10 static unsigned int gnr = 0;
11
12 /* This function must be optimized, otherwise it will include a test! */
13 uint8_t multx(uint8_t x)
14 {
15    uint8_t y = (x & 0x08) ? ((0x0f & (x << 1)) ^ 0x03) : 0x0f & (x << 1);
16    return y;
17 }
18
19 uint8_t mult(uint8_t x, uint8_t y)
20 {
21 #if 0 // Not to use with SAW plugin of LLVM
22    uint8_t z = 0;
23    uint8_t b = 0x8;
24    for(int i=3; i>=0; i--)
25    {
26       z = multx(z);
27       z ^= x & (-!!(y&b)); // Constant-time multiplication
28       b >>= 1;
29    }
30    return z;
31 #else
32    return x*y; // To simplify the analysis on an abstracted code
33 #endif
34 }
35
36 uint8_t __attribute__((optnone)) rnd(uint8_t r[])
37 {
38    return r[gnr++];
39 }
40
41 void refresh_masks(uint8_t x[2], uint8_t r[])
42 {
43    uint8_t new_r = rnd(r);
44    x[0] ^= new_r;
45    x[1] ^= new_r;
46 }
47
48 void __attribute__((optnone)) secmult(uint8_t x[2], uint8_t y[2], uint8_t z[2], uint8_t r[])
49 {
50 #if 0 // Security bug: the new_r self-demasks [To use for testing]
51    uint8_t new_r = rnd(r);
52    z[0] = (mult(x[0],y[0]) ^ new_r) ^ (mult(x[0],y[1]) ^ new_r) ^ mult(x[1],y[0]);
53    z[1] = mult(x[1],y[1]);
54 #else
55    uint8_t r01 = rnd(r);
56    uint8_t r10 = (r01 ^ mult(x[0],y[1]));
57    r10 ^= mult(x[1],y[0]);
58
59    z[0] = mult(x[0],y[0]);
60    z[0] ^= r01;
61    z[1] = mult(x[1],y[1]);
62    z[1] ^= r10;
63 #endif
64 /* Another option from \cite{2003-11949} would be:
65    c_1 = r                                                    (1)
```

```
66      c_2 = (((a_1 b_1 + r) + a_1 b_2 ) + a_2 b_1 ) + a_2 b_2 . (2)
67  */
68  }
69  void secsquare(uint8_t x[2], uint8_t z[2], uint8_t r[])
70  {
71      z[0] = mult(x[0],x[0]);
72      z[1] = mult(x[1],x[1]);
73  }
74
75  const uint8_t a[15] = { 12, 0, 7, 7, 14, 10, 12, 4, 7, 9, 9, 14, 12, 13, 13 };
76
77  uint8_t l1(uint8_t x)
78  {
79      uint8_t x_2 = mult(x,x);
80      uint8_t x_4 = mult(x_2,x_2);
81      uint8_t x_8 = mult(x_4,x_4);
82      return mult(a[1],x) ^ mult(a[2],x_2) ^ mult(a[4],x_4) ^ mult(a[8],x_8);
83  }
84
85  uint8_t l3(uint8_t x)
86  {
87      uint8_t x_2 = mult(x,x);
88      uint8_t x_4 = mult(x_2,x_2);
89      uint8_t x_8 = mult(x_4,x_4);
90      return mult(a[3],x) ^ mult(a[6],x_2) ^ mult(a[12],x_4) ^ mult(a[9],x_8);
91  }
92
93  uint8_t l5(uint8_t x)
94  {
95      uint8_t x_2 = mult(x,x);
96      return mult(a[5],x) ^ mult(a[10],x_2);
97  }
98
99  uint8_t l7(uint8_t x)
100 {
101     uint8_t x_2 = mult(x,x);
102     uint8_t x_4 = mult(x_2,x_2);
103     uint8_t x_8 = mult(x_4,x_4);
104     return mult(a[7],x) ^ mult(a[14],x_2) ^ mult(a[13],x_4) ^ mult(a[11],x_8);
105 }
106 /*
107  * The function which be symbolically interpreted by SAW
108  * All the shares are named 'm', and the masks 'r'
109 */
110 //              DATA IN       DATA OUT      RANDOMS
111 void sbox(      uint8_t m[2], uint8_t y[2], uint8_t r[])
112 {
113     uint8_t x_2[2];
114     uint8_t x_3[2];
115     uint8_t x_5[2];
116     uint8_t x_7[2];
117 #if 1 // DEBUG, SBOX_TYPE=cube
118     secsquare( m, x_2, r );
119     refresh_masks(m,r);
120     secmult( m, x_2, y, r );
121 #else // SBOX_TYPE=present
122
123     x_2[0] = mult(m[0],m[0]);
124     x_2[1] = mult(m[1],m[1]);
125     secmult( m, m, x_2, r );
126     secsquare( m, x_2, r );
127     refresh_masks(m,r);
128     refresh_masks(x_2,r);
129     secmult( m, x_2, y, r );
130
131     x_3[0] = mult(x_2[0],m[0]);
132     x_3[1] = mult(x_2[1],m[1]);
133     y[0] = x_2[0];
134     y[1] = x_2[1];
135     secmult( m, x_2, x_3, r );
136     refresh_masks(m,r);
137     refresh_masks(x_2,r);
138     refresh_masks(x_3,r);
139
140     x_5[0] = mult(x_3[0],x_2[0]);
141     x_5[1] = mult(x_3[1],x_2[1]);
142     secmult( x_2, x_3, x_5, r );
```

```
143     refresh_masks(x_2,r);
144     refresh_masks(x_3,r);
145
146     refresh_masks(x_5,r);
147     x_7[0] = mult(x_5[0],x_2[0]);
148     x_7[1] = mult(x_5[1],x_2[1]);
149
150     secmult( x_2, x_5, x_7, r );
151     refresh_masks(x_2,r);
152     refresh_masks(x_5,r);
153     refresh_masks(x_7,r);
154     y[0] = a[0] ^ l1(m[0]) ^ l3(x_3[0]) ^ l5(x_5[0]) ^ l7(x_7[0]);
155     y[1] =        l1(m[1]) ^ l3(x_3[1]) ^ l5(x_5[1]) ^ l7(x_7[1]);
156 #endif
157 }
158
159 int main()
160 {
161     uint8_t r[100];
162     for(int i=0; i<100; i++){
163         r[i] = rand();
164     }
165
166     uint8_t x[2];
167     uint8_t y[2];
168
169     const uint8_t sbox_ref[16] = { 12, 5, 6, 11, 9, 0, 10, 13, 3, 14, 15, 8, 4, 7, 1, 2 };
170     for( uint8_t i=0; i<16; ++i )
171     {
172         uint8_t mask = rand();
173         x[0] = i^mask;
174         x[1] =   mask;
175         sbox(x,y,r);
176         printf("%d␣-->␣%d␣(ref␣=␣%d)\n", x[0]^x[1], y[0]^y[1], sbox_ref[i]);
177     }
178     return 0;
179 }
```

Listing 1.1: C code representing polynomial computations

A.2 Code which cannot be analyzed

In this section, we present one example of code which cannot be analyzed (*automatically*) since simplifications as per Barthe [2] do not apply. Indeed, the masks are not used as in ISW [22]:

- in ISW: masks are added (XORed) and subsequently subtracted (XORed), whereas
- in Alg. 1.2: the masks are involved in computation as selection variable in a choice.

The listing 1.2 presents both a straightforward multiplexor and a multiplexor protected at first-order.

```
1  /* Unprotected function, computing a selection (= multiplexor) */
2  uint8_t MUX( uint8_t a, uint8_t b, uint8_t c )
3  {
4      // return c?b:a; // At bit-level
5      return (c&b)|(~c&a); // At word-level
6  }
7
8  /* Function whose 1st-order security can be demonstrated */
9  uint8_t MUX_masked(uint8_t am, uint8_t bm, uint8_t cm, uint8_t r[] /* m, m2 */ )
10 {
11     uint8_t m  = r[0];
```

```
12      uint8_t m2 = r[1];
13      return (((am & ~m2) ^ (bm &  m2)) & (~(cm ^ (m^m2)))) ^ \
14             (((am &  m2) ^ (bm & ~m2)) &   (cm ^ (m^m2)));
15  }
```

Listing 1.2: Function which conditionally returns one of the two arguments, protected against side-channel attacks at order $d = 1$

B Multi-variate attack at degrees two and three

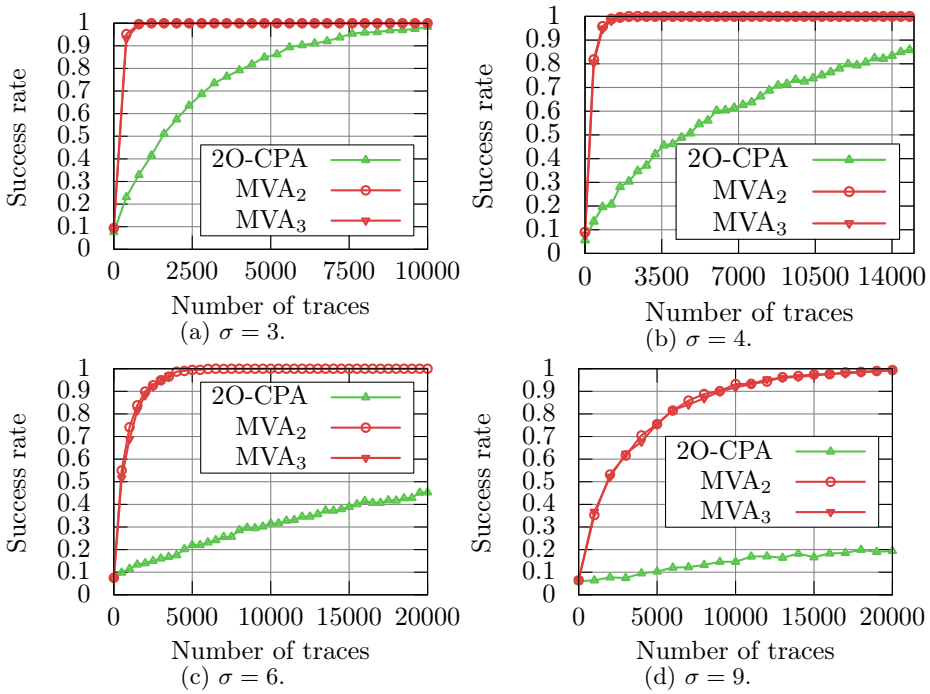

Fig. 4: Comparison of success rate for classical bi-variate attack and our multi-variate attack at degrees two and three