PROSECCO: Formally-Proven Secure Compiled Code

Nicolas Belleville[1], Damien Couroussé[1], Emmanuelle Encrenaz[2], Karine Heydemann[2] and Quentin Meunier[2]

[1] *Univ. Grenoble Alpes, CEA, List, F-38000 Grenoble, France*
[2] *Sorbonne Université, CNRS, LIP6, F-75005, Paris, France*

Abstract

The application and the verification of countermeasures against physical attacks still remain long, error-prone and expertise-demanding tasks. We propose a toolchain to help the expert in these tasks. Our toolchain is composed of two components: a compiler that automatically applies a set of countermeasures, and a formal verification tool that automatically verifies binary code for various leakage models and fault models. We describe different scenarios of usage of our toolchain, and then illustrate the flexibility of our toolchain in one of them.

Keywords

side-channel attack, fault injection attack, masking, hiding, fault tolerance, control flow integrity, formal verification, compilation

1. Introduction

Cybersecurity is a growing concern for embedded systems and connected objects. Such objects can be used as part of large-scale attacks. Among the possible attacks against such systems, side-channel attacks and fault injection attacks stand out from their ability to recover secret information manipulated by the device (such as cryptographic keys) and to modify the device behaviour [1, 2, 3, 4].

Securing devices against these attacks at the software level is costly as it requires highly technical and time-consuming tasks. In particular, some bottlenecks are due to the compilation flow and the fact that countermeasures applied on the source code can be optimised out or degraded by compilers [5].

A first bottleneck is related to the countermeasure deployment. In order to avoid any nefarious code transformations carried out by the compiler, the security expert may choose to apply countermeasures downstream to the compiler, i.e., at the assembly level or at the binary level. At the source level, an option is to use code tricks (such as volatile variables or specific code structures that the compiler is a priori not able to optimise

C&ESAR 2021
EMAIL: nicolas.belleville@cea.fr (N. Belleville); damien.courousse@cea.fr (D. Couroussé); emmanuelle.encrenaz@lip6.fr (E. Encrenaz); karine.heydemann@lip6.fr (K. Heydemann); quentin.meunier@lip6.fr (Q. Meunier)

out) to circumvent code optimisations. However, in all cases, such practices are fragile, error-prone, and require a significant amount of manpower.

A second bottleneck is related to security assessment. Traditionally, security evaluation is carried out experimentally, by replaying state-of-the art attacks or by measuring security-related metrics. However, a separate but complementary process can help reduce the complexity and time required for the security evaluation: formally verifying security properties against a security model. Formal methods are a powerful mean to build strong confidence of the security correctness of a component against a security model. Such a verification of security properties must take place at the binary level due to the compiler potential negative effects and to the necessity of using low-level models that better reflect the physical effects of the attacks than higher-level ones.

In workflows for the production of secured software, leveraging automation is a way to lower the overall production costs, and time to market. Hence, there is a need to automate the application of countermeasures, and a need to automate the verification of security properties at the binary level, while keeping the flexibility required for the expert.

In this context, we propose a toolchain for the compilation of several software countermeasures against physical attacks, and for the verification, at the binary level, of the associated security properties. The countermeasures supported by our compiler are configurable, can be selectively applied on program parts, and can be combined with other countermeasures manually applied by an expert. The expert can choose the model to use for the verification steps, as the verification tools support several leakage models and several fault models.

2. Background

2.1. Physical attacks

There are two classes of physical attacks: side-channel attacks and fault injection attacks.

Side-channel attacks consist in exploiting measurements of a physical quantity, such as power [1], electromagnetic emission [6], acoustic noise [7], etc., in order to infer information about the computations done in the hardware. Such measurements can be used to reverse-engineer a program [8], or to find secret data such as cryptographic keys [9].

Fault injection attacks consist in altering the behaviour of the chip by means of physical perturbations. For instance, an attacker can perform voltage glitches [10], clock glitches [11], electromagnetic disturbance [12], or laser fault injection [4]. The fault aims at modifying the program's execution. The attacker can obtain various effects from a fault injection attack, like authentication with a false password, or getting knowledge about some secret data through the program's output.

2.2. Countermeasures

Countermeasures against side-channel attacks are mainly divided between hiding and masking principles. Hiding consists in lowering the signal-to-noise ratio to make the attacker's measurements harder to exploit. Typical examples of hiding countermeasures include loop shuffling [13], random delays [14], code morphing [15] and code polymorphism [16]. Masking consists in breaking the correlation between the measurements and the sensitive data by splitting the sensitive variables into several variables called *shares*, each share being statistically independent of the sensitive data [2]. Splitting the variable into shares can be achieved using various operators, such as the exclusive or [17], finite field multiplication [18], arithmetic addition [19], etc.

Countermeasures against fault injection are of three types: fault tolerance, fault detection, and infection. They usually imply some form of spatial or temporal redundancy. Fault tolerance consists in modifying the code or data so that a fault has no effect on the final result of the program [20]. Fault detection consists in modifying the code or data so that a fault is detected, which enables to take appropriate actions afterwards like halting the system or self destruction [21, 22]. Last, infective countermeasures consist in diffusing the effect of the fault to prevent the attacker from exploiting the fault effect [23].

2.3. Security evaluation

In order to assess the security of a program on a device, an evaluator can use several methods.

First, they can try to attack the device [24, 25]. This method has the advantage of giving an estimate of the time and processing power necessary for the attack. However, it is tied to a particular attack and experimental setup.

In the case of side-channel attacks, an evaluator can also use empirical measures related to the attack difficulty. For instance, in the case of side-channel analysis, they can compute a SNR, or a t-test [26, 27]. This method has the advantage of being very sensitive and in the case of t-test, it can be independent of a leakage model. Though, it may not fully represent the difficulty of an attack, as attackers can perform preprocessing on traces, higher-order attacks, or more evolved attacks such as template attacks or machine-learning attacks. It is also dependent on the measurement setup.

Finally, an evaluator can use formal methods to analyse the binary [28, 29, 30]. The formal analysis tool relies either on a leakage model in the case of side-channel attacks, or on a fault model in the case of fault analysis. A leakage model defines how the values leak, for instance if there are value-based leakages or transition based leakages. It can also be more precise, defining a leakage as a function of the values, like its Hamming weight. A fault model defines what effects a fault can have on a program. For instance, instruction-skip, data modification, or instruction modification are some frequently used fault models. The use of formal methods for evaluation has the advantage of not being dependent on the experimental setup. The use of a model can lead to limitations though, as the result may not represent the full possibilities of the attacker depending on the model accuracy.

Ideally, the evaluator can combine several techniques to evaluate the security of the device.

3. Approach

3.1. Overview

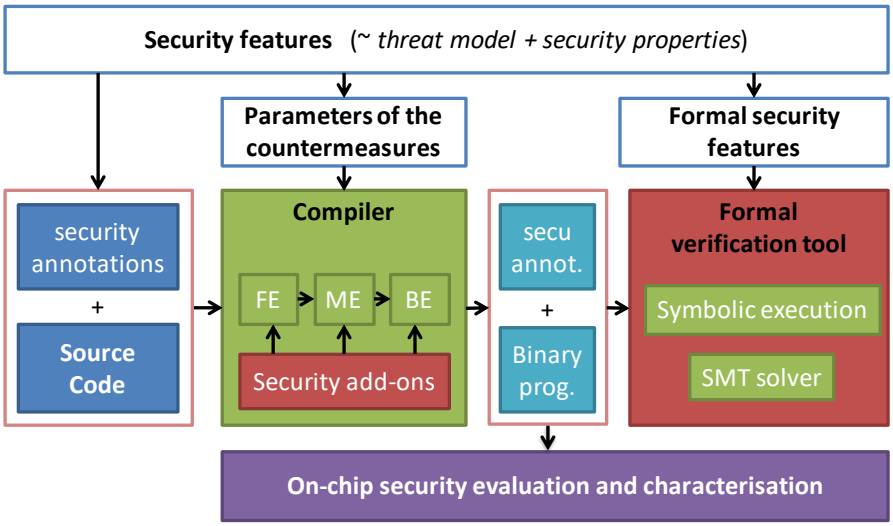

Figure 1: Design flow for securing software components against physical attacks

We present a methodology supported by tools to automatically secure components against physical attacks and/or verify secured binaries. Figure 1 shows an overview of this methodology. Our approach is constructed around two components: a compiler, and a formal verification tool. The compiler is in charge of applying various software countermeasures on a program annotated by the programmer, or driven by specific compiler options. The programmer specifies the parts of the program to secure, the countermeasures to use, and their parameters. After the compilation, the formal verification tool checks that security properties are correctly implemented at the binary level.

The following sections will detail the different possibilities offered by our compiler (section 3.2) and by our verifier (section 3.3).

3.2. Automated application of countermeasures during compilation

Our compiler, named Cogito, is based on the LLVM compiler infrastructure. It supports several countermeasures: masking and code polymorphism against side-channel attacks; fault tolerance and execution integrity against fault injection attacks. In all cases, the user annotates the code or uses specific compiler options to specify the target functions to secure, as well as the configuration parameters of the countermeasures.

Masking countermeasure against side-channel attacks The compiler can apply a first-order Boolean masking countermeasure [31]. The user handles the initial separation of the secret into shares. Then, the original instructions that manipulate the secret data are replaced by sequences of instructions that manipulate the shares. For this purpose, the compiler uses well-known transformations such as the `secMult` algorithm [32] to secure `and` operations and to secure finite field multiplications.

Two key elements stand out in our application of a masking countermeasure. First, the usual approach for the masking of table lookups (e.g., accesses to S-Boxes) consists in regularly re-computing masked tables [33]. Ideally, the whole masked table should be re-computed after each access, leading to important performance overheads. Still, some side-channel attacks can exploit observations of the table re-computation [34]. In our approach, table lookups are transformed into masked evaluations of an interpolating polynomial [35], and the compiler applies several optimisations to reduce the performance overhead of the countermeasure [31]. Second, other state-of-the-art masking tools require the control-flow of the input program to be flattened (e.g., with loops unrolled) before the application of the countermeasure. Our compiler applies the countermeasure without modifying the control-flow structure of the secured program, which increases the range of applications of the masking countermeasure and helps reducing the code size of the secured binary.

Code polymorphism countermeasure against side-channel attacks The compiler can also apply code polymorphism, a hiding countermeasure [16]. The core idea is to regularly generate new versions of the secure code, called *polymorphic instances*, by means of runtime code generation driven by random data. All of these polymorphic instances are functionally equivalent, but differ in their implementation, such that each execution leads to a different (side-channel) observation, thus raising the bar for an attacker. Our implementation of code polymorphism leverages the use of specialised runtime code generators (SGPC): each program function targeted by the countermeasure is associated with a dedicated SGPC, automatically generated by the compiler. At runtime, SGPCs leverage one or several of the following code transformations driven by random data:

- instruction shuffling: independent instructions are emitted in a random order;
- register permutation: a random permutation is done for the general-purpose registers;
- instruction substitution: an instruction is replaced by a sequence of instructions that give the same result;
- insertion of noise instructions: a random number of randomly selected noise instructions is inserted. Noise instructions are of the same nature as the normal instructions of the program, but use dead registers, and are randomly interleaved with the other instructions of the program;
- insertion of dynamic noise sequences: a jump instruction is inserted, followed by a sequence of noise instructions. The jump offset is randomly computed such as the jump target falls anywhere within the sequence of noise instructions. This

mechanism enables to have part of the execution behaviour that is independent from the paths taken during the generation.

Instruction replication against fault injection attacks The compiler can apply a fault tolerance countermeasure that protects against faults resulting in instruction skips [20]. The working principle of the countermeasure consists in producing idempotent instructions and then replicating each idempotent instruction [36]. The compiler offers two main securing parameters: the number of instruction copies introduced during replication, each copy providing tolerance against one fault injection; the distance between each instruction copy, which protects against the case where a fault injection can skip several consecutive instructions. Interestingly, the compiler, leveraging many optimisations, can reduce execution time and code size overheads of the countermeasure as compared to an application of the countermeasure at the assembly level by an expert [20].

Execution integrity against fault injection attacks The compiler can apply an execution integrity countermeasure that aims at detecting all faults that modify the program counter (PC) register, or that modify branch conditions [37]. As such, this countermeasure can be considered as a fine-grained Control-Flow Integrity countermeasure: the countermeasure is able to detect integrity violations of the control flow of the protected section of the program, protecting jumps, direct conditional and unconditional branches, and direct function calls. In addition, the countermeasure can detect modifications of the PC or alterations of the machine instructions inside basic blocks, e.g., caused by instruction skips.

3.3. Automated verification of countermeasures at binary level

Our verification tool integrates two components dedicated to the robustness analysis in presence of fault attacks or side channel attacks. The inputs consist in the binary program, as well as information about the region to analyse and the threat model. A common front-end based on symbolic execution is able to build the necessary information for each component.

Masked software implementation verification The verification is made by the component named ARISTI. It implements a symbolic approach to analyse the distribution of the value of some symbolic expressions with respect to some user-specified secret variables [28].

To verify a masked implementation, the masks and secret variables manipulated by the implementation must be provided with the binary program. From an execution trace generated by the symbolic engine, ARISTI computes, for each instruction in the trace, the symbolic expression of its result, corresponding either to a value written into a general purpose register of the processor or in the memory.

The analysis of the distribution of a symbolic expression relies on distribution type inference using specifically designed rules. The distribution types are either constant, either uniform, either statistically (in-)dependent from the secrets or unknown. The

symbolic variables appearing in the expression as well as the root operation impact the distribution type inference rules that can be applied. The goal of the analysis is to infer, a distribution type as precise as possible. By decreasing order of precision, this can be:

1. a constant value (CST), a leaky expression (SNM), a uniform distribution (RUD);
2. a secret independent distribution (SID);
3. an unknow distribution type (UKD), meaning that the verification can not conclude.

Moreover, when the resulting distribution type is UKD, ARISTI can perform an enumerative analysis to check for the absence of leakage. ARISTI supports two leakage models: the value-based and the transition leakage models. In the former, the result of each instruction is analysed. In the latter, the analysed expressions are obtained by xoring the result of two expressions, e.g. the one representing the value written in a register with the one representing the value previously contained in the register.

Fault robustness verification The dedicated component named RobustB combines some static analyses and symbolic execution to build a SMT formulation of the possible execution paths of the region to analyse and of the possible execution paths in presence of a fault injection [29]. The threat model specifies the fault models that an attacker may induce among instruction skip, register corruption, bitset or bit reset on instruction encoding. Faults are transient.

The robustness analysis consists in verifying, using a SMT formulation, that no fault can induce a vulnerability. If the user-provided security property is the integrity of some registers and memory contents at the end of the region execution (e.g., the output of an authentication function), the verification is performed by equivalence-checking between each original execution path and their faulty counterparts. When the security property is a predicate (assert-like, e.g., the output must be the Boolean value false when the entered password is not the expected one), the verification is performed by checking that this property holds at the end of all possible faulty paths.

In case of vulnerabilities detection, RobustB outputs some security metrics, which help the user pinpoint the vulnerable instructions or compare different countermeasures.

4. Application of the code securing toolchain

We consider the following scenarios:

SC1. Application of a countermeasure with the compiler and formal and/or empirical security evaluation;

SC2. Manual application of a countermeasure and verification with the formal verification tool;

SC3. Combination of several countermeasures applied at different levels, either manually or using the compiler, and verification of the countermeasures with the formal verification tool.

Several examples of the use of our toolchain with scenarios SC1 and SC2 have already been published. The automated application of code polymorphism has been demonstrated on a wide range of programs in [16]. The automated application of first order boolean masking has been applied and verified on a full AES in [31]. The automated application of instruction duplication was presented and demonstrated on the AES in [20], and verified for various `VerifyPin` implementations in [29] (scenario SC1). In [28], the masking countermeasure has also been verified on programs where the countermeasure was manually applied (scenario SC2).

Here, we propose to focus on the use of our toolchain with the scenario SC3, to illustrate the flexibility of the toolchain: a programmer wants to secure a program with two countermeasures: a masking countermeasure, applied by an expert at the source level, and a tolerance countermeasure against instruction-skip attacks automatically applied by the compiler. The expert also wants to check that the masking countermeasure is correctly applied at the binary level: i.e., that neither the performance optimisations nor the addition of the fault tolerance countermeasures did break the side-channel countermeasure, and that the resulting binary is resistant to instruction-skip attacks.

To illustrate our scenario, we consider the application of the instruction replication countermeasure on a `SecMult` function. We implement the function in C. In order to add the instruction replication countermeasure, we declare the function as to be secured and we set the countermeasure parameters (number of replications, distance between replicas) through the command-line arguments of Cogito. We then generate three different binaries:

1. one compiled with GCC (called `SecMult-gcc` later on);
2. one compiled with Cogito, transforming the code so that it uses only idempotent instructions (called `SecMult-idem` later on) but without replication of the instructions;
3. one compiled with Cogito, with the complete application of the instruction duplication countermeasure (called `SecMult-dup` later on).

We analyse all the three binaries with ARISTI, in the value-based leakage model and in the transition-based leakage model. To run the analysis, we declare the targetted function as well as the input secret variables. The analysis results are shown in Table 1. Leakages (indicated as LEAK in the table) can be found either from SNM, or from an enumerative analysis after a variable is marked as UKD. The analyses do not find any value-based leakage in these implementations. Though, transition-based leakages are found in all of them. The differences in terms of UKD and leakage between `SecMult-gcc` and `SecMult-idem` may be due to the difference between instruction selection and register allocation between GCC and LLVM. `SecMult-idem` and `SecMult-dup` have the same number of leakages, which means the application of the instruction duplication countermeasure did not introduce new leakages. The security expert could then try fixing the leakages found, either at the source code level, or by modifying directly the binary produced by our compiler.

We also run an analysis with RobustB to check that the instruction replication

Table 1
Analysis results with ARISTI for different `SecMult` binaries.

Binary	`Secmult-gcc`	`Secmult-idem`	`Secmult-dup`
Compiler used	GCC	Cogito (LLVM-based)	Cogito (LLVM-based)
Instruction duplication	no	no	yes
Analysis results with value-based leakage model:			
#RUD	79	13	17
#ISD	140	86	164
#CST	46	112	216
#SNM	0	0	0
#UKD	48	66	132
#LEAK	0	0	0
Analysis results with transition-based leakage model:			
#RUD	34	20	22
#ISD	150	93	92
#CST	34	58	311
#SNM	1	0	0
#UKD	73	98	98
#LEAK	1	7	7

countermeasure is correctly applied: no successful attack is found using an instruction-skip fault model.

This example shows how our toolchain could be used in a flexible way by an expert:

- countermeasures can be applied manually, automatically by the compiler, or both,
- formal verification tool can be used on the resulting binaries even if the compiler used is not the one from our toolchain.

5. Discussion

We illustrated the interest of our toolchain along with its countermeasures applied at compilation and security properties verified on the binaries. Many research questions still remain to be investigated for reaching better performance, ease-of-use, and security, both for the compilation of countermeasures and the verification of security properties.

First, the propagation of security information in the toolchain represents a crucial challenge. Promising results have been shown recently on this matter [5]. A correct propagation of information and handling of this information by the toolchain would:

- facilitate the review of assembly code containing countermeasures, whether the countermeasures were added at source code level or by the compiler;
- help preserving the security properties throughout the compilation, avoiding the degradation of countermeasure by optimisation passes;

- enable linking issues reported by the binary-level formal analysis tool with the source code.

Then, keeping up with state-of-the-art attacks, countermeasures and security properties forms another fundamental challenge. The adaptation of the tools requires an engineering effort, and also frequently requires some research as the automated application of countermeasures or verification of new models raise new challenges. For example, adding a new countermeasure in a compiler requires careful considerations about where in the compiler the transformation should be done to benefit from the compiler optimisations, ease the support of several architectures, and have a suitable representation of the program for the countermeasure. Adding a new security property in a formal verification tool may require to redesign some algorithms, find new heuristics to make the formal verification fast enough for targeted programs. In addition, the verification of the security properties should be low-level enough to avoid issues with link-time optimisation which could also alter some countermeasures, which constrains the representation that formal verification tools have to work with.

The combination of countermeasure also raises interesting questions to investigate. While some countermeasures are independent, interaction between others remain poorly understood. For instance, combination of masking and polymorphism requires special care: polymorphism transformations make use of dead registers but dead registers containing shares of a secret must not be combined to avoid unmasking. Keeping CFI protection while adding noise instructions is also challenging. Instruction duplication on a masked implementation would also raise issues if user requires duplicate instructions to be far from each other: the scheduling of the duplicate instruction may introduce transition leakage.

Finally, the information that the compiler is able to recover may be incomplete in comparison to the knowledge of an expert. As a consequence, in the context of automated code securing as in the context of code optimization for performance, the resulting generated code may not be highly optimized. As an example, masking scheme can sometimes be efficiently implemented using the knowledge about the algebraic structure of an operation. However, the compiler does not have this information, and it has to decompose any unknown non linear operation into a set of operations that it knows how to mask, which can result in a suboptimal implementation. Nevertheless, in some cases, the compiler can take advantages of code optimisation, and generate secured code which is faster than code where the countermeasure was applied by hand as shown for fault tolerance by [20]. In addition, apart from automating and accelerating the securing process especially for large code basis, having an automated tool enables the user to choose between a larger set of countermeasures and countermeasures parameters, as the compiler can hide complexity to the user. For instance, application of a low level CFI scheme or of polynomial interpolation of SBoxes by hand would not be easy.

To conclude, we argue that having flexible automated tools that support some countermeasures and some verification models is already an asset to ease securing a large number of potential targets currently left unsecure, against a wide range of attacks.

6. Conclusion

In this paper, we present an approach supported by tools to help an expert to secure a device against side-channel attacks and fault injection attacks. The toolchain can be used as part of various workflows and for various security requirements; it supports several countermeasures and verification models, and the compiler and verification tool are kept separated to ease their integration in the expert workflow.

Acknowledgments

This work was partially funded by the French National Research Agency (ANR) as part of the project PROSECCO, unded by the program AAP-2015 under grant agreement ANR-15-CE39.

References

[1] P. Kocher, J. Jaffe, B. Jun, Differential power analysis, in: CRYPTO, 1999.
[2] S. Mangard, E. Oswald, T. Popp, Power analysis attacks: Revealing the secrets of smart cards, Springer, 2008.
[3] D. Boneh, R. A. DeMillo, R. J. Lipton, On the Importance of Checking Cryptographic Protocols for Faults, in: EUROCRYPT, 1997.
[4] J.-M. Dutertre, C. De, A. Sarafianos, N. Boher, B. Rouzeyre, M. Lisart, J. Damiens, P. Candelier, M.-L. Flottes, G. Di Natale, Laser attacks on integrated circuits: From CMOS to FD-SOI, in: DTIS, 2014.
[5] S. T. Vu, K. Heydemann, A. de Grandmaison, A. Cohen, Secure delivery of program properties through optimizing compilation, in: CC, 2020.
[6] K. Gandolfi, C. Mourtel, F. Olivier, Electromagnetic analysis: Concrete results, in: CHES, Springer, 251–261, 2001.
[7] D. Genkin, A. Shamir, E. Tromer, Acoustic Cryptanalysis, Journal of Cryptology 30 (2) (2017) 392–443.
[8] V. Cristiani, M. Lecomte, T. Hiscock, A bit-level approach to side channel based disassembling, in: CARDIS, 2019.
[9] E. Brier, C. Clavier, F. Olivier, Correlation power analysis with a leakage model, LNCS 3156 (2004) 16–29.
[10] R. B. Carpi, S. Picek, L. Batina, F. Menarini, D. Jakobovic, M. Golub, Glitch It If You Can: Parameter Search Strategies for Successful Fault Injection, in: Smart Card Research and Advanced Applications, LNCS, Springer, 236–252, 2013.
[11] M. Agoyan, J.-M. Dutertre, D. Naccache, B. Robisson, A. Tria, When clocks fail: On critical paths and clock faults, LNCS 6035 (2010) 182–193.
[12] J.-J. Quisquater, D. Samyde, ElectroMagnetic Analysis (EMA): Measures and Counter-measures for Smart Cards, in: Smart Card Programming and Security, LNCS, Springer, 200–210, 2001.

[13] M. Rivain, E. Prouff, J. Doget, Higher-Order Masking and Shuffling for Software Implementations of Block Ciphers, in: CHES, LNCS, Springer, 171–188, 2009.
[14] J.-S. Coron, I. Kizhvatov, LNCS 6225 (2010) 95–109.
[15] G. Agosta, A. Barenghi, G. Pelosi, in: DAC, 77–82, 2012.
[16] N. Belleville, D. Couroussé, K. Heydemann, H.-P. Charles, Automated software protection for the masses against side-channel attacks, in: TACO, ACM, 2018.
[17] L. Goubin, J. Patarin, DES and Differential Power Analysis The "Duplication" Method, in: CHES, Springer, 1999.
[18] L. Genelle, E. Prouff, M. Quisquater, Secure Multiplicative Masking of Power Functions, in: Applied Cryptography and Network Security, vol. 6123, Springer, 200–217, 2010.
[19] L. Goubin, A sound method for switching between boolean and arithmetic masking, in: International Workshop on Cryptographic Hardware and Embedded Systems, Springer, 3–15, 2001.
[20] T. Barry, D. Couroussé, B. Robisson, Compilation of a Countermeasure Against Instruction-Skip Fault Attacks, in: CS2, 2016.
[21] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, F. Regazzoni, Countermeasures against fault attacks on software implemented AES: effectiveness and cost, in: WESS, 1–10, 2010.
[22] J. Bringer, C. Carlet, H. Chabanne, S. Guilley, H. Maghrebi, Orthogonal Direct Sum Masking: A Smartcard Friendly Computation Paradigm in a Code, with Builtin Protection against Side-Channel and Fault Attacks, in: Information Security Theory and Practice. Securing the Internet of Things, vol. 8501, Springer, 40–56, 2014.
[23] P. Rauzy, S. Guilley, Countermeasures against high-order fault-injection attacks on CRT-RSA, in: FDTC, IEEE, 68–82, 2014.
[24] B. Colombier, A. Menu, J.-M. Dutertre, P.-A. Moëllic, J.-B. Rigaud, J.-L. Danger, Laser-induced single-bit faults in flash memory: Instructions corruption on a 32-bit microcontroller, in: HOST, 2019.
[25] L. Masure, N. Belleville, E. Cagli, M.-A. Cornélie, D. Couroussé, C. Dumas, L. Maingault, Deep Learning Side-Channel Analysis on Large-Scale Traces, in: ESORICS, 2020.
[26] T. Schneider, A. Moradi, Leakage assessment methodology, in: CHES, Springer, 495–513, 2015.
[27] F.-X. Standaert, How (not) to Use Welch's T-test in Side-Channel Security Evaluations, in: CARDIS, 2018.
[28] I. B. El Ouahma, Q. L. Meunier, K. Heydemann, E. Encrenaz, Side-channel robustness analysis of masked assembly codes using a symbolic approach, JCEN .
[29] J.-B. Bréjon, K. Heydemann, E. Encrenaz, Q. L. Meunier, S. T. Vu, Fault attack vulnerability assessment of binary code, in: CS2, 2019.
[30] G. Barthe, S. Belaïd, G. Cassiers, P.-A. Fouque, B. Grégoire, F.-X. Standaert, Maskverif: Automated verification of higher-order masking in presence of physical defaults, in: ESORICS, 2019.
[31] N. Belleville, D. Couroussé, K. Heydemann, Q. Meunier, I. B. El Ouahma, Maskara: Compilation of a Masking Countermeasure With Optimized Polynomial Interpolation,

in: TCAD, IEEE, 2020.
[32] M. Rivain, E. Prouff, Provably secure higher-order masking of AES, in: CHES, Springer, 2010.
[33] M. Rivain, E. Dottax, E. Prouff, Block Ciphers Implementations Provably Secure Against Second Order Side Channel Analysis, in: FSE, LNCS, Springer, 127–143, 2008.
[34] M. Tunstall, C. Whitnall, E. Oswald, Masking Tables—An Underestimated Security Risk, in: FSE, 2014.
[35] J.-S. Coron, A. Roy, S. Vivek, Fast evaluation of polynomials over binary finite fields and application to side-channel countermeasures, in: CHES, 2014.
[36] N. Moro, K. Heydemann, E. Encrenaz, B. Robisson, Formal Verification of a Software Countermeasure Against Instruction Skip Attacks, JCEN .
[37] T. Barry, Outillage de conception et de compilation de protections logicielles pour la sécurité dans les systèmes embarqués., These, École nationale supérieure des mines de Saint-Etienne, 2017.