Formally verified software countermeasures for control-flow integrity of smart card C code

Karine Heydemann, Jean-François Lalande, Pascal Berthomé

Formally verified software countermeasures for control-flow integrity of smart card C code

Karine Heydemann[a], Jean-François Lalande[b], Pascal Berthomé[c]

[a]*Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, LIP6, F-75005 Paris, France*
[b]*CentraleSupélec, Univ Rennes, Inria, CNRS, IRISA, UMR 6074, F-35065 Rennes, France*
[c]*INSA Centre Val de Loire, University of Orléans, LIFO, EA 4022, 18022 Bourges, France*

Abstract

Fault attacks can target smart card programs to disrupt an execution and take control of the data or the embedded functionalities. Among all possible attacks, control-flow attacks aim at disrupting the normal execution flow. Identifying harmful control-flow attacks and designing countermeasures at the software level are tedious and tricky for developers. In this paper, we propose a methodology to detect harmful inter- and intra-procedural jump attacks at the source code level and automatically inject formally proven countermeasures into a C code. The proposed software countermeasures protect the integrity of individual statements at the granularity of single C statements. They support many control-flow constructs of the C language. The countermeasure scheme can detect an attack early either inside a control-flow construct or only at its exit. The secured source code defeats 100% of attacks that jump over at least two C source code statements. Experiments showed that the resulting code is also hardened against unexpected function calls and jump attacks at the assembly code level. Securing a source code automatically and extensively with our scheme degrades the performance. The performance overhead of our countermeasures on three well-known encryption algorithms available in C ranged from $+41\%$ to $+138\%$ on an x86 platform and from $+45\%$ to $+217\%$ on an ARM-v7 platform. However, combining code rewriting with hardening of sensitive code regions identified by the weakness detection step enables an application to be fully hardened while limiting the overhead.

Keywords: physical attacks, smart card, control-flow integrity, code securing, countermeasures

1. Introduction

Smart cards or, more generally, secure elements are essential building blocks for many security-critical applications. They are used for securing host applications and sensitive data such as cryptographic keys, biometric data, and pin counters. Malicious users aim to obtain access to these secrets by performing physical attacks on the secure elements. Fault attacks, which the current work focuses on, consist in disrupting the circuit's behavior by using a laser beam or by applying voltage, clock, or electromagnetic glitches [1, 2]. The goal is to alter the correct progress of the algorithm and, *e.g.*, by analyzing the deviation of the corrupted behavior with respect to the original one, to retrieve the secret information [3].

Many protection schemes have therefore been proposed to counteract fault attacks. Fault detection is generally based on spatial, temporal, or information redundancy at the hardware or software level [4]. Moreover, to render attacks more difficult, some schemes may add noise, for example, by inserting dummy operations [5] or by masking critical internal computations [2].

In practice, developers need knowledge of the target code vulnerabilities to locate code portions for securing and limiting the overhead of software protection. Then, software countermeasures are manually added to the code in a trial-and-error process: the security of the resulting

hardened code is analyzed by simulating or performing physical attacks. If the system is not considered secure enough, the code returns to the securing step. The securing process starting with the weakness determination is time consuming, with a direct impact on the cost and delivery of the product. Therefore, automation tools are crucial to substantially reduce the required development time.

The weakness detection and code securing processes can be performed at different code levels: source code [6], any intermediate code such as during the compilation process [7, 8], or binary code [9]. Working at the assembly or binary code level seems to be the most adapted level when considering physical faults. However, identification of weaknesses requires determining the faults that have an impact on the program security to understand their effect for adding a protection code to counteract them. At these levels, high-level information such as sensitive variables and control flow is not directly available owing to loss of information to code transformations and code optimization during the compilation process [10, 11, 9]. Matching assembly instructions with the source code to retrieve semantics about high-level elements is not trivial; moreover, sometimes, it is impossible and time consuming if manually performed. Further, weakness detection first requires enumerating all potential attacks, which is also a lengthy process or even virtually impossible to perform by simulation or with an experimental campaign. Furthermore, assembly programs are tightly coupled to specific architectures and simulating attacks at the assembly code level strongly limits the portability of the analysis. Hence, for practical and time-to-market reasons, it is more convenient to simulate attacks and perform vulnerability analysis at the source code level. Simulation of attack injection at the source code level speeds up the detection of weaknesses compared to injection at the assembly code level owing to the smaller number of source statements. It aggregates several physical attacks at a more coarse-grained level [12, 13]. Faults modeled at this level may not have a direct correspondence to faults at a lower level. However, it enables a faster analysis by offering a direct match with the impacted high-level elements. Thus, developers can efficiently gain knowledge of code weaknesses to delimit the code regions to be protected afterward.

Moreover, secure smart cards have strong security requirements that must be certified by an independent qualified entity before being placed on the market. Certification is most often based on the review of the code and the implemented software countermeasures. A full system certification then relies on a certified source code compiled by a certified compiler that runs on a certified hardware target. In this specific case, injecting countermeasures at compile time or even later would require either to review the code produced by the modified compiler or any tool or to certify the modified compiler or the tool itself. Despite the recent proposals on hardening approaches at compile time or at the binary code level [8, 9, 14], for portability concerns and owing to the difficulty related to the certification, countermeasures are still preferably designed and inserted at a high code level. Therefore, the industry is in demand of solutions to reduce the time to harden a source code.

In this paper, we propose a full methodology to answer these needs. We consider control-flow disruption, which is a harmful consequence of fault attacks that encompass both instruction skip faults and larger jumps that are easily performed with physical fault injection means. Such control-flow disruption enables an attacker to retrieve secret keys [15, 3], bypass certain implemented countermeasures [6], or obtain certain unauthorized privileges [16, 17]. Our approach, which is supported by tools and is illustrated in Figure 1, enables 1) the automatic detection of the weaknesses of native C programs to be embedded into a secure element (weakness detection is performed at the source code level and considers attacks that disrupt the control flow) and 2) the automatic injection of formally verified countermeasures at the granularity of single C statements.

In the first step of our methodology, the set of harmful attacks is determined through an exhaustive search of weaknesses. This step relies on simulations followed by a classification of attack effects. The classification uses a distinguisher to determine whether the functionality delivered by an attacked execution of the application could alter the security. The identified harmful attacks can then be visualized spatially to identify the affected functions and to precisely locate the corresponding sensitive code regions. This weakness detection scheme helps developers to gain knowledge of their application implementation and enables them to implement adequate software countermeasures.

In the second step of our methodology, a tool automat-

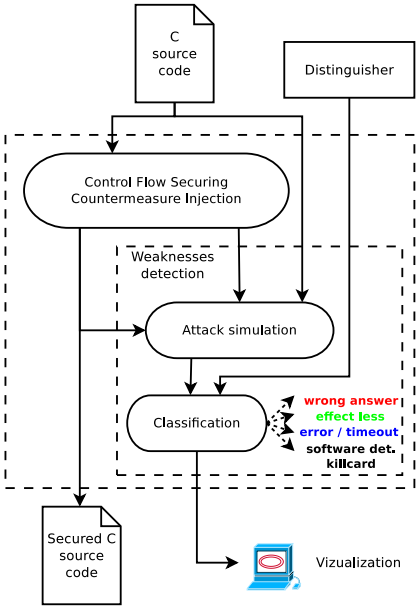

Figure 1: Code securing methodology

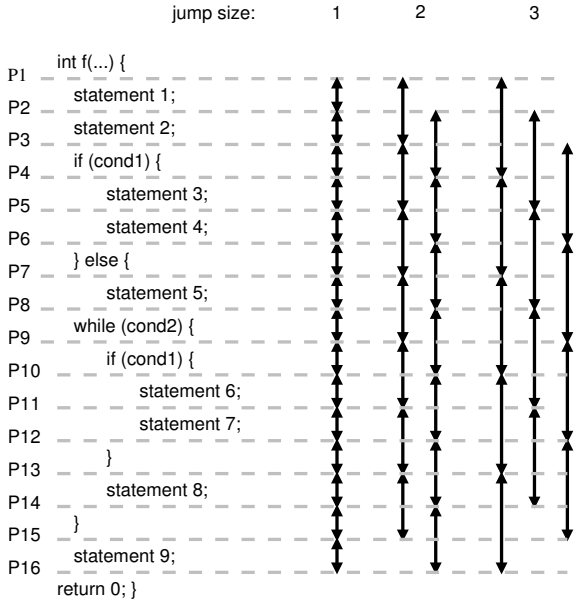

Figure 2: Injection of jump attacks at the C code level

ically injects countermeasures into the code to be protected without any direct intervention of a developer. The countermeasure scheme proposed in this paper operates at the function level and protects the control-flow integrity (CFI) at the granularity of single C statements. Protection schemes have been designed for hardening the control flow of functions (even recursive ones), conditionals and switch constructs, and even loops with break and continue statements. Countermeasures rely on counters that are incremented and verified in a nested way throughout the execution, enabling an early detection of any attack that disrupts the control flow by not executing at least two adjacent statements of the code. A securing scheme with deferred detection is also proposed to reduce the cost of detection: counters are still incremented in an imbricated manner; however, detection is performed at the boundaries of control-flow blocks.

The effectiveness of the countermeasures for both early and deferred detection has been formally verified. These countermeasures fully defeat any attack that impacts the control flow by jumping forward or backward by more than one C statement. Their effectiveness against the jump attack fault model was confirmed by experimental results. Nevertheless, they do not deal with data attacks. Experimental results also showed that 1) attacks are much more difficult to perform on the secured code; 2) attacks attempting to call an unexpected function are detected; and 3) the full methodology enables an application to become robust.

Parts of this work were presented, in preliminary form, at the 19th European Symposium on Research in Computer Security [18]. This paper provides additional content compared to the previous one. First, the countermeasures have been extended to handle more constructs: loops with break and continue statements, and switch. As the previous ones, they have been formally verified. New experiments were conducted on the FISSC benchmark [6] (two versions of the VerifyPIN code), at both the C code and the assembly code level.

The rest of the paper is organized as follows. Section 2 discusses the related works. Section 3 provides details on the detection of weaknesses and visualization. Sections 4 and 5 present the countermeasure schemes for hardening a code against control-flow attacks and the formal approach used for verifying their correctness, respectively. Section 6 presents the results of the experiments on three

cryptographic codes, namely AES, SHA, and Blowfish, as well as of a case study on an authentication code.

2. Related work

Control-flow integrity has been widely studied since the seminal work of Abadi et al. [19], in various contexts of threat, and is still currently a hot topic of research [20, 21, 22, 23]. As we focus on a smart card subject to physical attacks, it induces a specific attacker model and requires specific solutions. Thus, we first present our attacker model, and consequently, the considered fault model. Then, we briefly discuss previous research works on weakness detection before presenting the proposed protection schemes to ensure CFI.

2.1. Physical attacks and fault models

The attacker considered in this work is powerful as he or she can perform physical attacks on a secure element such as a smart card. Performing physical attacks and understanding their precise effects or the way to conduct them efficiently are active research areas. One part of this research work is dedicated to fault attacks. Simulating these faults to evaluate the robustness of a target system and designing protection schemes subsequently require fault models. This requires analyzing the consequences of physical attacks and model them at the desired level (architectural level, assembly code level, or source code level).

At the assembly code level, the link between an attack and its consequences is easier to establish. As an example, [24] studied the effect of electromagnetic pulse injections at the architectural level and proposed a fault model at the assembly code level. They showed that it can provoke a clock glitch during a transmission on the flash memory bus, corresponding to an instruction fetch or a data read. It can be modeled at the assembly code level by an instruction replacement or an erroneously loaded data value. In [25], such a model was used with a machine model for executing an assembly language program in a model checker to verify whether the detectors can handle the attack. These low-level models help to understand the consequences of a successful attack, although they are difficult to use at a high level [12]. For example, a security expert must understand whether an attack has an impact at the protocol level or on the cryptographic primitives [26]. This distinction can only be made if the expert has access to the semantics of the studied program.

Several recent works have studied the consequences of fault attacks at the program level [12, 7, 27]. For a corrupted execution, such studies require understanding which variables or control-flow constructs or conditions have been impacted. Then, they require to either deduce a high-level model or to understand a successful attack. In [12], we discussed the difficulty of understanding the link between a low-level attack and its consequence at a high level. Although some fault models are commonly admitted (*e.g.*, instruction skip, corrupted variable, etc.) and used at different levels, their link or their feasibility is not obvious. In [28], the authors proposed a methodology to infer a fault model by considering the probability of fault occurrences given the target platform and the attacker's equipment. In [29] (pp. 125–126), the author compared some fault models at a high and a low level. The results showed that, for each attack considered at a high level, there was at least one low-level attack that explained it. The author also reported that attacks with multiple conditional test inversions considered and investigated at a high level could not be reproduced at a low level owing to the computational cost. This advocates for high-level models that can considerably reduce the vulnerability search space.

Multiple physical faults can be used to perform complex attacks, and recent works have proposed to analyze multiple injections of faults by simulation [27]. Producing two faults in a short period of time would eventually help to attack the program and the countermeasures at the same time. Nevertheless, it is necessary to have an additional equipment such as a second laser beam to avoid the reloading time of the first one. In such a configuration, handling multiple antennas placed on the same x, y coordinate or using one antenna with a reloading time would result in additional difficulties for the attacker. Therefore, in this work, we hypothesize that only one fault is produced.

2.2. Control-flow attacks

In this work, we focused on physical attacks that impact the control flow of native C programs. Several works [3, 30] showed that attacks can induce instruction

replacements. Such replacements can provoke a control-flow disruption in the two following cases:

1. A whole instruction is replaced by a jump at any location of the program. The executed instruction becomes a jump to an unexpected target [31, 3]. The same effect is obtained if the target address of a jump is changed by corrupting the instruction encoding or, in the case of an indirect jump, if the computation of the target address is disrupted. This also happens if the program counter (PC) becomes the destination operand of the replacement instruction [16]; *e.g.*, arithmetic-logic unit instructions such as `PC = PC +/- cst` are the most likely to succeed into a correct jump. Moreover, other instruction replacements can provoke an arbitrary jump into the code [3].
2. The evaluation of a condition is altered by the replacement of one instruction involved in the computation, causing the wrong branch to be taken. Inverting the test of a conditional branch instruction by only replacing the opcode in instruction encoding has the same consequence and is covered by the first case.

In this work, we considered jump attacks as described in the first case mentioned above. Jump attacks can be induced by either the replacement of any instruction by a jump instruction or the replacement of an operand of a legacy jump leading to a modification of the target address.

2.3. Weakness detection

Code securing techniques can be applied to the entire application or only to specific parts. Securing only sensitive code regions requires discovering weaknesses that need to be strengthened for a given fault model. When considering convenient fault models or when varying inputs, tractable static analysis, such as taint analysis, can be used to infer the impact of a fault on control flow [32] or detect missing checks [33].

An automatic detection method proposed by Rauzy et al. focuses on the vulnerability assessment of cryptographic algorithms submitted to fault attacks. In [34], existing and optimized secured versions of the CRT-RSA algorithm were studied. The considered fault models included fault on data and skip attacks. The authors showed that, on such a precise code, skip attacks can be detected by observing the data manipulated by the algorithm.

To the best of our knowledge, no previous work considered a jump attack fault model for weakness detection. This fault model faces a potential combinatorial problem: all possible jumps from one point of the program to another point must be considered. Moreover, jump attacks may occur at each execution time of the source location. Thus, the level of threat induced by such attacks is difficult to statically predict. Nevertheless, simulation of such attacks can cover a substantial subset of all possible jumps at the cost of a longer simulation time. In this work, we modeled faults at the source code level, and thus, there were fewer attacks to consider. In addition, we leveraged profiling information to determine a representative subset of attacks for simulation.

In [35], the authors distinguished between full software simulation of attacks [12, 7, 27] that provides more control of the manipulated code and onboard approaches that model attacks considering the target hardware [36, 35]. Such approaches are not exclusive: in [35], the authors proposed to simulate a high-level fault, identified at the source code level, inside a smart card component. If such a component supports the embedding of a fault injection mechanism, the simulation occurs on the real target; however, it is very slow because of the limited available computational power.

Some recent papers [13, 7] studied a subset of all possible jump attacks where conditional tests can be inverted. Riviere et al. determined the harmful attacks using a concolic analysis of the control-flow graph of the source code. The proposed approach does not require simulating attacks to evaluate their impact but is limited to small jump attacks (conditional inversion). The jump attacks we considered in this work encompassed this case.

2.4. Code integrity and control-flow securing

Protections against control-flow attacks depend on the nature of the attacks. If the evaluation of a condition involved in a conditional branch is disrupted at runtime, recovering techniques must strengthen the condition computation. This can be achieved by temporal duplication at the software level by executing twice the algorithm or each instruction of the whole algorithm or of its sensitive parts [37, 4].

Countermeasures that protect against forward jumps of only one assembly instruction (instruction skip) should use duplication techniques at the assembly code level [8, 4, 38]. Nevertheless, such solutions require analyzing the assembly code to find available registers [4, 38]. Barry et al. showed how a securing compilation pass can help solve this problem [8]. Such a duplication scheme could be extended to protect against large skip attacks; however, this would induce a very high overhead.

Countermeasures designed for ensuring CFI can be split into two classes depending on the attack model. If the attacker cannot modify the application code but can only inject code (shell code) and/or write data, protection schemes must only ensure the control-flow graph integrity. In this case, countermeasures typically rely on checks to ensure the validity of the target address of jump instructions, particularly of indirect jumps and calls. Many of the previous and recent works on CFI targeted code-reuse attacks such as return-oriented attacks [39] (ROP) and jump-oriented attacks (JOP) [22, 23]. Hence, they only need to ensure the control-flow graph integrity such as the seminal method proposed by Abadi et al. [19], which checks for both the source and the destination of indirect jumps. This approach relies on a new machine instruction that manipulates an operand ID used to encode the legacy of control-flow transfer. Bletsch et al. [40] introduced a new technique for code-reuse attacks called control-flow locking: before a control-flow transfer, a code snippet locks a value of the memory and unlocks it after the jump. Stack canaries and shadow stacks are also useful for detecting bad return addresses [41], although not sufficient as a standalone protection for code-reuse attacks [22]. For such attacks, protection schemes have a low performance overhead because the countermeasures only need to be inserted at the source and at the target of possible indirect jumps. These solutions cannot address the problem of jump attacks from and to points inside basic blocks.

Code integrity is a security property to guarantee when an attacker can modify the application code. Protection schemes are most often based on signature techniques, which typically rely on an offline computation of a checksum for each basic block. At runtime, the protected code recomputes the checksum of the basic block being executed and compares it with the expected result. This extra computation can also be handled by another software thread or by a dedicated hardware component. Thus, a control-flow attack that leads to the execution of only a subpart of a basic block will be detected. Several solutions based on dedicated hardware have been proposed [42, 43, 44, 21, 20]. Although they have a lower overhead, they require hardware modifications, and thus, are impractical for smart cards based on off-the-shelf hardware. Hence, several works address pure software solutions for these types of countermeasures [45, 46, 47]. In [45], the authors performed signature checks at the destination of jumps between basic blocks using an extra watchdog thread. The check verifies whether the source basic block has a valid signature, *i.e.*, if it is known and has kept its integrity. The YACCA approach [47] for detecting hardware faults relies on specific signature computations performed throughout the execution and on checks performed at basic block boundaries for performance reasons. The main drawback of all these solutions is their inability to detect when a valid target of a conditional branch is badly taken during execution. The protection schemes proposed in this paper address this problem by capturing conditions involved in the control flow and by using them to detect control-flow attacks.

We identified two previously proposed approaches that use a step counter to protect a code region [48, 49]. The former targets computation disruption, whereas the latter combines counters with a signature approach at the assembly code level to ensure tolerance to hardware faults. Our approach, based on counters, is similar to the intra-basic block approach of [49] for securing sequential code but operates at a higher code level and can harden the control flow of high-level constructs. Hardening the control flow at a high level allows to use a certified compiler and produce a certified binary program. It provides an important security guarantee for the smart card industry, to obtain levels of certification, *e.g.*, EAL4+ for mobile SIM cards that embed payment, TV, and identity applications [50]. Moreover, in [48, 49], the effectiveness of the countermeasures was not evaluated. In this work, we not only formally verified the countermeasures but also experimentally evaluated their robustness.

Finally, in [51], a CFI countermeasure at the C code level was presented. Nevertheless, its applicability is limited to loops with a constant number of iterations. The countermeasures proposed in this paper can be applied to any type of loop (e.g., for loop; while loop, possibly

with break or continue statements; etc.), if-then-else, and switch constructs.

In summary, approaches that verify the source or the target of branches or jumps only harden the CFI at basic block boundaries. This is not sufficient to cover test inversion of conditional branches and intra-basic block jumps caused by physical fault attacks. The approach proposed in this paper enforces the CFI at the granularity of single C statements. Additionally, in the specific context of smart cards or secure elements, to the best of our knowledge, no research work has proposed formally verified and experimentally evaluated countermeasures at the C code level that ensure CFI during native execution in the presence of jump attacks.

3. Detection of weaknesses and visualization

In this section, we describe the part of our methodology that identifies harmful attacks. The identification of weaknesses is carried out by simulating, classifying, and visualizing the effects of physical attacks at the source code level.

3.1. Simulation of attacks

To discover harmful attacks, we simulate jump attacks by using software hacks at the C code level, as proposed in [12]. First, we generate a set of so-called attacked codes as explained below. For each function of the application, for each pair of lines (i,j), where $i \neq j$, of a function, we generate a C code corresponding to the original source code in which a jump attack from line i to line j has been injected. Thus, we generate as many attacked codes as all possible intra-procedural jump attacks that jump backward or forward C statements. Figure 2 illustrates all possible jumps within a function, sorted according to their jump distance expressed in statements. Statements in this context are C statements such as assignments, conditional expressions (*e.g.*, `if (cond1)` or `while(cond2)`) and also any bracket or syntactic elements (*e.g.*, `}else{`) that have an impact on the control flow. For example, the bracket between P14 and P15 in Figure 2 corresponds to the return to the beginning of the while loop (P8-P9). Jumping this bracket or jumping the `}else{` between P16 and P17 breaks the CFI of the loop or of the if-then-else construct.

Finally, the simulation campaign consists in executing the generated variant including the intra-procedural jump attack. As stated in Section 2.3, triggering attacks at any possible time of the execution would result in an explosion of execution tests. As an example, jump attacks that come from inside a loop whose execution count depends on an input could not be fully simulated without any assumption on the input. Thus, to cover a representative subset of jump attacks that could be triggered at different moments of the execution, we audit the execution count of each line of the original program for representative inputs. We then generate, for each attack starting from a line, as many simulations as the number of execution of this line. Thus, we obtain, for each line i of the code, its execution counts denoted as N_i. Then, for each line j of the function where line i belongs to, we simulate an attack that jumps from line i to line j at N_i different moments of the execution.

The outputs of all the simulations are provided to the classification tool. The tool analyzes them according to a distinguisher to produce the result of the weakness detection, *i.e.*, the set of harmful attacks with corresponding jump characteristics.

3.2. Classification of simulated attacks

The benefits of an attack differ depending on the application and the context of its use. A successful attack may break data confidentiality (by forcing a leak of sensitive data such as an encryption key or a PIN code) or may break the integrity of an application (by corrupting an embedded service). To cover the various benefits for an attacker in a general way, our methodology requires a distinguisher to be provided. The distinguisher is used to separate attacks according to their effects from a security point of view, *i.e.* harmful attacks (the *Wrong answer* class) from the harmless one (*Effect-less* class). A finer classification of the effects of an attack can be achieved by providing a more precise distinguisher. In the remainder of the paper, we consider four different classes, similarly to [47]:

- *Wrong answer* (WA): During execution, a benefit has been obtained by the attacker.
- *Effect-less* (EL): The behavior of the application remains unchanged.

- *Error* or *Timeout* (TO): The program does not seem to have terminated and has to be killed or finished with an error message or a signal (SIGSEGV, SIGBUS, etc.), or may have crashed.

- *Software detection* or *killcard* (SD): A countermeasure has detected an attack and triggered a security protection.

We assume that no benefit can be obtained by a crashing or endless execution. Thus, both *Error* and *Timeout* cases are distinguished from *Wrong answer* cases in the remainder of the paper. If an error is preceded by a gain, such as a leak of sensitive information, or if an endless execution provides an advantage to the attacker, the distinguisher must be able to discriminate between these attack effects and classify the attack into the *Wrong answer* class. The class *Software detection* gathers attacks that have been detected by a countermeasure. We also call this class *killcard* to refer to the fact that an attack detection usually results in a disabling of the card.

3.3. Weaknesses analysis and visualization

Because our securing scheme operates at the function level, the detection of weaknesses aims at identifying harmful attacks at the source code level to identify the functions to be secured. Thus, any function that, when attacked, exhibits a *Wrong answer* case shall be considered for the countermeasure injection. The tools supporting our methodology offer a visualization tool that can be used by a security expert or a developer to 1) quickly understand which variables and functionalities are involved in the generation of harmful attacks by analyzing the jumped part of the code, 2) gain knowledge of the vulnerabilities of the code, and 3) decide whether the corresponding code is really sensitive, and hence, should be secured.

The visualization tool builds a graphical representation of the results of the weakness detection by drawing a square at the coordinate $(i = source\_line, j = target\_line)$ using the color associated with its class. If several attacks $i \rightarrow j$ triggered at different moments lead to different classification results from the distinguisher, we always choose the class with the higher impact on the security.

An example of the graphical representation, associated with its source code, is shown in Figure 3. The studied function was `aes_addRoundKey_cpy`, an extract of an implementation of AES-256 [52] in C, used later in the experiments. In this example, the distinguisher considers as *Wrong answer* any execution of the application producing incorrect encrypted data, representing the attacker's ability to disrupt the encryption. On the right-hand side of Figure 3, the attacks on the diagonal are harmless (green squares): the corresponding jumps do not jump over any statement ($i \rightarrow i$). The attacks below the diagonal correspond to backward jumps. The attacks above the diagonal correspond to forward jumps. The orange squares correspond to harmful attacks that jump only one statement, whereas the red ones stand for attacks that jump more than two statements. All but one forward jump (that jumps the whole loop body without any effects on the variables) generate a *Wrong answer* case. Analyzing the statements impacted by these harmful attacks shows that the whole loop body, and hence, the whole function, must be secured.

4. Countermeasure for securing C code

In this section, we present the countermeasures designed to detect jump attacks with a distance of at least two C statements. These countermeasures deal with different high-level control-flow constructs such as straight-line flow, if-then-else, switch, and loops. Countermeasures use the macros shown in Figure 4 and are all expanded to only one line of source code. We have two versions of these countermeasures: countermeasures that detect jump attacks early (noted as CMED for CounterMeasures with Early Detection) and countermeasures that detect jump attacks at basic block boundaries (noted as CMDD for CounterMeasures with Deferred Detection). We start by presenting the CMED and will present the CMDD later in Section 4.7.

4.1. Protection of a function and straight-line flow of statements

Our securing scheme uses a dedicated counter to secure the CFI of a whole function or a whole block of straight-line statements. Each function and each block of straight-line code have their own counters to ensure their CFI.

```
237  void aes_addRoundKey_cpy(
         uint8_t *buf, uint8_t *
         key, uint8_t *cpk)
238  {
239    register uint8_t i = 16;
240
241    while (i--)
242    {
243      buf[i] ^= key[i];
244      cpk[i] = key[i];
245      cpk[16+i] = key[16+i];
246    }
247    ;
248  }
```

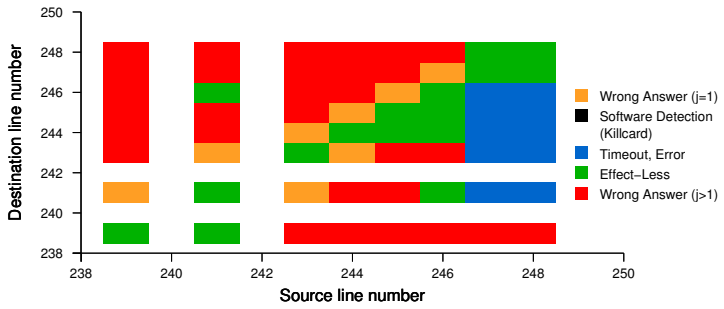

Figure 3: Result of the weakness detection for the aes_addRoundKey_cpy function

```
// STRAIGHT-LINE FLOW
#define DECL_INIT(cnt,val) unsigned short cnt = val;
#define CHECK_INCR(cnt,val) cnt = (cnt == val ? cnt + 1 : killcard());
// AFTER A FUNCTION CALL
#define CHECK_INCR_FUNC(cnt1, x1, cnt2, x2) cnt1 = \
                ((cnt1 == x1) && (cnt2 == x2) ? cnt1 + 1 : killcard());
// IF
#define CHECK_END_IF_ELSE(cnt_then, cnt_else, b, x, y) \
   if ( ! ( (cnt_then == x && cnt_else == 0 && b) \
         || (cnt_else == y && cnt_then == 0 && !b) ) )killcard();
#define CHECK_END_IF(cnt_then, b, x) \
   if (! ( (cnt_then == x && b) || (cnt_then == 0 && !b) ) )killcard();
// WHILE
#define CHECK_INCR_COND(b, cnt, val, cond) \
  (b = (((cnt)++ != val) ? killcard() : cond))
#define CHECK_END_LOOP(cnt_loop, b, val) \
  if ( ! (cnt_loop == val && !b) )killcard();
#define CHECK_LOOP_INCR(cnt, val, b) cnt = (b ? (cnt + 1) : 0);
#define RESET_CNT(cnt_while, val) cnt_while = \
  !(cnt_while == 0 || cnt_while == val) ? killcard() : 0;
```

Figure 4: Security macros used for securing the control flow

Counters have different initial values and evolve in disjoint intervals. They are incremented after each C statement of the original source code using the `CHECK_INCR` macro. Before any increment, a check of the expected value of the counter is performed. When a check fails, a handler called `killcard()`, which is the one used in the smart card community, stops the execution.

To ensure CFI, we need to nest the checks and increments of counters. Consider the example in Figure 5, which illustrates the countermeasure for a function `g` with a straight-line control flow composed of N statements. The dedicated counter `cnt_g` is declared and initialized outside the function, *i.e.*, in any function `f` calling `g` prior to each call to `g`. The initialization associated with the counter declaration is *surrounded* by two checks and increments of the counter `cnt_f` dedicated to the block of the function `f` where `g` is called. A reference to the counter `cnt_g` is passed to `g` as an extra parameter.

Moreover, the initialization value of the counter associated with the top-level region of each function is different, and as such, the ranges of counter values for all functions are separated. Using the same initial value of a counter for the two functions `f` and `g` would enable an attacker to make a fault when `f` is called. Before the call, the address of `cnt_f` is pushed onto the stack. Then, the fault could occur by forcing the control flow to execute

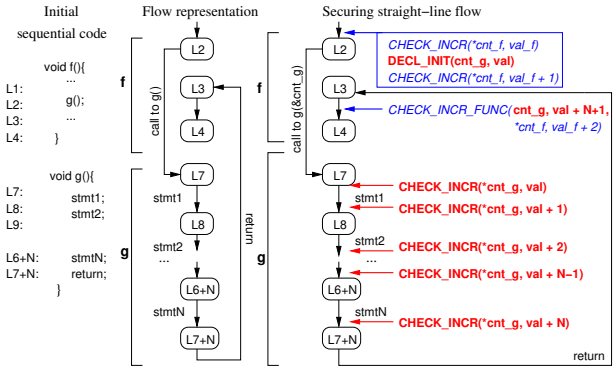

Figure 5: Securing a function call and a straight-line flow

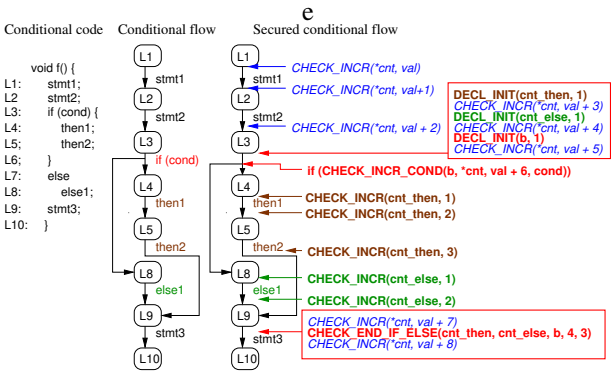

Figure 6: Securing a conditional control flow

`g` instead of `f`. Function `g` would pop the address of the counter off the stack (if the number of parameters of both functions is identical) without being able to detect the attack if the initialization values of both `cnt_g` and `cnt_f` counters were equal. As shown by the experiment results, our countermeasures prevented attacks that tried to substitute a function for another one during a call.

Upon returning from `g`, our countermeasure performs a check of the values of both counters `cnt_f` and `cnt_g` to detect any corruption of the flow inside the function `g`. This way, any jump to the beginning of the function `g` is detected inside the called function `g`. Any jump to the end of a called function is caught when the control flow returns to the calling function. The nesting of counter checks and increments is at the core of our countermeasure scheme to ensure CFI. It ensures in this case that the call has been correctly performed: any jump over the call would be detected by one of the checks on `cnt_f` in `f`, and any jump that would try to bypass the check upon returning from `g` would also be detected in `f`, owing to `cnt_f`.

4.2. Conditional if-then and if-then-else constructs

High-level conditional control flow refers to if-then or if-then-else constructs, as illustrated by the example on the left-hand side of Figure 6. The securing scheme for conditional flow is illustrated in the right-hand side of the figure. For such a construct, our securing scheme requires two counters, namely `cnt_then` and `cnt_else`, which are used to check the CFI of each conditional branch, and one extra variable `b` used for holding the value of the condition of the conditional flow. The declarations and initializations of `cnt_then`, `cnt_else`, and `b` are performed outside the if-then-else block. Similar to functions or straight-line blocks, these new statements are interlaced with checks and increments of the counter `cnt` used for the control flow of the surrounding block. This is performed by the additional statements in the red bold box on the upper right-hand side of Figure 6.

The condition evaluation in the secured version is performed through the macro `CHECK_INCR_COND`: if the counter `cnt` for the flow integrity of the surrounding block holds the expected value, `cnt` is incremented and the condition is evaluated. Thus, any jump attack over the condition evaluation is detected after the if-then-else construct, when checking the `cnt` counter. The extra variable `b` is set to the value of the condition, to be able to distinguish, after the execution of the if-then-else construct, which branch should have been taken. Both counters dedicated to the conditional branches are then checked according to the value of `b`. This is performed by the `CHECK_END_IF_ELSE` macro inserted between two checks of the counter `cnt`. Again, this interlacing of checks and increments of a counter and of one of the surrounding blocks is at the core of the effectiveness of our countermeasure scheme.

```
// Additional macros for switch construct
#define INCR_SW(n_sw, c, cnt, val) n_sw = ((cnt++ == val) ? c : killcard())
#define INCR_ASSIGN(cnt, val, cnt_sw, v1) cnt_sw += (cnt++ == val)? v1 : killcard();
#define CHECK_END_SW(cnt_sw, n_sw, val1, end_val1, val2, end_val2, end_default) \
      if (((n_sw == val1) &&( cnt_sw != end_val1))|| ((n_sw == val2) && (cnt_sw != end_val2)) \
          || (((n_sw != val1) && (n_sw != val2)) && (cnt_sw != end_default))) killcard();
```

```
// switch construct
// example

 stmt_before;
 switch(c){

  case a:
    stmt1_1;
    stmt1_2;
    break;

  case b:
    stmt2_1;
    stmt2_2;
    stmt2_3;
  break;
  default:
  stmt3_1;
  break;
 }
stmt_after;
```

```
 // securing example of switch contruct
 CHECK_INCR(cnt,v)
 stmt_before;
 CHECK_INCR(cnt, v+1)
 DECL_INIT(cnt_sw, 100)
 CHECK_INCR(cnt, v+2)
 DECL_INIT(n_sw, -1)
 CHECK_INCR(cnt, v+3)
 switch(INCR_SW(n_sw, c, cnt, v+4)) {
 case a:
  INCR_ASSIGN(cnt, v+5, cnt_sw, 1000)
  CHECK_INCR(cnt_sw, 1100)
  stmt1_1;
  CHECK_INCR(cnt_sw, 1101)
  stmt1_2;
  CHECK_INCR(cnt_sw, 1102) // mandatory for CMDD
  break;
 case b:
  INCR_ASSIGN(cnt, v+5, cnt_sw, 2000)
  CHECK_INCR(cnt_sw, 2100)
  stmt2_1;
  CHECK_INCR(cnt_sw, 2101)
  stmt2_2;
  CHECK_INCR(cnt_sw, 2102)
  stmt2_3;
  CHECK_INCR(cnt_sw, 2103) // mandatory for CMDD
  break;
 default:
  INCR_ASSIGN(cnt, v+5, cnt_sw, 3000)
  CHECK_INCR(cnt_sw, 3100)
  stmt3_1;
  CHECK_INCR(cnt_sw, 3101) // mandatory for CMDD
  break;
 }
CHECK_INCR(cnt, v+6)
 CHECK_END_SW(cnt_sw, n_sw, a, 1103 /*end case 1*/, b, 2104 /*
     end case 2*/, 3102 /*default*/)
CHECK_INCR(cnt, v+7)
 stmt_after;
```

Figure 7: Principle for securing a switch construct

4.3. Switch

The principle of the securing scheme we designed for switch constructs is illustrated in Figure 7. As for the conditional construct, an extra variable and a counter are needed to protect the control flow of the switch construct. The extra variable `n_sw` is assigned to the value of the variable that controls the switch (macro `INCR_SW`). In each case of the switch, the counter `cnt_sw` is first incremented by a specific value after a check on the surrounding counter, owing to the macro `INCR_ASSIGN`. The specific value is chosen for each case to prevent the ranges of values for the counter `cnt_sw` in each case from overlapping with each other. After the switch construct, the macro `CHECK_END_SW` checks the value of the counter `cpt_sw` accordingly to the chosen case held in the extra variable `n_sw` (the value `a`, the value `b`, or of any other values in the example). All cases end with a break statement in the given example, although cases without a break statement are supported: the expected final values of the counter of such cases should be adapted to perform a correct check (macro `CHECK_END_SW`); the securing principle, however, is identical.

4.4. Loop constructs

We also designed a countermeasure scheme for loops. As all the forms of loops (e.g. for loops or do-while loops) can be rewritten as a while loop, we only present the while loops. The left-hand side of Figure 8 shows a while loop and the corresponding control flow between statements `stmt_1`, `stmt_2`, and `stmt_3` of the surrounding sequential code. Our countermeasure scheme uses one counter, `cnt_while`, for securing the control flow of the loop body. Similar to conditional constructs, our countermeasure scheme requires an extra variable `b` to hold the value of the loop condition. This variable is needed at the end of the loop to verify the correct execution of the

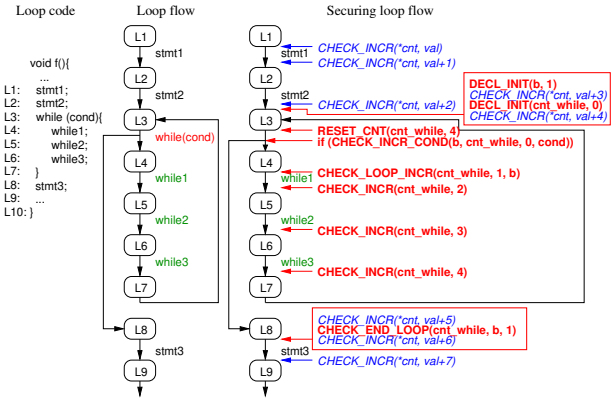

Figure 8: Securing a loop control flow

loop body and the correct termination of the loop. This is performed by the `CHECK_END_LOOP` macro, which is surrounded by the `CHECK_INCR` macro of the counter `cnt`. The `b` variable is declared and initialized outside the loop like in the other constructs. The initial value must be `true`: if an attack jumps over the loop, `b` holds `true` and the `CHECK_END_LOOP` macro, checking for `b` being `false` after the loop, detects the attack. The `cnt_while` counter is reset before each iteration using the `RESET_CNT(cnt_while, val)` macro, with `val` being the expected final value of the counter after one complete iteration. The reset is performed only if `cnt_while` is equal to 0 or to the value of `val`. As a jump from the end of the loop to the beginning of the body would result in a correct value for `cnt_while` that is reset before each new iteration, the first check inside the loop body of the while counter is guarded with `b` to detect such a jump attack, leading to an additional iteration of the loop. Moreover, evaluation of the condition (that may update an induction variable) is performed along with a check and an increment of the counter `cnt_while` using the `CHECK_INCR_COND` macro. Hence, any attack that jumps over the evaluation of the condition of the loop will then be detected inside the loop. This combination of additional statements, as validated by our formal verification presented in the next section, enables to ensure that the right number of iterations, each with a right control flow, is performed or that an attack is detected.

4.5. Continue and break statements in loops

Figure 10 contains several listings to illustrate the principle of the securing scheme for loops containing a continue or a break statement. Additional macros needed to protect these constructs are presented in Figure 9. First, for both cases, the extra Boolean (`b-break` or `b-cont`) used for the if-construct that guards the break or continue statement is required at the end or at the beginning of the loop to determine whether the break or the continue statement should have been executed or not. Any break or continue statement is removed to be able to check the construct that contains such statement (an if construct in the example, but any other control-flow block in the general case) and determine whether the statements inside the if-break or if-continue construct have been rightly executed. The exit of the loop (break) or the return to the beginning of the loop (continue) is then performed if needed. Special macros, namely, `CHECK_END_IF_CONTINUE` or `CHECK_END_IF_BREAK`, are devoted to these tasks. Moreover, for a continue statement, the macro checking the value of the loop counter in the header of the loop is changed slightly: it checks the value of the loop counter, which can be the initialization value, the value at the end of the body, or the value just before looping back to the header of the loop. This last case is determined by using the Boolean `b-cont`. For a break, at the exit of the loop, the loop counter value is also checked according to an extra Boolean `b-break`. If it holds a true value, then the counter must have the value set just before the `CHECK_END_IF_BREAK` macro.

4.6. Discussion on the limitations of countermeasures

The first limitation concerns the use of function pointers. This is a common case in the smart card code or system code. As recently shown by Arthur et al. [23], it is possible to rewrite the code to translate any indirect call (*i.e.*, a call through a function pointer) to a set of direct calls. This requires having the whole set of functions that may be called at each call site that uses pointers. In the general case, such analysis would fail statically because complex codes can even deploy new functions at execution time. However, for smart card native software, all functions of the embedded software are identified and can be statically determined. As a common case, function pointers are used in smart cards for choosing one of

```
// extra macro for in-loop continue statement
#define RESET_CNT_CONTINUE(cnt, val1, condition, val2, init_value) \
 cnt = !( \
   cnt == init_value || \
   (condition && (cnt == val2)) || \
   (!condition && cnt == val1) ) ? killcard() : init_value;
#define CHECK_END_IF_CONTINUE(cnt, b, val, initv) \
 if ( (b || cnt != initv) && ((( cnt != val && b) || (!b && cnt != initv)) ? killcard() : 1) ) \
     goto start_while;
// extra macro for in-loop break statement
#define RESET_CNT_BREAK(cnt_while, val1, val2) \
 cnt_while = !(cnt_while == val1 || cnt_while == val2) ? killcard() : val1;
#define CHECK_END_IF_BREAK(cnt, b, val, initv) \
if ( (b || cnt != initv) && \
  ((( cnt != val && b) || (!b && cnt != initv)) ? killcard() : 1) ) \
  goto end_while;
#define CHECK_END_LOOP_BREAK(cnt_loop, b, val1, b_break, val2) \
if (!(cnt_loop == val1 && !b && !b_break || b && cnt_loop == val2 && b_break))\
 killcard();
```

Figure 9: Extra macro for securing a loop with continue or break statements

the several offered functionalities by setting a pointer in a switch construct. Thus, this rewriting technique can be deployed: for example, using pragmas given by the developers, the process could be automated at the source code level. Once this technique is applied, our securing scheme can be deployed. Such rewriting avoids a coarse-grained CFI approach that would set the same initialization value for all counters of functions that could be a target of an indirect call as well as set the same ending value for these counters [22, 19]. As previously explained and as shown in a recent work, coarse-grained CFI approaches are not secure [22].

Another limitation is related to the content or the size of C statements owing to the statement granularity of our approach. First, as a C statement can be arbitrarily long, a single line of C can be compiled into a long sequence of assembly language instructions. This case reduces the efficiency of the countermeasures inserted between C lines. Second, a C statement can contain jumps, *e.g.*, the expression `cond ? a : b`. Such conditional expressions would not be addressed by our methodology. Nevertheless, for both cases, a simple solution is to automatically decompose the corresponding C lines into several ones. This is compatible with the current implementation, where a C parser inspects recursively any C statement to find function calls that have to be modified to add a new parameter (the counter). Thus, detecting and rewriting complex and long C lines that may contain conditional expressions would be feasible.

Other possible limitations of the countermeasures may come from the compilation process. As pointed out by recent works [8, 14], enabling compilation optimization that is too aggressive may damage the countermeasures. To enable such compiler optimization, developers must use tricks [14] (as an example, a variable susceptible to being optimized out can be declared as volatile). Optimizing the code while ensuring that code relative to security protection is not optimized out nor degraded is an open challenging problem that requires revisiting the entire compilation process. It is out of the scope of this paper. Disabling optimization makes the compiler respect the protection code. Currently, it is a last resort to compile without any optimization of the parts of the code whose security is a major concern. In the experiments, we disabled the optimization and the results showed that the countermeasures were still effective in the compiled program.

Finally, we reiterate the ability of our countermeasures to handle control-flow attacks but not data-only attacks. Thus, several physical attacks that would corrupt the content of a variable of the program would not be detected. Such attacks on sensitive variables or values must be protected with specific protection schemes for data.

4.7. Early and deferred detection

Our securing scheme checks counter values between each functional statement or extra securing macros.

```
// a loop with continue statement
 stmt1;
 stmt2;
 while (cond ! = 0) {
  cond --; // stmt_while1
  stmt_while2;
  if (continue_condition){
    stmt_if_cont1;
    continue; }
  stmt_while3; }
 stmt3;
```

```
// a loop with a break statement
 stmt1;
 stmt2;
 while (cond ! = 0) {
  cond --; // stmt_while1
  stmt_while2;
  if (break_condition){
    stmt_if_break1;
    break; }
  stmt_while3; }
 stmt3;
```

```
 // securing a loop with a continue statement
 CHECK_INCR((*cpt), v)
 stmt1;
 CHECK_INCR((*cpt), v+1)
 stmt2;
 CHECK_INCR((*cpt), v+2)
 DECL_INIT(b, 0) CHECK_INCR((*cpt), v+3)
 DECL_INIT(b_cont, 0) CHECK_INCR((*cpt), v+4)
 DECL_INIT(cnt_while, init_val)
 CHECK_INCR((*cpt), v+5)
 start_while: {
  RESET_CNT_CONTINUE(cnt_while, init_val+10, b_cont,
       init_val+8, init_val)
  if (!CHECK_INCR_COND(b, cnt_while, init_val, (cond
       != 0)))
    goto end_while;
  CHECK_LOOP_INCR(cnt_while, init_val+1, b)
  cond --; // stmt_while1
  CHECK_INCR(cnt_while, init_val+2)
  stmt_while2;
  CHECK_INCR(cnt_while, init_val+3)
  INIT(b_cont, 0)
  CHECK_INCR(cnt_while, init_val+4)
  DECL_INIT(cnt_if_cont, init_cont)
  CHECK_INCR(cnt_while, init_val+5)
  if (CHECK_INCR_COND(b_cont, cnt_while, init_val+6,
       continue_condition)) {
    CHECK_INCR(cnt_if_cont, init_cont);
    stmt_if_cont1;
    CHECK_INCR(cnt_if_cont, init_cont+1);
  }
  CHECK_INCR(cnt_while, init_val+7) //+8 -- end value
       if continue
  CHECK_END_IF_CONTINUE(cnt_if_cont, b_cont,init_cont
       +2, init_cont)
  CHECK_COND_INCR(cnt_while,init_val+8,!b_cont)
  stmt_while3;
  // mandatory for CMDD:
  CHECK_INCR(cnt_while, init_val+9) // +10 -- end
       value of loop
  goto start_while;
 }
 end_while:
 CHECK_INCR((*cpt), v+6)
 CHECK_END_LOOP(cnt_while, b, init_val+1)
 CHECK_INCR((*cpt), v+7)
 stmt3;
```

```
 // securing a loop with a break statement
 CHECK_INCR((*cpt), v)
 stmt1;
 CHECK_INCR((*cpt), v+1)
 stmt2;
 CHECK_INCR((*cpt), v+2)
 DECL_INIT(b, 0) CHECK_INCR((*cpt), v+3)
 DECL_INIT(b_break, 0) CHECK_INCR((*cpt), v+4)
 DECL_INIT(cnt_while, init_val)
 CHECK_INCR((*cpt), v+5)
 start_while: {
  RESET_CNT_BREAK(cnt_while, init_val, init_val+10)

  if (!CHECK_INCR_COND(b, cnt_while, init_val, (cond
       != 0)))
    goto end_while;
  CHECK_LOOP_INCR(cnt_while, init_val+1, b)
  cond --; // stmt_while1
  CHECK_INCR(cnt_while, init_val+2)
  stmt_while2;
  CHECK_INCR(cnt_while, init_val+3)
  INIT(b_break, 0)
  CHECK_INCR(cnt_while, init_val+4)
  DECL_INIT(cnt_if_break, init_break)
  CHECK_INCR(cnt_while, init_val+5)
  if (CHECK_INCR_COND(b_break, cnt_while, init_val+6,
       break_condition)) {
    CHECK_INCR(cnt_if_break, init_break);
    stmt_if_break1;
    CHECK_INCR(cnt_if_break, init_break+1);
  }
  CHECK_INCR(cnt_while, init_val+7) // +8 -- end
       value if break
  CHECK_END_IF_BREAK(cnt_if_break, b_break,
       init_break+2,init_break)
  CHECK_COND_INCR(cnt_while,init_val+8,!b_break)
  stmt_while3;
  CHECK_INCR(cnt_while, init_val+9) // +10 -- end
       value of loop
  goto start_while;
 }
 end_while:
 CHECK_INCR((*cpt), v+6)
 CHECK_END_LOOP_BREAK(cnt_while, b, init_val+1,
      b_break, init_val+8)
 CHECK_INCR((*cpt), v+7)
 stmt3;
```

Figure 10: Securing a loop with continue or break statements

```
#define INCR(cnt) cnt = cnt + 1;
#define INCR_COND(b, cnt, cond) (++cnt && (b = (cond)))
```


Figure 11: Additional security macros used for securing the control flow

Checking the value of counters, to eventually jump to an appropriate detection handler (*e.g.* killcard), has a very high impact on performance. Nevertheless, checking the counters between each original statement enables to trigger an attack detection handler as soon as possible. We can relax the frequency of these checks and perform counterchecks only at the boundaries of control-flow blocks. This approach detects a bad counter value either at the end of a control-flow block or in the calling function. This optimization has also been proposed for hardware fault tolerance [47]. In this case, the detection is performed as late as possible, but the overhead of the countermeasures is reduced.

Adapting the previous countermeasure scheme to make it lighter is simple. We just need to remove the check from two macros used in the early securing scheme, namely, the `CHECK_INCR` macro and the `CHECK_INCR_COND` macro, which become the `INCR` macro and the `INCR_COND`, respectively, as shown in Figure 11. However, for switch constructs, the last increment of the counter in each case must still be performed with a check (see the comment `// mandatory` in Figure 7). This prevents any jump from inside an executed case to the middle of another one from still matching the value expected at the end of the correct expected case. Moreover, for break and continue statements, when they are used in a deep imbrication of control-flow constructs inside a loop, all intermediate counters used in these constructs must be checked before exiting the loop or looping to its beginning. All other macros remain unchanged.

4.8. Example of the results for the aes_addRoundKey_cpy function

Before quantifying the effectiveness of our countermeasure schemes in Section 6, we qualitatively show in Figure 12 the impact of our countermeasures compared with the graph of Figure 3. Note that the number of C statements has increased because of the countermeasures. Figure 12 shows that the number of jump attacks of size greater than or equal to two decreased from 30 to 0. In the secured version, five out of the eight possible attacks of a jump distance equal to 1 remained. The five attacks jump over the following:

395 → 397: `register uint8_t i = 16;` // initially 239 → 241

407 → 408: `buf[i] ^= key[i];` // initially 243 → 244

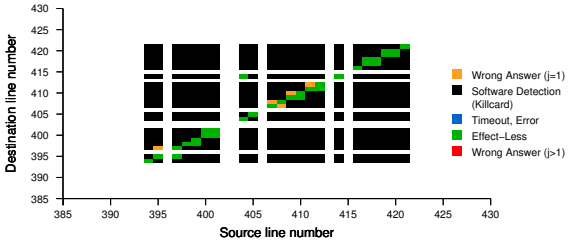

Figure 12: Weakness detection results for a secured version of the aes_addRoundKey_cpy function

408 → 407: `buf[i] ^= key[i];` // same attack but a backward jump

409 → 410: `cpk[i] = key[i];` // initially 244 → 245

411 → 412: `cpk[16+i] = key[16 + i];` // initially 245 → 246

We can observe that any statement involved in the encryption that is jumped impacts the global results. As the statement `buf[i] ^= key[i];` is not idempotent, then a backward jump also impacts the result. Securing individual lines requires an extra process such as line duplication for idempotent ones. Non-idempotent statements must be first decomposed into idempotent ones [38]. We use such a process in a case study of authentication process in Section 6.3 and show that it enables to thwart all harmful attacks.

5. Formal verification of countermeasures

In this section, we present the formal verification of our securing scheme. This verification helped us in designing effective countermeasures and gives us strong confidence in their effectiveness against attacks.

We first present the models used for program execution from a control flow perspective and for jump attacks, as well as the properties to check to ensure the CFI of a secured program execution even in the presence of attacks. The verification of the correctness of our countermeasure schemes is based on an equivalence-checking method between synthetic codes representing a specific control-flow construct and the corresponding secured versions.

5.1. Code representation and decomposition for CFI verification

From a control flow perspective, a program execution can be viewed as the execution of a sequence of statements. A high-level program can be represented as a transition system whose states are defined by the values of program variables (contents of the memory) and of the PC whose value specifies a source code line in the C program. Any transition mimics the state transformation induced by the execution of an individual statement: updating the PC and potentially some variables or the contents of memory. Figure 14 illustrates the representation of a program as a transition system.

A program can be decomposed into functions, and any function body can be decomposed into top-level code regions containing either only straight-line statements or a single control-flow construct (loops, if-then-else, and switch). Sequential execution of these regions guarantees that, if the CFI is ensured at the exit of a code region, the following code region starts with a correct input from a control flow perspective. Thus, the integrity of the control flow of both code regions with a small overlap can be proven by proving the CFI of each code region. Our countermeasure scheme relies on securing each control-flow construct (function call/sequential code, if-then-else, while constructs only and with continue or break statements, and switch) nested with a few straight-line statements of the surrounding block. Then, our verification approach verifies separately each control-flow construct enclosed with straight-line statements of the surrounding block so that all possible executions of the secured version are stopped by a countermeasure in the presence of harmful attacks or their control flow is upstanding with respect to the initial code.

As control-flow constructs can be nested, many combinations of control-flow constructs could be modeled. However, any control-flow construct can be viewed as a single statement that is correctly executed or not. Thus, in the verification models of our countermeasures, we only consider straight-line statements inside control-flow constructs. The idea is that, if properties hold for each individual construct, they hold for all of their combinations owing to the overlapping of regions considered for performing the verification.

5.2. Models for CFI verification

5.2.1. State machine model

To model and verify the integrity of control flow, we associate a dedicated statement counter denoted as `cntv_`α`i` with each statement `stmt_i` of the original code belonging to a function or to a control-flow construct called α. In the remainder of the paper, we refer to such counters as *statement counters* to distinguish them from counters used in the countermeasures, denoted as *protection counters*. We model the execution of a statement `stmt_i` by incrementing its associated statement counter `cntv_`α`i`.

Then, the execution of a sequence of statements is modeled by a transition system TS, defined by $TS = \{S, T, S_0, S_f, L\}$, where S is the set of states, T is the set of transitions $T : S \rightarrow S$, and S_0 and S_f are the subsets of S containing the initial states and final states, respectively. The final states from S_f are absorbing states. A state from S is defined by the value of the PC, the value of statement counters associated with every statement of the initial code, and the value of the variables involved in the control flow (*e.g.*, in iteration counts or in a condition). L is a set of labels corresponding to the possible values of the PC, *i.e.*, line numbers in the source code. Initial states are states with a PC value equal to the first line of the modeled code and where all statement counters hold 0 and all other variables are free; *i.e.*, there will be an initial state for all combinations of their possible values. Any transition from T is defined by the effect of the statement `stmt_i` associated with the PC value. Transitions change the PC value to the next line number to be executed and increment the statement counter `cntv_`α`i` associated with `stmt_i` of the construct α. A jump attack, as considered in this work, can only corrupt the PC. Thus, for such attacks, modeling the full memory contents and other machine registers is not relevant.

5.2.2. Models with and without countermeasures

To prove that our countermeasure scheme for a construct `c` is robust against a jump attack and that its secured version is equivalent to the initial one, we built two transition systems: one for the initial control-flow construct called `M(c)` and another one for the protected version including the countermeasure called `CM(c)`. Figure 13 shows the building principle using one statement `stmt1`.

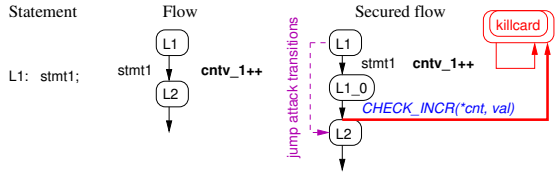

Figure 13: Principles of the transition systems `C(c)` and `CM(c)`

Note that the states of transition system `CM(c)` hold the value for the protection counters and extra variables used by the protection schemes to be able to simulate an attack detection.

We aim to prove that the `CM(c)` model under jump attacks provides the same global functional property from a control-flow point of view as the initial model when the attacks are harmless and that a harmful attack is always detected. Thus, we must model the jumps attacks into the `CM(c)` model (the `M(c)` model is not subject to attack and is used as a reference only).

Consequently, in `CM(c)`, we add all possible jump attacks to model the possibilities of the attacker. A jump attack is equivalent to the modification of the PC with an unexpected value. As by design, our countermeasures are effective against attacks that jump at least two C lines; therefore, we add faulty transitions between every pair of states of `CM(c)` separated by at least one line of the C code. These transitions are represented by the pink dashed arrow in Figure 13: they correspond to an update of the PC. As we assume that only a single fault can occur, every faulty transition is guarded by a Boolean indicating that a fault has already occurred.

Moreover, in `CM(c)`, the checks of protection counters, added by our countermeasure schemes, may result in a call to the attack detection handler `killcard()`. Hence, there is an additional PC value denoted as `killcard` in any `CM(c)`. All states with this PC value are final (and, therefore, absorbing). All transitions labeled with a countermeasure macro performing a check may change the PC to `killcard`. They are represented by a bold red arrow in Figure 13.

5.3. Equivalence checking of CFI

Our formal verification is based on an equivalence-checking method, *i.e.*, the `M(c)` model is considered as a reference to which the executions according to the `CM(c)` model are compared. To achieve this comparison, the model checker builds a product of both models. To prove the effectiveness of our countermeasures as well as their correctness (they do not change the program flow), we query the model checker to verify some properties on execution paths (*i.e.*, valid sequences of transitions in each model) expressed as computation tree logic (CTL) formulae. The model checker analyzes the states of the product of both models on all possible paths from the initial states and replies if the properties hold.

An initial state of one individual model corresponds to a state where 1) the PC is set to the first line of the corresponding code and 2) the control variables that influence the execution (*e.g.*, such as the number of iterations to be performed and the value of the condition used in an if construct) are set to a specific value. The set of initial states is then the union of all states with different values for the control variables (these variables are free). Leaving this variable free is mandatory to explore all possible execution paths. To force the model checker to analyze only the execution paths in the product model that start from the initial states in `M(c)` and `CM(c)` with the same value for all control variables, we must 1) set any control variable as input of both models 2) force the input of both models to be the same. Thus, equivalence will be verified on execution paths that have the same values for control variables in their initial states in both `M(c)` and `CM(c)`.

5.4. Control-flow integrity properties

To verify the correctness and robustness of the secured code modeled by `CM(c)`, in the absence or presence of an attack, we must prove three different properties, namely P1, P2, and P3, which are described below.

Property P1. The execution eventually terminates. This is proven by checking that any path in `M(c)` and `CM(c)` reaches a final absorbing state.

Property P2. In `CM(c)`, for execution paths that end in a correct state, all the program statements must have been executed as many times as expected. This is proven by querying the model checker to verify that the statement counter values in a reached correct final state in `CM(c)`

(with a PC value different from killcard) are equal to the statement counter values in the corresponding reached final state in M(c) (which is determined with the input values).

Property P3. At any time during the execution, the execution order of statements must be correct with respect to the initial program or an attack must eventually be detected. As the correctness of the execution order at any time during the execution is dependent on the program control-flow structure, for each control-flow construct, we designed a formula denoted as right_flow that correlates the value of the statement counters and the control variable and whose validity at any time and along any execution path ensures the right execution order.
As an example, let us consider a straight-line region composed of n statements stmt_i for all i $\in [1, n]$. For this straight-line flow, the designed right_flow formula expresses that, for all i $\in [1, n[$, the counters cntv_αi and cntv_α(i+1) for two adjacent statements stmt_i and stmt_i+1 must respect cntv_αi = cntv_α(i+1) or cntv_αi = cntv_α(i+1) + 1, respectively. In other words, the formula expresses that the statement stmt_i is always executed before the statement stmt_i+1. By transitivity, this ensures the right execution order of all the n statements of the region. The full Property P3 given to the model checker is a CTL formula expressing that the formula right_flow holds in CM(c) at any time and along any path or that the execution reaches a final detection state (a state with the killcard value for the PC).

Because Property P3 ensures the right order of execution of statements, it holds by construction for M(c) as it is not subject to jump attacks. However, verifying this property for M(c) is useful to ensure that the reference model from which CM(c) is derived and Property P3 for both models are correct. Then, it must be checked in CM(c), which is subject to at most one jump attack per execution path.

To ensure the correctness and the robustness of the countermeasure schemes, with early and deferred detection, we must verify the same three properties on CM(c) built for the countermeasures either with early detection or with deferred detection.

5.5. Verification of the countermeasure schemes

We present and explain in detail in the following section both the models and the properties to verify the product model for a function call and a straight-line flow in the called function. We provide insight on the other constructs in Section 5.5.2.

5.5.1. Verification for a function call and a straight-line flow

Figure 14 illustrates a compact representation of both transition systems for a generic example code with a call from a function f to a function g composed only of straight-line statements. In Figure 14, the pink dashed arrows represent a subset of possible attacks. Owing to the high number of such transitions, only a subset is presented to keep the figure readable. In CM(c), intra-procedural jump attacks can occur anywhere in g: in Figure 14, the pink dashed arrows in g illustrate all possible attacks coming from L8. In f, all jump attacks that do not target the final correct states are modeled: in Figure 14, the pink dashed arrows in f represent all possible attacks starting at L3; states with a PC value equal to Line 4 are not the target of such attacks. In any CM(c), the final correct states can be the source, but cannot be the target, of jump attacks; otherwise, Property P2 could obviously not be verified. The absence of an attack targeting a final correct state does not affect the verification objective because the modeled attacks in CM(c) cover all needed cases to perform the verification for the control-flow construct: the effects of any forward or backward jump over the call are covered by those that target or come from a point after the call, *e.g.*, from and to Line 3.

The three properties instantiated for the example of Figure 14 are presented in Figure 15. For clarity, in the figure and in the remainder of this paper, we use the notation Before(cntv_A, cntv_B) to express that cntv_A is equal either to cntv_B or to cntv_B+1, *i.e.*, that statement A precedes statement B.

Property P1 (Lines 2 and 3 of Figure 15) ensures the termination of the execution. Property P2 (Lines 5-7 of Figure 15) expresses the right execution count of the statements in CM(c) when the execution ends in a final correct state (where PC is equal to Line 4) by comparing the statement counter values in both the M(c) and the CM(c) model.

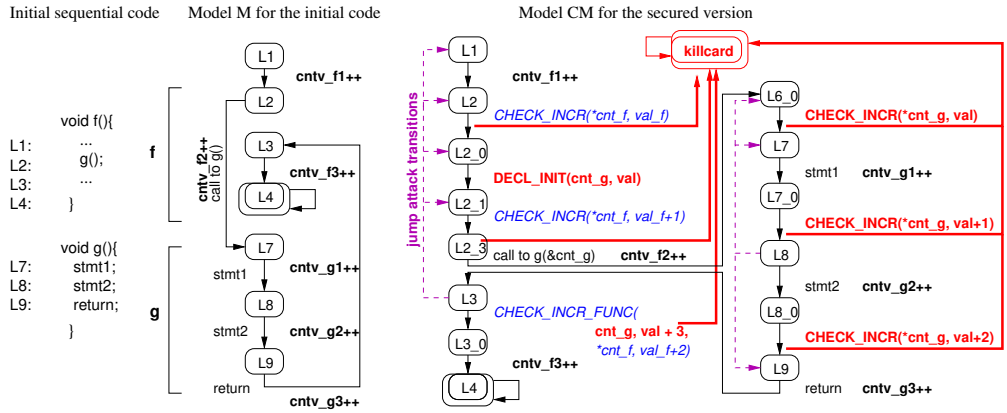

Figure 14: Compact representation of the transition system for a function call and a straight-line flow

```
; P1 : final state reachability in M and CM                                                                    1
AG(AF(M.pc = L4))                                                                                              2
AG(AF(CM.pc = L4 + CM.pc = killcard))                                                                          3
; P2 : right statement execution counts in CM and M when the execution reaches a correct final state          4
AG( ((M.pc = L4) . (CM.pc = L4)) => (M.cntv_f1 = CM.cntv_f1) . (M.cntv_f2 = CM.cntv_f2)                        5
                                 . (M.cntv_f3 = CM.cntv_f3) . (M.cntv_g1 = CM.cntv_g1)                         6
                                 . (M.cntv_g2 = CM.cntv_g2) . (M.cntv_g3 = CM.cntv_g3))                        7
; P3 on CM : right execution order of statements at any time and along any path in CM or attack detection     8
AG(right_flow_cm + AF(CM.pc = killcard))                                                                       9
right_flow_cm = Before(CM.cntv_f1, CM.cntv_f2) . Before(CM.cntv_f2, CM.cntv_f3) ; right order in f             10
              . Before(CM.cntv_g1, CM.cntv_g2) . Before(CM.cntv_g2, CM.cntv_g3) ; right order in g             11
              . Before(CM.cntv_f2, CM.cntv_g1) . Before(CM.cntv_g3, CM.cntv_f3) ; right order of the call stmt  12
; P3 on M : right execution order of statements at any time and along any path in M                           13
AG(right_flow_m)                                                                                               14
right_flow_m = Before(M.cntv_f1, M.cntv_f2) . Before(M.cntv_f2, M.cntv_f3) ; right order in f                  15
             . Before(M.cntv_g1, M.cntv_g2) . Before(M.cntv_g2, M.cntv_g3) ; right order in g                  16
             . Before(M.cntv_f2, M.cntv_g1) . Before(M.cntv_g3, M.cntv_f3) ; right order of the call stmt      17
```

Figure 15: Model checker properties for a call to a function composed of straight-line statements

For the right execution order of statement, we aim to prove not only that a straight-line flow is secured by our countermeasure (in `g`) but also that a whole function execution (`g`) is protected, as well as calls to a function (the call to `g` from `f`). Thus, the verification must ensure the right execution of the call statement (in `f`). To this end, the `right_flow_cm` formula for `CM(c)` (Lines 10-12 of Figure 15) ensures the right straight-line flow in `f` (Line 10), the right straight-line flow in `g` (Line 11), and the right flow for the call statement (Line 12). Finally, Property P3 for `CM(c)` (Line 9 in Figure 15) ensures that, during any execution, either the execution order is as expected or an attack is detected.

5.5.2. *Verification for the other constructs*

The same verification (code modeling and expression of the three properties) process has been applied for the other constructs as well.

As previously explained, Property P3 differs between constructs and should be redesigned.

Figure 16 illustrates Property P3 for the `CM(c)` associated with the if-then-else construct of Figure 6. The variables `cntv_*` are associated with the statements `*` of the example of Figure 6. The `right_flow_cm` formula specifies that Subproperties P3_1, P3_2a, P3_2b, and P3_3 must all be valid. P3_1 ensures that the straight-line flow order before the if construct, condition included, is correct; P3_2a and P3_2b express the correct straight-

```
; P3: In each state of the CM model, the right execution order or an attack will be detected
AG(right_flow_cm = 1 + AG(AF(CM.pc = killcard)));

right_flow_cm = P3_1 . P3_2a . P3_2b . P3.3;

; P3_1: straight_line flow before the branch
P3_1 = Before(CM.cntv_1, CM.cntv_2) . Before(CM.cntv_2, CM.cntv_cond)

; P3_2a: straight_line flow for the then branch
P3_2a = Before(CM.cntv_cond, CM.cntv_then1) . Before(CM.cntv_then1, CM.cntv_then2)

; P3_2b: straight-line flow for the else branch
P3_2b = Before(CM.cntv_cond, CM.cntv_else1)

; P3_3: straight_line flow of the statement after the branch as well as exclusive and valid execution (with
   respect to the condition value) of the last statement of the branch
P3_3 = ((CM.cntv_3 = 0) + ((cond = 0) . (M.cntv_else1 = 1))^((cond = 1) . (M.cntv_then2 = 1)))
```

Figure 16: Property P3 expressing the right execution order property for the if-then-else construct given in Figure 6

line flow order inside each branch, condition included; and, finally, P3_3 checks that the statement just after the if-then-else construct is executed after the last statement of the right single branch. Note that the conjunction of Subproperties P3_2a, P3_2b, and P3_3 ensures that the branches of the `CM(c)` model are exclusive.

For switch constructs, the straight-line flow order of each case must be correct and exclusively executed. The variable `n_sw` added by the protection scheme is used to express such an exclusive execution (as performed with the variable `cond` for the if-then-else case). Cases without a break statement are also handled.

For a simple loop (with a straight-line flow body without any break nor any continue statement as in the example in Figure 8), the `right_flow` formula expresses that the statements inside the loop respect the straight-line flow order (the condition must be executed before the first statement of the body, and the statements of the body must be executed in the right order as in a straight-line flow). Owing to the looping back, the formula also expresses that the condition is executed either as many times as the last statement or one more time. Additionally, the `right_flow` formula also expresses that the execution order before the loop (condition of the loop included) is correct and that when the statement after the loop has been executed, the condition of the loop must have been executed exactly one more time than the last statement of the loop.

The presence of break and continue statements in a loop body makes Property P3 more intricate as some iterations can be interrupted. However, it remains possible to express the right execution order whatever the execution path (the break and continue statements are controlled by a condition that is used in Property P3 to handle them).

5.5.3. Verification results

We chose the Vis model checker [53] to prove the effectiveness of our countermeasure schemes. This tool can take a transition system described using a subset of the Verilog Hardware Description Language as input. Using Verilog is convenient to model transition systems as previously defined. The Vis model checker supports symbolic model checking techniques, which enable to perform the proof in a symbolic way.

We modeled all synthetic codes representing the different constructs presented in Section 4 without countermeasures and with both versions of the countermeasures (CMED and CMDD). We also expressed all the three properties required to perform the verification as CTL properties for all the resulting models. The three properties hold for all the constructs and for both countermeasure schemes. The verification required less than a few minutes for each construct verification. Figure 17 shows an example of the output of the verification of the loop construct protection with early detection.

```
vis> read_verilog main_while.v
vis> init_verify
vis> model_check -i prop_eq_check.ctl
# MC: formula passed --- AG(AF(while_c.pc=L9))
# MC: formula passed --- AG(AF((while_cm.pc=L9 + while_cm.pc=killcard)))
# MC: formula passed --- AG((right_flow=1 + AG(AF(while_cm.pc=killcard))))
# MC: formula passed --- AG(right_flow_g=1)
# MC: formula passed --- AG(((while_c.pc=L9 * while_cm.pc=L9) -> stmt_exec_count_eq=1))
```

Figure 17: Output of the verification of the loop construct protection with early detection

6. Experimental results

We implemented all the software components presented in Figure 1. The current implementation processes the C code and outputs secured corresponding C versions. The following constructs are currently supported: function calls as statement or expressions, even imbricated ones; if-then-else constructs; and loop constructs (for and while). Switch cases and loops with break or continue statements are not fully implemented yet. The results presented in this section show the capability of our tool to handle C codes automatically while being a research prototype available online[1].

For the experiments, we considered two sets of programs: the first set consisted of three well-known encryption algorithms available in C (AES [52], SHA [54], and Blowfish [54]) and the second one was the FISSC benchmark, which is a recent set of reference programs for smart cards [6]. In FISSC, we selected the VerifyPIN benchmark, which simulates the checking of an incorrect PIN code. VerifyPIN exists in different versions, which correspond to different levels of software protection. The protection mechanisms have been manually added at the C code level by the authors of FISSC [6]. We considered the two versions of VerifyPIN: `verifyPIN_1` and `verifyPIN_7` (protection levels 1 and 7, respectively). We also considered hand-written C construct templates to validate the effectiveness of our protection schemes per construct via simulation.

6.1. Countermeasure effectiveness

First, we simulated all the intra-procedural transient jump attacks from line i to line j at the C code level for each function (such as in Figure 2). In all our experiments, the distinguisher classified as *Wrong answer* any attack that let the program finish with a corrupted output even if a corrupted output did not necessarily imply the existence of an exploit for the cryptographic algorithm we considered. This classification aims at judging the capability of our securing scheme to protect a control flow.

Table 1 shows the results of jump attack simulations at the source code level. The second column of Table 1 shows that, as formally verified, all attacks with a jump distance greater than or equal to two C statements were captured by our countermeasures designed for this fault model. For example, $32\,811$ jump attacks were harmful for SHA, whereas there was none for both its secured versions (SHA + CMED and SHA + CMDD). Similar results were obtained for the VerifyPIN benchmark and the C construct templates.

Moreover, as there is no one-to-one correspondence between the high-level fault models and the low-level ones, we simulated all possible intra-procedural jump attacks at the assembly code level, considering as a target an ARM Cortex-M3 processor. We used the Keil ARM-MDK debugger and Keil simulator [55] for the replacement of every instruction during execution by a jump anywhere into the same function. We considered only two functions of the AES benchmarks (`aes_encrypt` and `mix_column`) owing to a very long simulation time (up

[1]The source code of our framework is published online under the GPL license and is available at `http://cfi-c.gforge.inria.fr/`

Table 1: Source code jump attack effect classification for original version, early detection (+CMED) and deferred detection (+CMDD) (WA: Wrong Answer; EL: Effect Less; SD: Software Detection; TO: Error or Timeout)

	WA		WA		EL		SD: KILLCARD		TO		TOTAL
C JUMP ATTACKS	size > 1		size = 1								
C construct templates	**Attacking all functions at C level for all transient rounds**										
if-then-else	9	37 %	2	8.3%	13	54 %	0		0		24
if-then-else + CMDD/CMED	0		4	0.9%	85	19 %	334	75%	17	3.9%	440
while	50	41 %	27	22 %	43	35 %	0		0		120
while + CMDD/CMED	0		23	1.5%	206	13 %	1245	83%	26	1.7%	1500
switch case	46	32 %	8	5.6%	89	62 %	0		0		143
switch case + CMDD/CMED	0		9	0.8%	494	42 %	652	56%	0		1155
continue (loop)	227	55 %	47	11 %	117	28 %	0		16	3.9%	407
continue (loop) + CMED	0		41	0.7%	604	10 %	4971	88%	1	0.0%	5617
continue (loop) + CMDD	0		41	0.7%	1096	17 %	4971	81%	1	0.0%	6109
break (loop)	141	65 %	31	14 %	44	20 %	0		0		216
break (loop) + CMED	0		30	0.7%	608	14 %	3462	84%	0		4100
break (loop) + CMDD	0		30	0.7%	731	17 %	3462	81%	0		4223
Fibonacci (recursivity test)	212	48 %	58	13 %	160	36 %	0		10	2.3%	440
Fibonacci + CMED	0		59	0.9%	282	4.1%	6478	94%	37	0.5%	6856
Fibonacci + CMDD	0		59	0.8%	637	8.4%	6623	87%	238	3.1%	7557
Ciphering	**Attacking all functions at C level for all transient rounds**										
AES	7835	29 %	1104	4.2%	17,323	65 %	0		108	0.4%	26,370
AES + CMED	0		603	0.2%	18,339	6.7%	255,614	93%	1	0.0%	274,557
AES + CMDD	0		603	0.2%	37,334	13 %	236,619	86%	1	0.0%	274,557
SHA	32,811	75 %	1527	3.5%	8531	19 %	0		405	0.9%	43,274
SHA + CMED	0		1143	0.3%	5015	1.3%	368,941	98%	290	0.1%	375,389
SHA + CMDD	0		1144	0.3%	5015	1.3%	368,736	98%	494	0.1%	375,389
Blowfish	70,318	32 %	3553	1.7%	134,360	62 %	0		5490	2.6%	213,721
Blowfish + CMED	0		2468	0.2%	312,745	25 %	887,364	73%	6852	0.6%	1,209,429
Blowfish + CMDD	0		2467	0.2%	312,745	25 %	869,690	71%	24,527	2.0%	1,209,429
FISSC	**Attacking all functions at C level for all transient rounds**										
VerifyPIN_1	49	7.4%	2	0.3%	600	90 %	0		13	2.0%	664
VerifyPIN_1 + CMED	0		2	0.0%	1229	11 %	9525	88%	16	0.1%	10,772
VerifyPIN_1 + CMDD	0		2	0.0%	1285	11 %	9392	87%	93	0.9%	10,772
VerifyPIN_7	187	2.4%	3	0.0%	4483	57 %	3094	39%	17	0.2%	7784
VerifyPIN_7 + CMED	0		2	0.0%	39,624	32 %	82,598	67%	0		122,224
VerifyPIN_7 + CMDD	0		2	0.0%	39,624	32 %	82,598	67%	0		122,224

to 3 weeks per simulation), highlighting the benefits of performing the attack simulation at the source code level. The considered functions have a different control flow: `aes_encrypt` consists of a loop with an if construct and several function calls, whereas `mix_column` only contains a straight-line flow inside a loop. The results are presented in Table 2. For the `aes_encrypt` function, the total number of harmful attacks in the secured version with early detection (respectively with deferred detection) represented only 30.8% (482/1566) (respectively 22.7%) of those in the original code: our countermeasures were able to defeat 68.4% (respectively 76.3%) of the attacks on this function. For the `mix_column` function, the countermeasure with deferred detection (respectively with early detection) defeated 69.2% (respectively 57.6%) of the harmful attacks on the original code. The difference between the number of remaining harmful attacks with early and deferred detection was owing to the additional instructions added by the checks necessary for the early detection. Some of these instructions add a few starting points for harmful jump attacks. The attack surface was significantly reduced as only less than 1.99% of the attacks provided an advantage to the attacker.

Finally, we simulated attacks that call an unexpected function instead of the expected one for all the benchmarks using the Keil simulator. Such attacks are a specific case of inter-procedural attacks. However, a harmful inter-procedural jump attack (i.e., leading to a jump from the middle of a function to the middle of another one) requires the induced jump to come from a point where the stack, the protection counters, and extra variables have a consistent and expected state at the destination of the jump. Hence, the probability of any inter-procedural jump to succeed is very low. Hence, we focused on attacks

Table 2: Classification of the effects of an assembly code jump attack for the original version, early detection (+CMED), and deferred detection (+CMDD) (WA: Wrong Answer; EL: Effect Less; SD: Software Detection; TO: Error or Timeout)

	WA		WA		EL		SD: KILLCARD		TO		TOTAL
	size > 1		size = 1								
AES	Jump attacks at the assembly code level for the first transient round										
aes_encrypt	1566	82.8 %	36	1.9 %	179	9.5 %			111	5.9 %	1892
aes_encrypt + CMED	482	0.3 %	24	0.01 %	19,895	11.1 %	155,639	87.3 %	2466	1.3 %	178,506
aes_encrypt + CMDD	355	0.4 %	24	0.02 %	6088	6.5 %	82,902	88.8 %	3961	4.2 %	93,330
aes_mixcolumn	2042	86.8 %	45	1.9 %	200	8.5 %			65	2.8 %	2352
aes_mixcolumn + CMED	834	1.3 %	51	0.08 %	7592	11.6 %	56,547	86.6 %	256	0.4 %	65,280
aes_mixcolumn + CMDD	615	1.9 %	29	0.09 %	7329	23 %	22,012	69.1 %	1877	5.9 %	31,862

corresponding to a wrong function call. The results, presented in Table 3, showed that all harmful attacks were captured and many harmless attacks were also detected. Thus, our countermeasures are also effective against unexpected function calls.

6.2. Overhead

In this section, we discuss the overhead induced by our countermeasures. We compare both countermeasure schemes. We chose to apply a full code securing, which often implies high overheads [38, 49] and gives an idea of the upper bound of the overheads.

Table 4 shows the code sizes as well as the execution times of the original version and the secured ones (+CMED or +CMDD) for an x86 target machine and a Cortex-M3 processor. For the x86 platform, the execution time overhead ranged from $+41\%$ (Blowfish) up to $+81\%$ (AES) for the deferred detection. Of course, early detection that performed a check of the counter value between each C statement increased the overheads, which ranged from $+56\%$ (Blowfish) up to $+138\%$ (AES). For the embedded ARM processor, the overhead was higher. This was primarily due to the simpler processor design, which does not exploit instruction-level parallelism and does not have branch prediction. The highest overhead was incurred for AES ($+219\%$). The results also showed that deferred detection reduced the overhead for both code size and performance. However, early detection may be more appropriate if a fault could provide benefits to the attacker in the presence of deferred detection, as it can be the case of combined side channel and fault attacks [56]

The performance of the secured benchmarks depended on the presence of IO operations and their speed. In the considered use cases, AES did not use input files, whereas SHA and Blowfish did. For example, the multiple file reads in Blowfish limited the measured performance overheads on the x86 platform. In the ARM-v7 simulation platform, the performance was different because we had to remove some IO operations (those manipulating a file as a read from a file).

The overhead of our countermeasure scheme is not negligible; however, critical routines are only a subset of the whole software embedded in a smart card. Thus, based on the weakness detection, a security expert could decide which functions must be secured to ensure security while reducing the impact on performances without sacrificing security.

6.3. VerifyPIN use case

To illustrate the proposed methodology for weakness detection and code hardening as well as the effectiveness of the proposed protection scheme even if deployed at the C code level, we considered the VerifyPIN benchmark [6]. This small benchmark is mainly composed of two functions: `verify_pin` and `byte_compare`. The function `verify_pin` is in charge of the authentication and calls the `byte_compare` function, which performs a comparison between a given pin code and a hard-coded one. A harmful attack corresponds to an authentication success whereas the entered pin code is wrong.

We first simulated all possible jump attacks at the assembly code level for the function `verify_pin`. The results in Table 5 show that, without any protection, in version `1 (FISSC)` there were 189 harmful jump attacks (187 with a distance greater than one and two instruction skips). When secured with our protection scheme, the total number of harmful attacks decreased to 74, which still represents 39.1% of the initial set of harmful attacks. This result shows that our protection schemes, while adding code, are effective in reducing the attack

Table 3: Classification of the effects of an assembly bad call attack for the original version and early detection
(WA: Wrong Answer; EL: Effect Less; SD: Software Detection; TO: Error or Timeout)

	WA		EL		SD: KILLCARD		TO		TOTAL
ASSEMBLY CALL ATTACKS	Attacking all function calls at the assembly code level for the first transient round								
AES	249	59.3%	139	33.1%			32	5 %	420
AES + CMED	0		21	5 %	398	94.8%	1	0.2%	420
SHA	35	48.7%	13	18 %			24	33.3%	72
SHA + CMED	0		8	11.1%	61	84.7%	3	4.2%	72
Blowfish	9	21.4%	18	42.9%			15	35.7%	42
Blowfish + CMED	0		18	42.9%	17	40.5%	7	16.6%	42

Table 4: Size and overhead for the original version, early detection (+CMED), and deferred detection (+CMDD)
(WA: Wrong Answer; EL: Effect Less; SD: Software Detection; TO: Error or Timeout)

	x86					ARM-V7M			
	Simulation time	Size		Execution time		Size		Execution time	
		Bytes	Overhead	Time	Overhead	Bytes	Overhead	Time	Overhead
AES	22 min	18,016		0.77 ms		4224		62.18 ms	
AES + CMED	8 h 40 min	38,552	+113%	1.83 ms	+138%	13,472	+219%	186.7 ms	+200%
AES + CMDD	6 h 05 min	30,360	+68%	1.39 ms	+81%	9732	+110%	131 ms	+110%
SHA	58 min	13,528		1.04 µs		3184		504.3 µs	
SHA + CMED	12 h 34 min	21,776	+60%	2.10 µs	+102%	7032	+121%	851.36 µs	+69%
SHA + CMDD	9 h 37 min	17,680	+30%	1.79 µs	+72%	5272	+66%	730.4 µs	+45%
Blowfish	4 h 39 min	30,352		0.67 µs		5546		4.55 ms	
Blowfish + CMED	2 days 0 h 44 min	46,784	+54%	1.05 µs	+56%	14,668	+164%	14.46 ms	+217%
Blowfish + CMDD	1 day 14 h 30 min	38,592	+32%	0.89 µs	+41%	10,820	+95%	13.01 ms	+186%

surface. The manually protected C version of the function (7 (FISSC)) includes redundancy checks and uses step counters all the way to detect control-flow disruption. However, simulating jump attacks at the assembly code level on this function revealed 279 harmful attacks. This demonstrates the difficult task of protecting source code as well as a side effect of adding protection mechanisms: as the code grows, it may contain more weaknesses than that without any protection.

Many implementation flaws were detected in the source code of the function `verify_pin` in both versions 1 (FISSC) and 7 (FISSC): the Boolean true value, even if associated with a specific value, is assigned in different places to the variable holding the authentication result. This makes attacks easier to be performed as any jump to one of these assignments results in an authentication success. We changed the source code to avoid these programming flaws in a new version denoted as 2 (new) in Table 5. As a result, no harmful jump attacks exist in the function of this version (2 (new)). Therefore, writing the C code in a secure manner is crucial.

We also analyzed the `byte_compare` function shown in the left-hand side of Figure 18. Without any protection, there were 16 harmful attack points. We changed

```
BOOL byteArrayCompare(
     UBYTE* a1, UBYTE* a2
     , UBYTE size) {

  int i;
  BOOL ret = BOOL_TRUE;
  i = 0;
  while(i < size)
  {
    if(a1[i] != a2[i])
    {
      ret = BOOL_FALSE;
    }
    i++;
  }

  return ret;
}
```

```
BOOL
byteArrayCompare_pinsize(
     UBYTE* a1, UBYTE* a2
     ) {
  int i, tmp, n;
  BOOL ret = BOOL_FALSE;
  n = 0; i = 0;
  while(i < PIN_SIZE) {
    if(a1[i] == a2[i]) {
      n++;
    } i++;
  }
  if (PIN_SIZE == n) {
    if (PIN_SIZE == n) {
      ret = BOOL_TRUE;
  }}
  return ret;
}
```

Figure 18: The `byte_compare` function: the original (1) and the new version (2_dup)

the function to avoid the default assignment of the returned value to a true value at the beginning of the function, which offers by default many ways to force the function to return a Boolean true value. Then, we used our methodology and simulated all jump attacks at the C code level for the new version. We analyzed the weaknesses

Table 5: Classification of the effects of an assembly code jump attack for the original version, early detection (+CMED), and deferred detection (+CMDD) (WA: Wrong Answer; EL: Effect Less; SD: Software Detection; TO: Error or Timeout)

VERIFYPIN		WA size > 1		WA size = 1		EL		SD: KILLCARD		TO		TOTAL
Version	Function	Jump attacks at the assembly code level for the first transient round										
1 (FISSC)	verify_pin	187	15.7 %	2	0.17 %	939	79 %			61	5.1 %	1189
1 (FISSC)	verify_pin + CMDD	72	0.18%	2	0.005%	8801	22.7 %	29,410	75.8 %	526	1.35%	38,809
7 (FISSC)	verify_pin	277	3.04%	2	0.02 %	4482	49.14%	4275	46.88%	84	0.92%	9120
2 (new)	verify_pin	0	0 %	0	0 %	1089	97.5 %			33	2.95%	1122
1 (FISSC)	byte_compare	13	6.22%	3	1.43 %	165	98.95%			28	13.39%	209
2_dup (new)	byte_compare + CMDD	0	0 %	0	0 %	18,383	55.8 %	14,020	42.56%	539	1.64%	32,942
2_dup (new)	byte_compare + partial	0	0 %	0	0 %	4054	50.61%	3759	46.93%	197	2.46%	8010

using our visualization tool: one statement was at the target of all harmful attacks, i.e., the one that set the return value to true. The condition of the if statement protecting this assignment may be successfully corrupted as our protection scheme cannot protect against intra C-line jump attacks. We then duplicated the if-statement line (which was idempotent) to evaluate the condition twice on different lines. The corresponding source code, called `2_dup (new)`, is shown in the right-hand side of Figure 18. The results of the simulations of jump attacks at the assembly code level considering this new version (`2_dup (new)`) protected with our deferred detection protection scheme showed that the new version is fully protected, as reported in Table 5. This demonstrates the importance of our source code weakness visualization tool in detecting the sensitive parts and the effectiveness of our protection scheme in detecting any jump attacks at the assembly code level. Note that even single-instruction skips are harmless.

Moreover, as only a small part of the code of the entire new version of VerifyPIN is sensitive, we only needed to protect the code partially. Thus, we query our tool to harden only a portion of the code (from line 10 to 14 in the right-hand side of Figure 18) as well as the call to the `byte_compare` function from `verify_pin`. All functions are minimally secured, as explained at the end of the previous section (Section 6.2). The resulting code, called version `2_dup (new) byte_compare + partial` in Table 5, is still fully protected. Code size and performance degradation are also far less impacted. The execution of the initial version requires 3.43 μs, whereas the new version runs in 3.54 μs; both have the same code sizes. The slowdown of the fully secured version is really high ($+317\%$), whereas the partially secured one runs only 60% slower. The size increase for the fully secured version is 118%, whereas it is only 40.5% for the partially secured one. Although this benchmark represents a small but a critical part of a smart cards functionalities, these results show that our approach can efficiently secure a smart card C code against control-flow disruption at the source code level. Moreover, its overhead can be mitigated by securing only the sensitive regions.

In practice, as presented in this section, developers and security experts choose the code regions inside a function to be secured. Our approach automates the process from weakness detection to code security. It is possible to extend it, and thus, extend our tools to adapt the countermeasure injection for targeting only specific regions inside functions to reduce the overhead. Automatic finer selection of code regions to be secured will be studied in a future work.

7. Conclusion

This paper presented a methodology to automatically secure, at the source code level, any C applications without a goto statement, supported by formally verified countermeasures. Three main contributions can be emphasized. First, we set up a methodology to analyze and visualize the vulnerabilities of a C code that is subject to physical faults disturbing the control flow. Second, we proposed countermeasures to detect control-flow attacks that jump more than two C lines of the code. Third, we formalized and verified the countermeasure schemes. This approach is original and can be adapted for new countermeasures that security researchers or experts would like to verify. Finally, we automated the injection of the countermeasures and the simulated attacks to secure any regular C code or to experiment with the effect of a physical

fault. The developed tools are open source and can be freely downloaded from the Web.

Experimental results showed that the proposed countermeasures defeated 100% of C jump attacks with a distance of two statements or beyond. Moreover, our countermeasures were able to capture all unexpected function calls. They were also able to significantly reduce the number of attacks injected at the assembly code level and can even be used in combination with benchmark rewriting to defeat all attacks, given a fine classification of harmful attacks. The overhead is high; however, in a smart card, the code that needs to be fully protected represents only a small part of all the embedded software.

Our future work will address the optimization of countermeasure injection according to the weakness detection step. If harmless attacks are found inside a function, injection of countermeasures might be adapted accordingly to reduce their cost while preserving their effectiveness. Working on the compatibility of countermeasures against data fault attacks with the presented work is also a promising research direction.

References

[1] S. Bhasin, P. Maistri, F. Regazzoni, Malicious wave: A survey on actively tampering using electromagnetic glitch, in: International Symposium on Electromagnetic Compatibility, 2014, pp. 318–321.

[2] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, C. Whelan, The sorcerer's apprentice guide to fault attacks, Proceedings of the IEEE 94 (2) (2006) 370–382. doi:10.1109/JPROC.2005.862424.

[3] A. Dehbaoui, A.-P. Mirbaha, N. Moro, J.-M. Dutertre, A. Tria, Electromagnetic Glitch on the AES Round Counter, in: 4th International conference on Constructive Side-Channel Analysis and Secure Design, Vol. 7864, Springer Berlin / Heidelberg, Paris, France, 2013, pp. 17–31. doi:10.1007/978-3-642-40026-1_2.

[4] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, F. Regazzoni, Countermeasures against fault attacks on software implemented AES: effectiveness and cost, in: 5th Workshop on Embedded Systems Security, ACM, New York, NY, USA, 2010, pp. 1–10. doi:10.1145/1873548.1873555.

[5] S. Yen, S. Kim, S. Lim, S. Moon, A countermeasure against one physical cryptanalysis may benefit another attack, in: K. Kim (Ed.), 4th International Conference on Information Security and Cryptology, Vol. 2288 of LNCS, Springer, Seoul, Korea, 2001, pp. 414–427. doi:10.1007/3-540-45861-1_31.

[6] L. Dureuil, G. Petiot, M.-L. Potet, A. Crohen, P. D. Choudens, FISSC: a Fault Injection and Simulation Secure Collection, in: International Conference on Computer Safety, reliability and Security, Vol. 9922 of LNCS, Springer Berlin / Heidelberg, Trondheim, Norway, 2016, pp. 3–11. doi:10.1007/978-3-319-45477-1_1.

[7] M.-L. Potet, L. Mounier, M. Puys, L. Dureuil, Lazart: A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections, in: 7th International Conference on Software Testing, Verification and Validation, IEEE, Cleveland, OH, USA, 2014, pp. 213–222. doi:10.1109/ICST.2014.34.

[8] T. Barry, D. Couroussé, B. Robisson, Compilation of a Countermeasure Against Instruction-Skip Fault Attacks, in: 3rd Workshop on Cryptography and Security in Computing Systems, ACM Press, Prague, Czech Republic, 2016, pp. 1–6. doi:10.1145/2858930.2858931.

[9] R. D. Keulenaer, J. Maebe, K. D. Bosschere, B. D. Sutter, Link-time smart card code hardening, Int. J. Inf. Sec. 15 (2) (2016) 111–130. doi:10.1007/s10207-015-0282-0.

[10] S. S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[11] A. W. Appel, M. Ginsburg, Modern Compiler Implementation in C, Cambridge University Press, New York, NY, USA, 2004.

[12] P. Berthomé, K. Heydemann, X. Kauffmann-Tourkestansky, J.-F. Lalande, High level model of control flow attacks for smart card functional security, in: 7th International Conference on Availability, Reliability and Security, IEEE Computer Society, Prague, Czech Republic, 2012, pp. 224–229. doi:10.1109/ARES.2012.79.

[13] L. Riviere, M.-L. Potet, T.-H. Le, J. Bringer, H. Chabanne, M. Puys, Combining High-Level and Low-Level Approaches to Evaluate Software Implementations Robustness Against Multiple Fault Injection Attacks, in: 7th International Symposium on Foundations & Practice of

Security, Vol. 8930, IEEE, Montreal, Canada, 2015, pp. 92–111. doi:10.1007/978-3-319-17040-4_7.

[14] J. Proy, K. Heydemann, A. Berzati, A. Cohen, Compiler-assisted loop hardening against fault attacks, ACM Trans. Archit. Code Optim. 14 (4) (2017) 36:1–36:25. doi:10.1145/3141234.

[15] D. Boneh, R. A. DeMillo, R. J. Lipton, On the importance of eliminating errors in cryptographic computations, Journal of Cryptology 14 (2) (2001) 101–119. doi:10.1007/s001450010016.

[16] N. Timmers, A. Spruyt, M. Witteman, Controlling PC on ARM Using Fault Injection, in: Fault Diagnosis and Tolerance in Cryptography Workshop, IEEE, Santa Barbara, CA, USA, 2016, pp. 25–35. doi:10.1109/FDTC.2016.18.

[17] N. Moro, K. Heydemann, A. Dehbaoui, B. Robisson, E. Encrenaz, Experimental evaluation of two software countermeasures against fault attacks, in: International Symposium on Hardware-Oriented Security and Trust (HOST), IEEE, Arlington, VA, USA, 2014, pp. 112–117. doi:10.1109/HST.2014.6855580.

[18] J.-F. Lalande, K. Heydemann, P. Berthomé, Software Countermeasures for Control Flow Integrity of Smart Card C Codes, in: M. Kutylowski, J. Vaidya (Eds.), 19th European Symposium on Research in Computer Security, Vol. 8713 of LNCS, Springer International Publishing, Wroclaw, Pologne, 2014, pp. 200–218. doi:10.1007/978-3-319-11212-1_12.

[19] M. Abadi, M. Budiu, Ú. Erlingsson, J. Ligatti, Control-flow integrity principles, implementations, and applications, ACM Transactions on Information and System Security 13 (1) (2009) 1–40. doi:10.1145/1609956.1609960.

[20] R. de Clercq, R. D. Keulenaer, B. Coppens, B. Yang, P. Maene, K. D. Bosschere, B. Preneel, B. D. Sutter, I. Verbauwhede, SOFIA: software and control flow integrity architecture, in: L. Fanucci, J. Teich (Eds.), Design, Automation & Test in Europe Conference & Exhibition, IEEE, Dresden, Germany, 2016, pp. 1172–1177.

[21] M. Werner, E. Wenger, S. Mangard, Protecting the control flow of embedded processors against fault attacks, in: N. Homma, M. Medwed (Eds.), 14th International Conference Smart Card Research and Advanced Applications, CARDIS, Vol. 9514 of LNCS, Springer, Bochum, Germany, 2015, pp. 161–176. doi:10.1007/978-3-319-31271-2_10.

[22] M. Payer, A. Barresi, T. R. Gross, Fine-grained control-flow integrity through binary hardening, in: 12th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, Milan, Italy, 2015, pp. 144–164. doi:10.1007/978-3-319-20550-2_8.

[23] W. Arthur, B. Mehne, R. Das, T. Austin, Getting in control of your control flow with control-data isolation, in: 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, IEEE Computer Society, San Francisco, CA, USA, 2015, pp. 79–90. doi:10.1109/CGO.2015.7054189.

[24] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, E. Encrenaz, Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller, in: Workshop on Fault Diagnosis and Tolerance in Cryptography, Ieee, Santa Barbara, CA, USA, 2013, pp. 77–88. doi:10.1109/FDTC.2013.9.

[25] K. Pattabiraman, N. M. Nakka, Z. T. Kalbarczyk, R. K. Iyer, SymPLFIED: Symbolic Program-Level Fault Injection and Error Detection Framework, IEEE Transactions on Computers 62 (11) (2013) 2292–2307. doi:10.1109/TC.2012.219.

[26] I. Verbauwhede, The fault attack jungle - A classification model to guide you, in: Fault Diagnosis and Tolerance in Cryptography, IEEE Computer Society, Tokyo, Japan, 2011, pp. 3–8. doi:10.1109/FDTC.2011.13.

[27] M. Puys, L. Riviere, J. Bringer, T.-H. Le, High-Level Simulation for Multiple Fault Injection Evaluation, in: 3rd International Workshop on Quantitative Aspects in Security Assurance, Vol. 8872 of LNCS, Springer Berlin / Heidelberg, Wroclaw, Poland, 2015, pp. 293–308.

[28] L. Dureuil, M. Potet, P. de Choudens, C. Dumas, J. Clédière, From code review to fault injection attacks: Filling the gap using fault model inference, in: N. Homma, M. Medwed (Eds.), 14th International Conference Smart Card Research and Advanced Applications, CARDIS, Vol. 9514 of LNCS, Springer, Bochum, Germany, 2015, pp. 107–124. doi:10.1007/978-3-319-31271-2_7.

[29] L. Riviere, Securing software implementations against fault injection attacks on embedded systems, Ph.D. thesis, Ecole doctorale EDITE de Paris, Paris, France (9 2015).

[30] J. Balasch, B. Gierlichs, I. Verbauwhede, An in-depth and black-box characterization of the effects of clock glitches on 8-bit MCUs, in: L. Breveglieri, S. Guilley, I. Koren, D. Naccache, J. Takahashi (Eds.), 8th Workshop on Fault Diagnosis and Tolerance in Cryptography, IEEE Computer Society, Nara, Japan, 2011, pp. 105–114. doi:10.1109/FDTC.2011.9.

[31] G. Bouffard, J. Iguchi-Cartigny, J.-L. Lanet, Combined software and hardware attacks on the java card control flow, in: 10th Smart Card Research and Advanced Application IFIP Conference, Springer Berlin / Heidelberg, Leuven, Belgium, 2011, pp. 283–296. doi:10.1007/978-3-642-27257-8_18.

[32] D. Ceara, Detecting Software Vulnerabilities - Static Taint Analysis, Bsc thesis, Universitatea Politehnica Bucuresti, Verimag (2009).

[33] F. Yamaguchi, C. Wressnegger, H. Gascon, K. Rieck, Chucky: exposing missing checks in source code for vulnerability discovery, in: A.-R. Sadeghi, V. D. Gligor, M. Yung (Eds.), Conference on Computer and Communications Security, ACM Press, Berlin, Germany, 2013, pp. 499–510. doi:10.1145/2508859.2516665.

[34] P. Rauzy, S. Guilley, Countermeasures against high-order fault-injection attacks on CRT-RSA, in: Workshop on Fault Diagnosis and Tolerance in Cryptography, IEEE Computer Society, Busan, South Korea, 2014, pp. 68–82. doi:10.1109/FDTC.2014.17.

[35] L. Riviere, J. Bringer, T.-H. Le, H. Chabanne, A novel simulation approach for fault injection resistance evaluation on smart cards, in: 6th international Workshop on Security Testing, no. Sectest, IEEE Computer Society, Graz, Austria, 2015, pp. 1–8. doi:10.1109/ICSTW.2015.7107460.

[36] G. A. Kanawati, N. A. Kanawati, J. A. Abraham, FERRARI: a flexible software-based fault and error injection system, IEEE Transactions on Computers 44 (2) (1995) 248–260.

[37] A. Barenghi, L. Breveglieri, I. Koren, D. Naccache, Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures, Proceedings of the IEEE 100 (11) (2012) 3056–3076. doi:10.1109/JPROC.2012.2188769.

[38] N. Moro, K. Heydemann, E. Encrenaz, B. Robisson, Formal verification of a software countermeasure against instruction skip attacks, Journal of Cryptographic Engineering 4 (3) (2014) 1–12. doi:10.1007/s13389-014-0077-7.

[39] H. Shacham, The geometry of innocent flesh on the bone, in: 14th Conference on Computer and communications security, ACM Press, Alexandria, USA, 2007, pp. 552–561. doi:10.1145/1315245.1315313.

[40] T. Bletsch, X. Jiang, V. Freeh, Mitigating code-reuse attacks with control-flow locking, in: 27th Annual Computer Security Applications Conference, ACM Press, Orlando, Florida, USA, 2011, pp. 353–362. doi:10.1145/2076732.2076783.

[41] L. Szekeres, M. Payer, T. Wei, D. Song, Sok: Eternal war in memory, in: Symposium on Security and Privacy, SP, IEEE Computer Society, Berkeley, CA, US, 2013, pp. 48–62. doi:10.1109/SP.2013.13.

[42] A. M. Fiskiran, R. B. Lee, Runtime execution monitoring (REM) to detect and prevent malicious code execution, in: International Conference on Computer Design: VLSI in Computers and Processors, IEEE Computer Society, San Jose, California, 2004, pp. 452–457. doi:10.1109/ICCD.2004.1347961.

[43] H. Chen, B. Zang, CFIMon: Detecting violation of control flow integrity using performance counters, in: IEEE/IFIP International Conference on Dependable Systems and Networks, IEEE Computer Society, Boston, USA, 2012, pp. 1–12. doi:10.1109/DSN.2012.6263958.

[44] J. Danger, S. Guilley, T. Porteboeuf, F. Praden, M. Timbert, HCODE: hardware-enhanced real-time CFI, in: M. D. Preda, J. T. McDonald (Eds.), 4th Program Protection and Reverse Engineering Workshop, ACM, New Orleans, USA, 2014, pp. 6:1–6:11. doi:10.1145/2689702.2689708.

[45] N. Oh, P. P. Shirvani, E. J. McCluskey, Control-flow checking by software signatures, IEEE Transactions on Reliability 51 (1) (2002) 111–122. doi:10.1109/24.994926.

[46] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, M. H. Jakubowski, Oblivious hashing: A stealthy software integrity verification primitive, in: 5th International Workshop on Information Hiding, Springer-Verlag Berlin Heidelberg, Noordwijkerhout, The Netherlands, 2003, pp. 400–414. doi:10.1007/3-540-36415-3_26.

[47] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, M. Violante, Improved software-based processor control-flow errors detection technique, in: Annual Reliability and Maintainability Symposium, IEEE Computer Society, Alexandria, VA, USA, 2005, pp. 583–589. doi:10.1109/RAMS.2005.1408426.

[48] G. Bouffard, B. N. Thampi, J.-L. Lanet, Detecting Laser Fault Injection for Smart Cards Using Security Automata, in: International Symposium on Security in Computing and Communications, Vol. 377, Springer Berlin / Heidelberg, Mysore, India, 2013, pp. 18–29. doi:10.1007/978-3-642-40576-1_3.

[49] B. Nicolescu, Y. Savaria, R. Velazco, SIED: Software Implemented Error Detection, in: 18th International Symposium on Defect and Fault Tolerance in VLSI Systems, IEEE Computer Society, Boston, MA, USA, 2003, pp. 589–596. doi:10.1109/DFTVS.2003.1250159.

[50] SFR, (u)sim java card platform protection profile basic and scws configurations, Tech. rep., SFR and AFOM and Trusted Labs (2010).

[51] M.-L. Akkar, L. Goubin, O. Ly, Automatic Integration of Counter-Measures Against Fault Injection Attacks, in: Proceedings of E-Smart'2003, Nice, 2003, pp. 1–13.

[52] I. Levin, A byte-oriented AES-256 implementation (2007). URL http://www.literatecode.com/aes256

[53] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. R. Staple, G. Swamy, T. Villa, VIS: A system for verification and synthesis, in: R. Alur, T. A. Henzinger (Eds.), Computer Aided Verification, Vol. 1102 of LNCS, Springer, Heidelberg, New Brunswick, USA, 1996, pp. 428–432. doi:10.1007/3-540-61474-5_95.

[54] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, R. B. Brown, MiBench: A free, commercially representative embedded benchmark suite, in: 4th Annual Workshop on Workload Characterization, IEEE Computer Society, Austin, TX, 2001, pp. 3–14. doi:10.1109/WWC.2001.990739.

[55] Keil, Keil uVision for ARM processors (2012).

[56] F. Amiel, K. Villegas, B. Feix, L. Marcel, Passive and active combined attacks: Combining fault attacks and side channel analysis, in: Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography, IEEE Computer Society, Washington, DC, USA, 2007, pp. 75–79. doi:10.1109/FDTC.2007.10.