# Chapter 1
# The multiple ways to automate the application of software countermeasures against physical attacks: pitfalls and guidelines.

Nicolas Belleville, Karine Heydemann, Damien Couroussé, Thierno Barry, Bruno Robisson, Abderrahmane Seriai, and Henri-Pierre Charles

**Abstract** While the number of embedded systems is continuously increasing, securing software against physical attacks is costly and error-prone. Several works proposed solutions that automatically insert protections against these attacks in order to reduce this cost and this risk of error. In this paper, we present a survey of existing approaches, and classify them by the level at which they apply the countermeasure. We consider three different levels: the source code level, the compilation level and the assembly/binary level. We explain the advantages and disadvantages of each level considering different criteria. Finally, we encourage future works to take compilation into account when designing tools, to consider the problem of combining countermeasures, as well as the interactions between countermeasures and compiler optimisa-

Nicolas Belleville
Univ Grenoble Alpes, CEA, List, F-38000 Grenoble, France e-mail: nicolas.belleville@cea.fr

Karine Heydemann
Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, LIP6, F-75005, Paris, France e-mail: karine.heydemann@lip6.fr

Damien Couroussé
Univ Grenoble Alpes, CEA, List, F-38000 Grenoble, France e-mail: damien.courousse@cea.fr

Thierno Barry
Univ Grenoble Alpes, CEA, List, F-38000 Grenoble, France e-mail: thierno.barry@cea.fr

Bruno Robisson
CEA/EMSE, Secure Architectures and Systems Laboratory CMP, 880 Route de Mimet, 13541 Gardanne, France e-mail: bruno.robisson@cea.fr

Abderrahmane Seriai
Univ Grenoble Alpes, CEA, List, F-38000 Grenoble, France e-mail: abderrahmane.seriai@cea.fr

Henri-Pierre Charles
Univ Grenoble Alpes, CEA, List, F-38000 Grenoble, France e-mail: henri-pierre.charles@cea.fr

tions. Going one step further, we encourage future works to imagine how compilation could be modified or redesigned to optimize both performance and security.

## 1.1 Introduction

Nowadays, embedded systems have become integral part of our daily life and are of the largest consumer electronics market segment. The number of embedded systems a person manipulates every day is expected to rise massively due to the Internet of Things. Back in 2008, this number was already huge as a person used about 230 embedded chips every day [73].

These embedded systems often manipulate sensitive data. For instance, privacy-critical data are handled every day by payment cards, transport cards, as well as smartphones, GPS, etc. Therefore, the security of these systems reveals itself as a major concern for both industrials and state organizations.

Secure devices rely on cryptography to protect sensitive data. While they use cryptographic algorithms that are robust against cryptanalysis, attackers can exploit a physical access to a device either to extract sensitive data such as a cryptographic key, or to bypass authentication, or in certain cases to reverse engineer intellectual properties. These attacks, known as physical attacks, are of two categories. (1) Side channel attacks, introduced in 1996 by Kocher et al. [48], exploit the correlation between the data being processed inside the device and a set of physical quantities that can be measured from outside the device. These physical quantities can be the power consumption of the device [23, 49, 54, 78, 63], the electromagnetic radiation [6, 42], the acoustic emissions [43], execution time [36, 48], etc. (2) Fault injection attacks, introduced in 1997 by Boneh et al. [21], exploit the effect of a deliberate disturbance of a system during its operation. Fault injection attacks can be carried out by means of laser/light beam [39, 75], electromagnetic injection [65, 62, 35], variation of the supply voltage [12, 24], clock glitch [5], temperature [74, 46], etc.

Several protections to thwart physical attacks have been proposed at software and hardware levels. There are also some mixed hardware-software approaches [31, 19, 9]. In practice, secure elements rely both on hardware and software countermeasures. Moreover, hardware-based solutions are considered as to expensive for IoT devices that face strong cost requirements. The current software hardening process is most often manual, and so costly as well as error prone and tedious. Automating the deployment of software countermeasures is becoming paramount in order to reduce the overall cost and also to offer code hardening solutions for IoT devices.

In this survey, we present how automatic application of software countermeasures has been carried out in the literature by categorising approaches by the level where the countermeasure is applied; either on source code, or on assembly, or within the compilation process. We begin by a brief back-

ground (Section 1.2) about side-channel attacks, fault injection attacks, their countermeasures and the issues related to the compilation of secured code as well as usual ways to circumvent them. Then, we present the approaches that propose an automated application of a countermeasure (Section 1.3) at source code level, compilation level, and assembly level, and we point out their pros and cons. Then we take a step back to compare the different levels (Section 1.4.1). Finally we discuss the important remaining challenges (Section 1.4.2) before concluding.

## 1.2 Background

### *1.2.1 Side-channel attacks*

Instructions and data manipulated by a processor during a program execution affect the processor's power consumption, electromagnetic emissions, and execution time. Side-channel attacks exploit this correlation. Many side-channel attacks where proposed in the literature: [23, 49, 54, 78, 63] exploit the power consumption of a chip, [36, 48] the execution time of the implementation, [6, 42] the electromagnetic radiation of a chip.

During an attack, the attacker makes measurements of a physical quantity while the processor executes the targeted program. She then retrieves the data manipulated by the processor from these measurements, by statistically comparing the measurements with a behavioural model. In this survey, we focus on side-channel attacks that exploit the power consumption or the electromagnetic emissions.

In the case of a correlation power analysis (CPA), the attacker chooses the data she provides as an input to the program. To find the encryption key of an AES, she proceeds byte by byte. Each byte is found as follows: the attacker places an electromagnetic probe on the processor, or directly measures its electrical consumption with an oscilloscope. She carries out electrical consumption measurements during several AES executions. For each new run, she gives a random clear text to the program. She calculates theoretical consumptions for each value of the key byte that she is attacking using a consumption model (e. g. the Hamming weight of the value returned by the SBox of the first round). She compares the measurements obtained on several executions with the theoretical consumptions using a statistical operator, here the Pearson correlation. The byte hypothesis that gives the strongest correlation between theoretical and measured consumptions corresponds to the true value of the key byte if enough measurements have been taken.

### 1.2.2 Fault injection attacks

Processors are designed to work under certain conditions. By using a processor outside these conditions, for example at a high temperature, faults appear in the calculations [74, 46]. Fault injection attacks exploit this principle. They can use various physical means to provoke faults: light [39, 75], electromagnetic injection [65, 62, 35], temperature [74, 46], etc.

The effects of faults are manifold:

- bit flips in a register or a memory cell [20, 22, 14, 61, 38, 60],
- random modification of a value in a register,
- random modification of a value while it is transferred between the CPU and dynamic or non-volatile memory [56, 37],
- instruction replacement when the instruction fetch gets corrupted [56].

Fault injection attacks can then be used to hijack the execution flow of a program (e.g. to bypass a password verification of a VerifyPIN), or to retrieve information about data manipulated by the program (e.g. finding a cryptographic key). To retrieve a secret data, the attacker analyses the erroneous output that result from these faults, or even the absence of an error on the output, and compares this information using a fault attack model that makes the link between the expected output and the possible outputs in presence of faults.

### 1.2.3 Combined attacks

Combined attacks are physical attacks that combine side-channel analysis and fault injection.

Currently, all fault attacks are combined with a side channel observation in practice, in order to monitor the injection of the fault, i.e. to (1) find a suitable moment for the fault injection and (2) precisely control the moment when the fault is injected [77].

Second, some attacks use side-channel analysis and fault injection attack as steps of a wider attack [10]. Several approaches showed that these attacks can break implementations that were protected against both side-channel attacks and fault injection attacks, for example on an AES [71, 32] or ECC implementation [41].

### 1.2.4 Countermeasures

This section presents the main categories of countermeasures against side-channel attacks and fault injection attacks.

For side-channel attacks, we focus on side-channels related to power consumption or electromagnetic emissions, and on approaches that were evaluated on these side-channels.

Software countermeasures against side-channel attacks can be of 2 different natures: hiding and masking.

A hiding countermeasure is designed to make attacker's measurements too noisy to be exploitable [26]. For example, one can use dummy rounds or random delays, so that the measurements gathered in 2 different executions are no more aligned. The link between the measurements and the targetted information is not removed, but the exploitation of the measurements becomes more complicated. There are several types of hiding countermeasures: dummy rounds, random delays, static multiversionning, dynamic polymorphism, dual rail, etc [26, 27, 28, 8]. Static multiversionning consists in generating statically several different equivalent execution paths and choosing between them randomly at runtime. Runtime polymorphism consists in dynamically changing the binary code in memory, so that the code is renewed regularly. It was introduced by Amarilli et al. who indicated that it was possible to automatically implement such countermeasure [8]. Both static multiversionning and runtime polymorphism can use random delay insertion or instruction shuffling for example to make the code vary. As complementary approaches, the dual-rail and random precharging countermeasures are sometimes used. Dual rail with precharge logic consists in changing the value encoding so that the Hamming weight of manipulated values becomes a constant value, and precharging destination registers at the value 0 so that the Hamming distance becomes constant too. Random precharging consist in putting a random value in a register before loading a sensitive value into it in order to prevent transition based leakages. Note that hiding has been also used outside the scope of power consumption and electromagnetic emission side channels [30, 66, 45].

A masking countermeasure is designed to remove the direct link between the measurements and sensitive data manipulated by the processor [44]. For this purpose, the algorithm of the target program is modified so that all intermediate results that depend on the secret data are separated into several shares, where all the shares are needed to reconstruct the results. For example, Boolean masking consists in performing an "exclusive or" between the secret data and a random number and then carrying out all calculations with this masked data. The masked data and the random number are the two shares here. The random number is changed at each execution, so that the values of the shares change randomly from one execution to another. In practice, hiding and masking countermeasures are (and need to be) combined [70]. Indeed, masking need a certain amount of noise to be effective [47], and hiding can increase the noise.

Software countermeasures against fault attacks can be of 3 different types: fault tolerance, fault detection, or infective.

A fault tolerance countermeasure aims to ensure that a fault does not alter the output of a program. For example, an instruction duplication countermeasure can be used to tolerate a fault of the type "replacement of an instruction by a nop" [57]. A fault detection countermeasure is intended to detect an attack, and then allows to adapt the response to produce (e.g. destroying the system). Control flow integrity countermeasures are fault detection countermeasures that detect a change in control flow [33]. One can also duplicate instructions in order to compare the results to detect a fault. An infective countermeasure aims to make the result of a fault more difficult for an attacker to exploit [67]. The goal is that the attacker does not derive information from the program result when a fault occurred. It can be used as a reaction to a fault detection.

### *1.2.5 Compilation of secured code*

In this section, we give a brief background about compilation and the problem that can arise when compiling secured applications.

Compilation is the process of translating a source code into a binary program for a target architecture [76, 59, 11]. Compilers are usually divided in 3 parts.

- The front-end is in charge of parsing the source code and generating an intermediate representation (IR).
- The middle-end is responsible of target-independent optimisations. It is composed of a sequence of analysis and transformation passes that optimise the IR code.
- The back-end is responsible of target-dependant optimisations, as well as instruction selection and register allocation, and finally emits the binary program.

Several works have shown that the compiler can alter countermeasures against both side channel and fault injection attacks when these countermeasures are applied on the source code [13, 15, 72].

Countermeasures can be threatened by various passes. In the case of masking, the passes that simplify arithmetic operations, the instruction scheduling and register allocation passes may alter the countermeasure. For example, the compiler could invert the order of two xors, revelling a secret data. In the case of addition of noise instructions or of instruction redundancy, all the passes that suppress dead code may threaten the countermeasure. In the case of instruction shuffling, the instruction scheduling pass may also alter the countermeasure. Please note that this list is not an extensive list of passes that could threaten the countermeasures. Such a list depends on the compiler, its version, the target architecture, etc.

In order to circumvent this problem, one can use various ways:

- One can compile code using the -O0 optimisation flag so that few optimisations remain enabled. Yet, the compilation process remains risky: the code still go through instruction selection, register allocation, and instruction scheduling for example, each of these passes being able to alter some countermeasures. In addition, it increases the code surface available for an attack, and there are a lot of register spilling and filling, which increase the information available via side-channel.
- One can use the `volatile` keyword in C/C++ source code to force the compiler to not perform memory-access optimisations on some selected variable.
- One can disable some specific passes by using the compiler command line options.
- One can inline assembly code in its source code. However, this solution leads to complex implementations, as developers have to make the link between the C/C++ variables and physical registers. Moreover, the source code is no more portable, and becomes harder to maintain.
- One can apply directly the countermeasure on assembly, so that the compilation problem is bypassed. We will see later however that this solution has drawbacks too.

## 1.3 Automatic application of software countermeasures

This section presents several state-of-the-art approaches proposed for automatic application of software protection. Table 1.1 shows an overview of the approaches presented in this section. We present the different approaches, both in this section and in the Table 1.1, gathered according to the code level on which the automatic application is carried out.

The different levels of application for automated approaches will be mainly compared on both their ease of use and the complexity of their implementation. The pros and cons of each level of application will also be presented.

We consider as a usage constraint either the replacement of programming language or the replacement of tools in the developer's usual production chain, such as the compiler. Indeed, changing the programming language can prevent from reusing reference implementations. Also, replacing one of the tools in a production toolchain may not be possible: as an example, closed-source software components do not offer the ability to modify the source code or some components may have been certified and any modification would require a new certification process.

While comparing the security level achieved for a specific approach as well as the impact of a protection on performance and code size would be of high interest, it is quite impossible to achieve. Evaluations carried out in the literature vary with the target platform, the considered benchmarks and the attacks or tests performed. To fairly compare all the approaches, it

would then require to dispose of all approaches, to choose a common target of evaluation and to mount realistic security evaluation scenarios. Hence, we only report fair performance comparison of approaches available in the literature (between approaches [57] and [16]).

### *1.3.1 At source code level*

Several approaches proposed to automatically apply countermeasures at source code level.

#### 1.3.1.1 Side channel attack countermeasures

Luo et al. proposed an automated hiding countermeasure where independent C operations are shuffled [52]. The associated tool takes C code as input. It gathers statements by group of independent statements, and shuffling is performed at runtime inside each group. It adds dummy statements when too few independent statements have been found for a group in order to increase shuffling effect. It assumes that the code does not contain any loop or branch.

Couroussé et al. proposed an approach to deploy a hiding countermeasure based on runtime polymorphic code generation [29]. Their approach requires to use a dedicated language (DSL). The written code is translated by a tool that produces the C code of a specialized polymorphic code generator. The generator regularly produces new versions of the machine code at runtime using semantic variants at machine instruction level, instructions and registers shuffling, and insertion of noise instructions.

Eldib et al. proposed an approach to automatically find and apply a masking countermeasure, with the help of a SMT solver [40]. They assume that the program has an input-independent control flow. The program is parsed and transformed into LLVM's intermediate representation (LLVM IR) by clang. The code in LLVM IR format is then transformed into a Boolean program. Then, each operation of the program is masked, directly if it is a linear operation, or by finding a sequence of equivalent masked instructions found out by a SMT solver. Then, the secured code is emitted as C++ code and compiled in -O0 (this information comes from a discussion with authors).

#### 1.3.1.2 Fault injection countermeasures

Lalande et al. proposed to apply a control flow integrity countermeasure based on counters and additional variables at the source code level [50]. The countermeasure is applied in 2 phases; first, all vulnerabilities of the original code are searched for by simulating control flow hijacking faults at the source

**Table 1.1** Overview of existing automated approaches for side channel attacks or fault injection attacks gathered by the code level at which they are deployed. Few approaches consider several different countermeasures, and none of them considers countermeasures for both families of attacks simultaneously.

| | Approach | Countermeasure principle | | Requirements or constraints |
|---|---|---|---|---|
| | | Side channel attacks | Fault injection attacks | |
| Source level | [52] | static multiversionning (hiding) | none | Straight-line code |
| | [29] | polymorphism with runtime code generation (hiding) | none | Domain specific language |
| | [40] | masking | none | No input-dependent control flow |
| | [50] | none | control flow integrity | - |
| | [7] | none | control flow integrity | - |
| Compiler level | [53] | static multiversionning (hiding) | none | - |
| | [2] | polymorphism with runtime code modification (hiding) | none | - |
| | [4] | static multiversionning (hiding) and partial masking | none | - |
| | [3] | other | none | - |
| | [58] | masking | none | Domain specific language |
| | [1] | masking | none | - |
| | [18] | random precharging and masking | none | Measurements (optional) |
| | [51] | threshold implementation (masking) | none | - |
| | [16] | none | instruction duplication (fault tolerance) | - |
| | [69] | none | instruction duplication (fault detection) and control flow integrity | - |
| | [64] | none | instruction duplication on loop exits (fault detection) | - |
| | [25] | none | instruction and data redundancy (fault detection) | Availability of SIMD instructions |
| Assembly level | [17] | random precharging | none | Measurements to set up the protection |
| | [68] | dual-rail with precharge logic | none | Bitsliced input code |
| | [57] | none | instruction duplication (fault tolerance) | - |
| | [34] | none | various fault detection and fault tolerance countermeasures | - |

code level, then the countermeasure is applied to vulnerable points. Jump attacks larger than two C statements are systematically detected. However, smaller faults, e.g. that only affect one assembly instruction, are not always detected.

Akkar et al. also presented an automated application of a control flow integrity countermeasure [7]. The developer must annotate his code beforehand using pragmas to indicate the areas to secure. The application is done by a tool that comes in the form of a pre-processor.

#### 1.3.1.3 Pros and cons of source code level

The source code level has the advantage of being compatible with the use of proprietary compilers, and even of allowing the use of several different toolchains without any compatibility concerns.

In addition, a substantial amount of information is available at this level, such as variable typing information.

This level of application also enables to be independent of the target architecture. Thus, the development of a tool may be easier at this level if a lot of architectures have to be supported.

However, the countermeasures may be altered by compilation. This is not always the case, for example in the COGITO approach [29], the countermeasure is applied at runtime by a dedicated generator and therefore there is no risk that it will be altered by the compilation. Approaches [50] and [40] suggest to compile the secure parts without compiler optimizations to circumvent this problem, which does not remove completely the risk as discussed in Section 1.2.5. Thus, developers will have to check for each hardened application at source code level that the countermeasures are still present and correct after compilation. This typically involves reviewing the assembly code produced by the compiler, which is a tedious and error-prone task.

### *1.3.2 During compilation*

Several approaches proposed to apply countermeasures during compilation. Table 1.2 summarizes the level of application inside the compiler and the passes that have been modified for each approach.

#### 1.3.2.1 Side channel attack countermeasures

Malagón et al. proposed to deploy a hiding countermeasure based on static generation of several variants of a function [53]. This countermeasure consists in randomly choosing between different versions of the same code at

**Table 1.2** Level of application and modified passes within the compiler for compiler-level approaches

| Approach | Level of application | Modified passes |
|---|---|---|
| Malagón et al. [53] | Middle-end | Loop unwinding pass |
| Agosta et al. [2] | Unknown | - |
| Agosta et al. [4] | Middle-end and back-end | Several (unknown) passes in middle-end and back-end |
| Agosta et al. [3] | Middle-end and back-end | Instruction selection |
| Moss et al. [58] | Middle-end | - |
| Agosta et al. [1] | Middle-end | - |
| Bayrak et al. [18] | Middle-end and back-end | - |
| Eldib et al. [40] | Middle-end | - |
| Barry et al. [16] | Backend | Instruction selection and register allocation |
| Reis et al. [69] | Unknown | - |
| Proy et al. [64] | Middle-end and back-end | Branch folding and register allocation |
| Chen et al. [25] | Middle-end | - |
| Luo et al. [51] | Middle-end | - |

runtime. The source code must be annotated using pragmas by developers to indicate functions where sensitive data are being manipulated. The compiler then generates several different versions of the function code by changing optimizations configuration parameters, for example using the loop unwinding pass. It also inserts the code that is in charge of randomly selecting at runtime the version of the code to be executed.

Agosta et al. proposed another hiding countermeasure based on dynamic modification of code [2]. The code is modified at runtime using semantic equivalence at instruction level, randomization of table accesses, mixed instructions. The countermeasure is automatically applied by a compiler: some transformation passes have been added in LLVM in order to statically prepare the transformations made at runtime.

Agosta et al. also proposed a hiding countermeasure based on static generation of several variants [4]. The authors propose to generate automatically a code containing multiple execution paths, with choice between the different paths at runtime, which is also a hiding countermeasure. This approach also incorporates some masking elements, since the SBox accesses are masked. In addition, the process of saving registers on the stack is modified: one register is dedicated to hold a random value used to mask any register value stored in the stack. When the content of the register is restored, it is also unmasked so that it can be used again. All these transformations are handled by new transformation passes in LLVM. Some existing passes have also been modified. Among other things, modifications to existing passes are intended to ensure that an instruction that was in an area to be protected cannot leave this area because of optimizations. The developer must provides a C file annotated so as to specify the code regions to protect and the SBox. In addition

to the source file, the compiler takes an input file that specifies the equivalent instructions to be used.

Agosta et al. also proposed a new countermeasure against side-channel attacks that aims to bring out several key hypotheses instead of one during an attack so that the attacker cannot know which one is the right hypothesis [3]. This countermeasure is entirely applied during compilation, in several steps. Several passes have been added in the middle-end and back-end, also the instruction selection pass has been changed. The compiler takes an input file annotated by the developer that specifies the parts of the code to be protected.

Moss et al. proposed to automatically apply a boolean masking countermeasure during compilation [58]. The developer must write his program in a domain specific language (DSL). This DSL allows to express with predefined types the level of confidentiality of variables, for example to indicate that a variable is secret. The compiler then uses this information to determine which intermediate values are to be masked, and thus masks these values.

Agosta et al. also proposed an approach for the application of a masking countermeasure. Their approach allows to generate higher-order masked code [1]. The compiler calculates for each key-dependent value the number of key bits on which the value depends. This analysis enables to apply the countermeasure only to intermediate values that depend on a small number of key bits. For example, intermediate values dependent on all bits of the key are not masked. This principle reduces the overhead of the countermeasure.

Bayrak et al. also proposed a compilation approach to apply boolean masking to a program [18]. An important difference with the other approaches is that they use the compiler to decompile a binary program to a higher level representation and then recompile the program while applying the protection. To find out where to apply the countermeasure, they suggest to start by identifying instructions that may reveal sensitive data through a side-channel. This analysis is either done using measurements provided by the user or statically. The countermeasure is then applied to all instructions that were found to be critical compared to a predefined threshold. This enables to partially apply the countermeasure and to reduce the performance overheads. In addition, the compiler can also apply a random precharging countermeasure.

Luo et al. proposed a similar approach to generate a threshold implementation automatically on LLVM IR [51]. Threshold implementation is a countermeasure close to the masking countermeasure, as the secret is split into shares. Yet, in threshold implementation, every function is independent from at least one of the shares, which is not the case for masking. They use a SAT solver along with a transformation step in order to find suitable solution. Every function is split into a succession of smaller functions so that the SAT solver can find solutions effectively.

#### 1.3.2.2 Fault injection countermeasures

Barry et al. used the compiler to automatically apply a fault tolerance countermeasure [16]. They duplicate assembly instructions to tolerate the skip of one instruction. The use of the compiler is twofold compared to a lower level approach: it favours the selection of instructions compliant with the duplication scheme, increasing the number of idempotent instructions, and takes advantage of optimisations to gain performance. To this end, several passes have been added to LLVM and the instruction selection and register allocation passes have been modified. The overheads obtained are lower than those obtained by applying this countermeasure at the assembly level.

Reis et al. proposed to deploy a fault detection countermeasure during compilation [69]. Instructions are duplicated so that their results are compared in order to detect faults. In addition, additional checks are added to ensure that the control flow is not hijacked. The authors indicate that the approach could be easily extended to include fault tolerance.

Proy et al. proposed to use the compiler to apply a countermeasure to secure loops against fault injection attacks [64]. The instructions involved in the computation of conditions for exiting the loop are duplicated to add checking blocks in charge of detecting an early exit or a extra iterations. This transformation is applied at IR-level. They explain that some compiler passes had to be modified to keep the countermeasure correctly applied until the code is emitted.

Finally, Chen et al. proposed to achieve operation redundancy by using SIMD instructions [25]. Their compiler vectorises some instructions in order to have instruction redundancy, and adds error checking codes. All the code transformations are performed at the IR-level, and the approach is architecture-independent. It only requires the target architecture to have support for SIMD instructions. The use of SIMD instructions allows to obtain a smaller performance overhead compared to classic instruction duplication approaches.

#### 1.3.2.3 Pros and cons of compiler level

The compiler level is interesting if several source languages need to be supported, as the front-end usually support various languages.

Moreover, the back-end must most often be modified and therefore the approach depends on the architecture. However, some elements applied in the middle-end are common for all architectures, so adding support for an architecture is done without starting from scratch.

What is more, the application of countermeasures during the compilation process makes it possible to finely control the transformations carried out in the compiler, and to choose when to apply the countermeasure to avoid the risk that it will be altered by the compilation. The compiler allows to have

both high-level information such as the types of variables, as well as low-level information that depends on the target architecture. Thus, countermeasure can be applied in several transformation passes, strategically placed in the compilation process. As an example, Reis et al. [69] and Barry et al. [16] exploit the scheduling instruction pass to reduce the countermeasure overhead by creating parallelism at the instruction level (depending on the latency of the instructions). In addition, several approaches modify compiler transformation passes such as instruction selection or register allocation to prepare the countermeasure application in order to produce a more efficient code.

Moreover, if developers manage to propagate the necessary information throughout the compilation process, developers can add a check pass before issuing instructions to confirm that the countermeasure has been correctly applied and that it has not been altered by possible downstream optimizations.

The engineering effort deployed to implement such approaches is important, nevertheless, the control offered by this level of application makes it possible to obtain an important confidence in the produced code. In case a checking pass is added before code emission, it is not necessary to manually check the presence and the effectiveness of the countermeasures in the produced assembly code for each hardened application. In that case, the developer does not need to be an expert in security to be able to effectively secure his applications.

This level of application requires to have access to the compiler source code. If the developer uses a closed-source compiler, using a compiler approach would implies to use an open-source one to disassemble a file, reconstruct an intermediate representation, apply the countermeasure to the code and recompile it, which is a tough process.

### *1.3.3 At link time / at assembly level*

This section presents approaches that apply countermeasures directly on an assembly file, during or before the linking phase.

#### 1.3.3.1 Side channel attack countermeasures

Bayrak et al. proposed to automatically apply a random precharging countermeasure at assembly level [17]. The application of this countermeasure is quite natural at this level, as register allocation has already been performed. Empirical measurements made on unsecure code are used to determine the instructions to be secured.

Rauzy et al. also implemented a side-channel countermeasure at assembly level: dual-rail with precharge logic [68]. Their approach requires that the

code has previously been bitsliced. Their approach also makes it possible to prove that the transformation is correct and that the program obtained after transformation remains semantically correct.

#### 1.3.3.2 Fault attack countermeasures

Moro et al. proposed a countermeasure based on instruction duplication to achieve fault tolerance [55]. This countermeasure is intended to tolerate the jump of one instruction. For this purpose, each instruction is replaced by a sequence of instructions, this sequence being semantically equivalent to the original instruction and being tolerant to one instruction skip. As this countermeasure requires additional registers, it is sometimes necessary to spill some registers. In addition, some instructions (e.g. volatile loads) cannot be replaced by a fault tolerant sequence. This is the same countermeasure as the one automated by Barry et al. [16] afterwards at compilation level.

De Keulenaer et al. showed how to automatically deploy various countermeasures against fault attacks at binary level using link-time rewriting [34]. Their tool combines both fault tolerance countermeasures and fault detection countermeasures: duplication of conditional jumps, call graph integrity, verification of memory entries, duplication of loop counters.

#### 1.3.3.3 Pros and cons of assembly level

This level is mostly used to apply countermeasures that are quite low level, as applying higher level countermeasures at this level is complicated since it is then necessary to reconstruct a certain amount of information that has been lost. For example, variable typing information is no longer present. In addition, the application of countermeasures often requires the use of additional registers, which requires either register spilling or a complete reallocation of registers.

Thus, during the development of an automatic approach at this level, a major engineering effort is necessary to obtain information that was available at compilation, or to redo treatments that had been done by the compiler in a way that was not optimal with respect to the countermeasure to be applied.

However, applying countermeasures at this level avoids having to check manually if the countermeasure is still present in the final code, since the compilation process takes place entirely before the countermeasures application. This allows the use of such a tool by a non-security expert developer. Moreover, this level of application allows to be independent of source code language, which is interesting if several source languages need to be supported. In addition, it allows to secure code after link time optimisation, and to potentially secure binary libraries.

## 1.4 Discussion

### *1.4.1 Confrontation of pros and cons of the different levels*

This section discusses the advantages and disadvantages of the aforementioned levels of automatic application of countermeasures.

The first aspect to consider is the time taken for developing an automated tool. This aspect depends on the countermeasure that has to be applied. A masking countermeasure is easier to apply at source code level than at assembly level because it requires a modification of the algorithm. The compiler is a place where various countermeasures can be applied, as during compilation the compiler manipulates both quite high level representations (e.g. with typed variables) and low level representations (e.g. with assembly instructions).

Developing an automated tool implies parsing and emitting code in the targetted formats. Compilers already have the necessary code for that, and usually the developer only has to add a pragma support to delimit the code zones to be secured. For source code and assembly levels approaches, the developers often have to implement or reuse a parser and/or an emitter for the targetted codes.

The engineering cost taken at using the tools must be considered too. As these tools are automatic, the cost of producing secured code is close to zero, yet the development of the tools requires a lot of work. When the tool applies the countermeasure at source code level, code-review is facilitated, but the user has to check that the countermeasure is still valid at the assembly level. This time consuming task is one of the main drawback of the source code level approach. The assembly approach does not suffer from this drawback: applying a countermeasure at the assembly level prevent from alteration during compilation. Applying a countermeasure during compilation allows to check that the countermeasure is still valid just before assembly/binary code emission if the developer manage to propagate the necessary information throughout the compiler. If checking the countermeasure before code emission is not possible, a step of assembly code review is still needed.

Considering performance in terms of code size and of execution time, the compiler level allows fine tuning. When a countermeasure is applied within the compiler, it can benefit from optimisations, whereas if it is applied outside the compilation process, it requires to redevelop some optimisations afterwards. Several approaches that use compilers modify some passes of the compiler to reduce the cost of the countermeasures. The passes that apply the countermeasure can be carefully interleaved with compilers passes to take advantage of these passes without risking the countermeasure to get altered by optimisations [16]. At other levels, tuning transformations for performance may be harder. For example, at assembly level, the need for additional reg-

isters either require to do register spilling, or to perform again the register allocation. As a comparison, Moro et al. and Barry et al. implemented the same countermeasure at assembly level and compiler level respectively. Barry et al. obtained execution time overheads and size overheads lower than Moro et al.

### *1.4.2 Future works*

All of these approaches target either side-channel attacks or fault injection attacks, and few of them consider the application of several different countermeasures. Yet, programs have to be secured against both families of attacks, and within each family of attack, have to be secured against a large number of variants. Thus, countermeasures have to be combined so that the programs meet the security requirements.

The problem of automatic application of combined countermeasure has not been investigated yet to the best of our knowledge. It raises important questions in order to be able to guarantee that every countermeasure is correctly applied on the produced code.

Similarly to the conflicts that can appear between countermeasures and some optimisations passes of a compiler, conflicts can appear between different countermeasures. The order of application of the countermeasures should be well thought: which countermeasure must be applied first? Must the countermeasures be applied in a combined way? Several compiler approaches proposed to apply a countermeasure in several steps, interleaved with compiler passes. How should one interleave all the different steps to apply two very different countermeasures? This issue is present whatever the level at which countermeasures are applied, and refrains the simple approach that would consist in simply combining several different tools one after the other as they would not be aware of the countermeasures that are applied by the others.

In addition, when the compiler level is chosen to apply the countermeasures, strategies for the modification of compiler passes have to be made with all countermeasures in mind. For example, register allocation should be compliant with several countermeasures that may have different objectives: one may want to constrain register spilling to prevent distance-based leakage in presence of a masking countermeasure, while needing new registers to implement a fault detection countermeasure.

We encourage future works to consider the problem of compilation for security, to study the interaction between the different countermeasures and performance optimisations, and to rethink the compilation process so that it can optimize at the same time the performance and security goals.

## 1.5 Conclusion

The automatic application of countermeasures against physical attacks is a crucial research problem as a lot of platforms are concerned by these threats while securing them manually is costly. We presented the different approaches to automatically deploy software countermeasures against these attacks. Some of them directly modify the source code, others modify the assembly code, while others propose to modify the compiler so that the countermeasure is applied during the compilation process. While developing solutions at the compilation level is not always possible, we encourage this practice as it allows to tune performance while providing confidence that the countermeasure remains correctly applied in the assembly file. We also encourage future research to consider the problem of automatic application of combined countermeasures that as not yet been addressed, their interaction with compiler optimisations, and to try to create compilers that optimise both security and performance. These issues are interesting and challenging issues to solve to be able to offer security automatically.

## References

1. G. Agosta, A. Barenghi, M. Maggi, and G. Pelosi. Compiler-based side channel vulnerability analysis and optimized countermeasures application. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–6. IEEE, 2013.
2. G. Agosta, A. Barenghi, and G. Pelosi. A code morphing methodology to automate power analysis countermeasures. In *DAC*, pages 77–82, 2012.
3. G. Agosta, A. Barenghi, G. Pelosi, and M. Scandale. Information leakage chaff: feeding red herrings to side channel attackers. pages 1–6. ACM Press, 2015.
4. G. Agosta, A. Barenghi, G. Pelosi, and M. Scandale. The MEET Approach: Securing Cryptographic Embedded Software Against Side Channel Attacks. *IEEE TCAD*, 34(8):1320–1333, 2015.
5. M. Agoyan, J.-M. Dutertre, D. Naccache, B. Robisson, and A. Tria. When clocks fail: On critical paths and clock faults. *LNCS*, 6035 LNCS:182–193, 2010.
6. D. Agrawal, B. Archambeault, J. Rao, and P. Rohatgi. The em Side-Channel(s). *LNCS*, 2523:29–45, 2003.
7. M.-L. Akkar, L. Goubin, and O. Ly. Automatic integration of counter-measures against fault injection attacks. *Pre-print found at http://www. labri. fr/Perso/ly/index. htm*, 2003.
8. A. Amarilli, S. Müller, D. Naccache, D. Page, P. Rauzy, and M. Tunstall. Can Code Polymorphism Limit Information Leakage? In Springer, editor, *WISTP*, volume LNCS 6633, pages 1–21, 2011.
9. J. Ambrose, R. Ragel, and S. Parameswaran. RIJID: Random Code Injection to Mask Power Analysis based Side Channel Attacks. In *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pages 489–492, June 2007.
10. F. Amiel, K. Villegas, B. Feix, and L. Marcel. Passive and active combined attacks - Combining fault attacks and side channel analysis -. pages 92–99, 2007.
11. A. W. Appel and M. Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, New York, NY, USA, 2004.

12. C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert. Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures. *LNCS*, 2523:260–275, 2003.
13. J. Balasch, B. Gierlichs, V. Grosso, O. Reparaz, and F.-X. Standaert. On the Cost of Lazy Engineering for Masked Software Implementations. *LNCS*, 8968:64–81, 2015.
14. H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.
15. M. Barbosa, A. Moss, and D. Page. Constructive and Destructive Use of Compilers in Elliptic Curve Cryptography. *Journal of Cryptology*, 22(2):259–281, Apr. 2009.
16. T. Barry, D. Couroussé, and B. Robisson. Compilation of a Countermeasure Against Instruction-Skip Fault Attacks. In *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems*, pages 1–6. ACM, 2016.
17. A. Bayrak, F. Regazzoni, P. Brisk, F.-X. Standaert, and P. Ienne. A first step towards automatic application of power analysis countermeasures. pages 230–235, 2011.
18. A. G. Bayrak, F. Regazzoni, D. Novo, P. Brisk, F.-X. Standaert, and P. Ienne. Automatic application of power analysis countermeasures. *IEEE Transactions on Computers*, 64(2):329–341, 2015.
19. A. G. Bayrak, N. Velickovic, P. Ienne, and W. Burleson. An Architecture-independent Instruction Shuffler to Protect Against Side-channel Attacks. *ACM Trans. Archit. Code Optim.*, 8(4):20:1–20:19, Jan. 2012.
20. I. Biehl, B. Meyer, and V. Müller. Differential Fault Attacks on Ellitic Curve Cryptosystems. In M. Bellare, editor, *Advances in Cryptology (CRYPTO 2000)*, volume 1880 of *LNCS*, pages 131–146. Springer, 2000.
21. D. Boneh, R. A. DeMillo, and R. J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 37–51. Springer, 1997.
22. D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of eliminating errors in cryptographic computations. *J. Cryptology*, 14:101–119, 2001.
23. E. Brier, C. Clavier, and F. Olivier. Correlation power analysis with a leakage model. *LNCS*, 3156:16–29, 2004.
24. R. B. Carpi, S. Picek, L. Batina, F. Menarini, D. Jakobovic, and M. Golub. Glitch It If You Can: Parameter Search Strategies for Successful Fault Injection. In *Smart Card Research and Advanced Applications*, Lecture Notes in Computer Science, pages 236–252. Springer, Cham, Nov. 2013.
25. Z. Chen, J. Shen, A. Nicolau, A. Veidenbaum, and N. Farhady. CAMFAS: A Compiler Approach to Mitigate Fault Attacks via Enhanced SIMDization. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 57–64. IEEE, 2017.
26. C. Clavier, J.-S. Coron, and N. Dabbous. Differential Power Analysis in the Presence of Hardware Countermeasures. In *Cryptographic Hardware and Embedded Systems — CHES 2000*, Lecture Notes in Computer Science, pages 252–263. Springer, Berlin, Heidelberg, Aug. 2000.
27. J.-S. Coron and I. Kizhvatov. An efficient method for random delay generation in embedded software. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5747 LNCS:156–170, 2009.
28. J.-S. Coron and I. Kizhvatov. Analysis and improvement of the random delay countermeasure of CHES 2009. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6225 LNCS:95–109, 2010.
29. D. Couroussé, T. Barry, B. Robisson, P. Jaillon, O. Potin, and J.-L. Lanet. Runtime Code Polymorphism as a Protection Against Side Channel Attacks. volume Volume 9895, pages pp 136–152, Sept. 2016.
30. S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. Internet Society, 2015.

31. J.-L. Danger, S. Guilley, T. Porteboeuf, F. Praden, and M. Timbert. HCODE: Hardware-Enhanced Real-Time CFI. pages 1–11. ACM Press, 2014.
32. F. Dassance and A. Venelli. Combined fault and side-channel attacks on the AES key schedule. pages 63–71, 2012.
33. R. de Clercq and I. Verbauwhede. A survey of Hardware-based Control Flow Integrity (CFI). *arXiv preprint arXiv:1706.07257*, 2017.
34. R. De Keulenaer, J. Maebe, K. De Bosschere, and B. De Sutter. Link-time smart card code hardening. *International Journal of Information Security*, 15(2):111–130, 2016.
35. A. Dehbaoui, J. M Dutertre, B. Robisson, P. Orsatelli, P. Maurine, and A. Tria. Injection of transient faults using electromagnetic pulses Practical results on a cryptographic system. Jan. 2012.
36. J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestré, J.-J. Quisquater, and J.-L. Willems. A practical implementation of the timing attack. *LNCS*, 1820:167–182, 2000.
37. L. Dureuil, M. Potet, P. de Choudens, C. Dumas, and J. Clédière. From code review to fault injection attacks: Filling the gap using fault model inference. In *Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers*, pages 107–124, 2015.
38. P. Dusart, G. Letourneux, and O. Vivolo. Differential Fault Analysis on AES. In M. Yung, Y. Han, and J. Zhou, editors, *Applied Cryptography and Network Security (ANCS 2003)*, volume 2846 of *LNCS*, pages 293–306. Springer, 2003.
39. J.-M. Dutertre, C. De, A. Sarafianos, N. Boher, B. Rouzeyre, M. Lisart, J. Damiens, P. Candelier, M.-L. Flottes, and N. Di. Laser attacks on integrated circuits: From CMOS to FD-SOI. 2014.
40. H. Eldib and C. Wang. Synthesis of Masking Countermeasures Against Side Channel Attacks. In *International Conference on Computer Aided Verification*, pages 114–130. Springer, 2014.
41. J. Fan, B. Gierlichs, and F. Vercauteren. To infinity and beyond: Combined attack on ECC using points of low order. *LNCS*, 6917 LNCS:143–159, 2011.
42. K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: Concrete results. *LNCS*, 2162:251–261, 2001.
43. D. Genkin, A. Shamir, and E. Tromer. Acoustic Cryptanalysis. *Journal of Cryptology*, 30(2):392–443, 2017.
44. L. Goubin and J. Patarin. DES and Differential Power Analysis (The "Duplication" Method). In *Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '99, pages 158–172, London, UK, UK, 1999. Springer-Verlag.
45. A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Librando: transparent code randomization for just-in-time compilers. pages 993–1004. ACM Press, 2013.
46. M. Hutter and J.-M. Schmidt. The temperature side channel and heating fault attacks. *LNCS*, 8419 LNCS:219–235, 2014.
47. A. Journault and F.-X. Standaert. Very High Order Masking: Efficient Implementation and Security Evaluation. In *Cryptographic Hardware and Embedded Systems – CHES 2017*, Lecture Notes in Computer Science, pages 623–643. Springer, Cham, Sept. 2017.
48. P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology—CRYPTO'96*, pages 104–113. Springer, 1996.
49. P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. *LNCS*, 1666:388–397, 1999.
50. J.-F. Lalande, K. Heydemann, and P. Berthomé. Software Countermeasures for Control Flow Integrity of Smart Card C Codes. In *European Symposium on Research in Computer Security*, pages 200–218. Springer, 2014.
51. P. Luo, K. Athanasiou, L. Zhang, Z. H. Jiang, Y. Fei, A. A. Ding, and T. Wahl. Compiler-Assisted Threshold Implementation against Power Analysis Attacks. pages 541–544. IEEE, Nov. 2017.

52. P. Luo, L. Zhang, Y. Fei, and A. A. Ding. Towards secure cryptographic software implementation against side-channel power analysis attacks. In *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*, pages 144–148. IEEE, 2015.
53. P. Malagón, G. de, M. Zapater, J. Moya, and Z. Banković. Compiler optimizations as a countermeasure against side-channel analysis in MSP430-based devices. *Sensors (Switzerland)*, 12(6):7994–8012, 2012.
54. S. Mangard, E. Oswald, and T. Popp. *Power Analysis attacks: Revealing the secrets of smart cards*. Power Analysis Attacks: Revealing the Secrets of Smart Cards. 2007. DOI: 10.1007/978-0-387-38162-6.
55. N. Moro. *Security of assembly programs against fault attacks on embedded processors*. Theses, Université Pierre et Marie Curie - Paris VI, Nov. 2014.
56. N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz. Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, pages 77–88. IEEE, 2013.
57. N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson. Formal Verification of a Software Countermeasure Against Instruction Skip Attacks. *Journal of Cryptographic Engineering*, 4(3):145–156, 2014.
58. A. Moss, E. Oswald, D. Page, and M. Tunstall. Compiler assisted masking. *LNCS*, 7428 LNCS:58–75, 2012.
59. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.
60. S. Ordas, L. Guillaume-Sage, and P. Maurine. EM injection: Fault model and locality. *2015 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 00:3–13, 2015.
61. S. Ordas, L. Guillaume-Sage, K. Tobich, J.-M. Dutertre, and P. Maurine. Evidence of a larger em-induced fault model. In *International Conference on Smart Card Research and Advanced Applications*, pages 245–259. Springer, 2014.
62. S. Ordas, L. Guillaume-Sage, K. Tobich, J.-M. Dutertre, and P. Maurine. Evidence of a larger EM-induced fault model. *LNCS*, 8968:245–259, 2015.
63. E. Peeters. *Advanced DPA theory and practice: Towards the security limits of secure embedded circuits*, volume 9781461467830 of *Advanced DPA Theory and Practice: Towards the Security Limits of Secure Embedded Circuits*. 2013. DOI: 10.1007/978-1-4614-6783-0.
64. J. Proy, K. Heydemann, A. Berzati, and A. Cohen. Compiler-Assisted Loop Hardening Against Fault Attacks. *ACM Trans. Archit. Code Optim.*, 14(4):36:1–36:25, Dec. 2017.
65. J.-J. Quisquater and D. Samyde. ElectroMagnetic Analysis (EMA): Measures and Counter-measures for Smart Cards. In *Smart Card Programming and Security*, Lecture Notes in Computer Science, pages 200–210. Springer, Berlin, Heidelberg, 2001. DOI: 10.1007/3-540-45418-7_17.
66. A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing Digital Side-channels Through Obfuscated Execution. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 431–446, Berkeley, CA, USA, 2015. USENIX Association.
67. P. Rauzy and S. Guilley. Countermeasures against high-order fault-injection attacks on CRT-RSA. pages 68–82, 2014.
68. P. Rauzy, S. Guilley, and Z. Najm. Formally proved security of assembly code against power analysis: A case study on balanced logic. *Journal of Cryptographic Engineering*, 6(3):201–216, 2016.
69. G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *Proceedings of the international symposium on Code generation and optimization*, pages 243–254. IEEE Computer Society, 2005.
70. M. Rivain, E. Prouff, and J. Doget. Higher-Order Masking and Shuffling for Software Implementations of Block Ciphers. In *Cryptographic Hardware and Embedded Systems - CHES 2009*, Lecture Notes in Computer Science, pages 171–188. Springer, Berlin, Heidelberg, 2009. DOI: 10.1007/978-3-642-04138-9_13.

71. T. Roche, V. Lomné, and K. Khalfallah. Combined fault and side-channel attack on protected implementations of AES. *LNCS*, 7079 LNCS:65–83, 2011.
72. H. Seuschek, F. De Santis, and O. M. Guillen. Side-channel leakage aware instruction scheduling. pages 7–12. ACM Press, 2017.
73. J. Sifakis. A vision for computer science — the system perspective. *Open Computer Science*, 1(1), Jan. 2011.
74. S. Skorobogatov. Local heating attacks on flash memory devices. pages 1–6, 2009.
75. S. Skorobogatov and R. Anderson. Optical Fault Induction Attacks. *LNCS*, 2523:2–12, 2003.
76. Y. Srikant and P. Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*. CRC Press, 2007.
77. N. Timmers and A. Spruyt. Bypassing secure boot using fault injection, 2016.
78. J. VanLaven, M. Brehob, and K. Compton. A computationally feasible SPA attack on AES via optimized search. *IFIP Advances in Information and Communication Technology*, 181:577–588, 2005.