Software countermeasures for control flow integrity of smart card C codes

Jean-François Lalande[13], Karine Heydemann[2], and Pascal Berthomé[3]

[1] Inria, Supélec, CNRS, Univ. Rennes 1, IRISA, UMR 6074
35576 Cesson-Sévigné, France, jean-francois.lalande@insa-cvl.fr
[2] Sorbonne Universités, UPMC, Univ. Paris 06, CNRS, LIP6, UMR 7606
75005 Paris, France, karine.heydemann@lip6.fr
[3] INSA Centre Val de Loire, Univ. Orléans, LIFO, EA 4022
18022 Bourges, France, pascal.berthome@insa-cvl.fr

Abstract. Fault attacks can target smart card programs in order to disrupt an execution and gain an advantage over 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 as well as designing countermeasures at software level are tedious and tricky for developers. In this paper, we propose a methodology to detect harmful intra-procedural jump attacks at source code level and to automatically inject formally-proven countermeasures. The proposed software countermeasures defeat 100% of attacks that jump over at least two C source code statements or beyond. Experiments show that the resulting code is also hardened against unexpected function calls and jump attacks at assembly level.

Keywords: control flow integrity, fault attacks, smart card, source level

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, pin counters, etc. Malicious users aim to get access to these secrets by performing attacks on the secure elements. Fault attacks consists in disrupting the circuit's behavior by using a laser beam or applying voltage, clock or electromagnetic glitches [5, 6, 24]. Their goal is to alter the correct progress of the algorithm and, by analyzing the deviation of the corrupted behavior with respect to the original one, to retrieve the secret information [14]. For java card, fault attacks target particular components of the virtual machine [3, 4, 9].

Many protections have therefore been proposed to counteract attacks. Fault detection is generally based on spatial, temporal or information redundancy at hardware or software level. In java card enabled smart cards, software components of the virtual machine can perform security checks [18, 20, 10].

In practice, developers of security-critical applications often manually add countermeasures into an application code. This operation requires knowledge about the target code vulnerabilities. Both these tasks are time-consuming with direct impact on the certification of the product. One harmful consequence of fault attacks is control flow disruption which may bypass some implemented countermeasures. It is difficult for programmers to investigate all possible control flow disruption in order to detect sensitive parts of the code and then investigate how to add countermeasures inside these sensitive parts. Moreover, secure smart cards have strong security requirements that have to be certified by an independent qualified entity before being placed on the market. Certification can rely on a review of source code and the implemented software countermeasures. The effectiveness of software security countermeasures is then guaranteed by the use of a certified compiler [21]. Injecting control flow integrity checks at compile time would require to certify the modified compiler. To avoid this difficult and expensive task, countermeasures must be designed and inserted at a high code level.

In this paper, we propose a full methodology 1) to detect harmful attacks that disrupt the control flow of native C programs executed on a secure element and 2) to automatically inject formally verified countermeasures into the code. In the first step of our methodology, the set of harmful attacks is determined through an exhaustive search relying on a classification of attack effects from a functional point of view. The identified harmful attacks can be visualized spatially in order to identify the affected functions and to precisely locate the corresponding sensitive code regions. Following our methodology, a tool automatically injects countermeasures into the code to be protected without any direct intervention of a developer. The countermeasure scheme proposed in this paper operates at function level. Countermeasures rely on counters that are incremented and checked throughout execution enabling detection of any attack that disrupts control flow by not executing at least two adjacent statements of the code. The effectiveness of the proposed countermeasure scheme has been formally verified: any attack that jumps over more than two C statements is detected. This is confirmed by experimental results for three well-known encryption algorithms and additionally results show that 1) attacks are much more difficult to perform on the secured code and that 2) attacks trying to call an unexpected function are detected.

The paper is organized as follows: Section 2 discusses related work. Section 3 gives an overview of our methodology detecting application weaknesses and automatically securing an application code. Section 4 details the detection of weaknesses and visualization. Section 5 and 6 respectively presents the countermeasures for hardening a code against control flow attacks and the formal approach used for verifying their correctness. Section 7 presents experimental results.

2 Related work

This section discusses work related to fault models before presenting previously proposed countermeasures for smart card and control flow integrity.

2.1 Fault models

Countermeasures are necessarily designed with respect to a fault model specifying the type of faults an attacker is able to carry out [27]. Elaborating a fault model requires analysis of the consequences of physical attacks and modeling them at the desired level (architectural level, assembly code level, source code level). Consequences of fault attacks, at program level, include the processing of corrupted values by the program, a corrupted program control flow or a corrupted program as a result of changed instructions. In this paper, we focus on attacks that impact control flow of native C programs.

Several works [24, 2] have shown that attacks can induce instruction replacements. For example, electromagnetic pulse injections can induce a clock glitch on the bus during transmission of instruction from the Flash memory resulting in an instruction replacement [24]. Such an instruction replacement can provoke a control flow disruption in the two following cases:

1. 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 condition of a conditional branch instruction by only replacing the opcode in the instruction encoding has the same consequence.
2. The replacement of a whole instruction by a jump at any location of the program. The executed instruction becomes a jump to an unexpected target [9, 24]. The same effect is obtained if the target address of a jump is changed by corrupting the instruction encoding or, in case of indirect jump, if computation of the target address is disrupted. This also happens if the program counter becomes the destination operand of the replacing instruction, *e.g.* an ALU instruction such a `PC = PC +/- cst` which are the most likely to succeed into a correct jump.

In this paper, we consider jump attacks as described in the second case just above.

2.2 Code securing and control flow securing

Code securing techniques can be applied to the whole application or only to specific parts. Securing only sensitive code regions requires to know weaknesses which need to be strengthened for a given fault model. When considering some convenient fault models or when varying input, tractable static analysis, such as taint analysis, can be used to infer the impact of a fault on control flow [12] or to detect missing checks [29]. To the best of our knowledge, no previous work has considered a jump attack fault model probably due to its complexity: all possible jumps from one point of the program to another point have to be considered.

Protections against control flow attacks depend then 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 inserting redundancy and appropriate checks [5]. Countermeasures designed for ensuring control flow integrity or code integrity often rely on signature techniques or on checks to ensure the validity of accesses to the instruction memory or of the target address of jump instructions.

Signature techniques 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 with the expected result. Dedicated hardware [28, 15] can be used to compute signatures dynamically. However, solutions requiring hardware modification are unpractical for smart cards. Several works proposed to integrate the checks at software level [26, 13, 16]. Oh et al. [26] only checks the destinations of all jumps. Bletsch et al. [8] focus on return-oriented attacks. Abadi et al. [1] proposed a broader method for ensuring control flow integrity which checks for both the source and the destination of jumps. However, this approach relies on a new machine instruction.

For javacard enabled smart card, software components of the virtual machine can perform security checks. Basic block signature computations and checks can then be carried out by the virtual machine, as proposed by [18]. A transition automaton, in which each state corresponds to a basic block and each transition corresponds to allowed control flow, can also be built by analyzing the bytecode [10]. Calls to `setState()`, added to the source code, instruct the virtual machine to check the integrity of the control flow by comparing the current state with the allowed ones according to the automaton. The virtual machine can also check the validity of the bytecode address to avoid the execution of any bytecode stored outside the applet currently being executed [20]. However, a small jump inside the allowed bytecode, for example inside a function, would not be detected and might have serious consequences for security. These java card approaches rely on the ability to perform runtime checks during the interpretation of the bytecode. For native programs, it is mandatory to include any software countermeasure inside the code, as proposed in [26, 13, 16].

Approaches that verify the direction or the target address of branches or jumps only harden the control flow integrity at basic blocks boundaries. This is not sufficient to cover physical faults that cause an unconditional jump from an instruction inside a basic block to another instruction inside another basic block. The approach proposed in this paper enforces the control flow integrity with a granularity of one C statement, which is finer than basic block granularity.

Two previously proposed approaches also use a step counter to protect a code region [10, 25]. The former targets computation disruption while the latter combined counters with a signature approach at assembly level to ensure tolerance to hardware fault. The use of a certified compiler requires to work at the source code level as proposed in this paper. Our approach, based on counters, is similar to the intra basic block approach of [25] for securing sequential code. But our approach operates at higher code level and is able to harden control flow of high level constructs.

Thus, 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 C level that ensure control flow integrity in the presence of jump attacks during native execution.

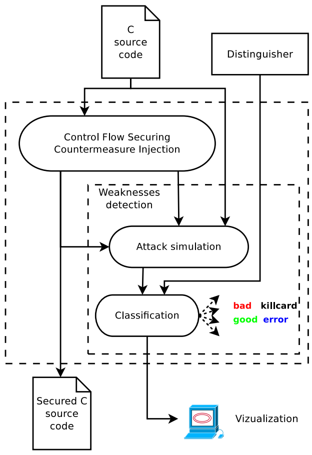

Fig. 1. Code securing methodology

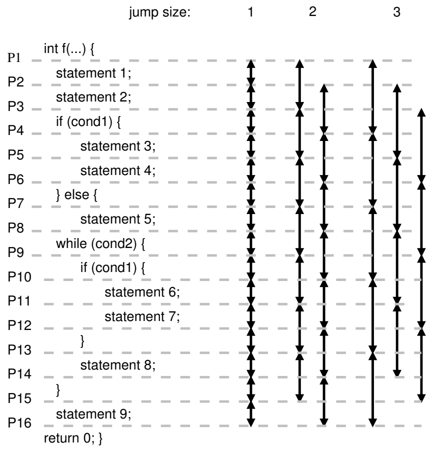

Fig. 2. Injection of jump attacks at C level

3 Weakness detection and code securing; overview

This section gives an overview of our methodology, supported by tools and represented in Figure 1, to help developers to improve the security of smart card C codes subject to physical attacks inducing jumps. Our approach starts with a functional C code of the application. A distinguisher is necessary in order to discriminate between an execution that gives advantage to an attacker and an execution that does not. The full code security analysis and securing methodology can be split into two main steps.

The first step identifies the weaknesses in the code by simulating all possible attacks during code execution. The output of all these executions is passed to a classification tool that classifies attacks as harmful or as harmless. We refer to these classes as *bad* and *good*, respectively. While the bad class contains all attacks that give advantages to the attacker, the good class is the set of attacks that have no effect from a security point of view.

The second step consists in hardening the code. It relies on automated injection of countermeasures that ensure control flow integrity for any function that has been successfully attacked in the first step. The design of the countermeasures is detailed in Section 5 and the verification of their correctness is explained in Section 6. Note that, once a code has been secured, the evaluation of the efficiency of the implemented countermeasures or the identification of remaining weaknesses can be achieved by returning to the weaknesses detection step.

We have also implemented a visualization tool offering a graphical representation of the attacks classification and enabling to have a look at the code regions corresponding to harmful attacks.

4 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 physical attacks at source code level.

4.1 Simulation of attacks

Motivations to work at source code level. As code securing is often performed at source level by developers, simulating attacks at this level allows to identify the harmful ones as well as the code regions that should be secured. Simulating attacks at assembly level would require to match assembly instructions with the source code which is not trivial [7]. Furthermore, assembly programs are tightly coupled to specific architectures. Thus, simulating attacks at assembly programming limits portability. It is also time-consuming. Indeed, simulation of attack injection at the source code level speeds up the detection of weaknesses compared to injection at assembly level due to the lower number of source statements. However, jump attacks that start/arrive inside a C statement cannot be simulated at C level [7]. Nevertheless, it is helpful to detect as many weaknesses as possible at source code level, and as we show in the experimental results, working at this level enables to strengthen code security by making successful attacks very difficult to perform.

Simulation of C level attacks. In order to discover harmful attacks, we simulate jump attacks using software hacks at C level, as proposed in [7]. For each function of the application, all possible intra-procedural jump attacks that jump backward or forward C statements are injected at source level. Figure 2 illustrates all possible jumps within a function, sorted according to their 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{` or the bracket between P14 to P15) that impact control flow.

4.2 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). In order to cover the various benefits for an attacker in a general way, our methodology requires a distinguisher to be provided. This distinguisher must be able to classify as bad any execution where an attack has succeeded in breaking the expected security property of the code. Other attacks can be assigned to the good 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: *bad*: during execution a benefit has been obtained by the attacker; *good*:

```
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  }
```

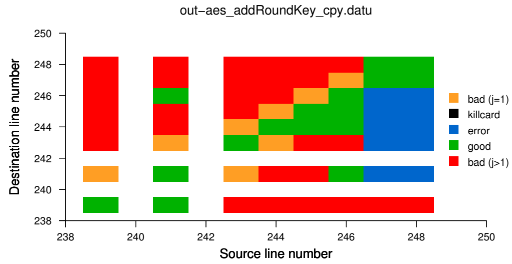

Fig. 3. Result of weakness detection for the aes_addRoundKey_cpy function

the behavior of the application remains unchanged; *error* or *timeout*: the program has seemed to not terminate and has to be killed or finished with an error message, a signal (SIGSEGV, SIGBUS, ...) or crashes; *killcard*: a countermeasure has detected an attack and has triggered a security protection to terminate the program and possibly to destroy the card. We assume that no benefit can be obtained by an erroneous or endless execution, so both *error* and *timeout* cases are distinguished from *bad* cases in the remainder of the paper. If an error is preceded by a gain, such as a leak of sensitive information, the distinguisher must be able to discriminate between these attack effects.

4.3 Weaknesses analysis and visualization

Since our securing scheme operates at function level, the detection of weaknesses aims at identifying harmful attacks at source code level in order to identify the functions to be secured. Thus, any function that, when attacked, exhibits a *bad* case is 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 quickly understand which variables and functionalities are involved in the generation of harmful attacks by analyzing the jumped part of the code.

The visualization tool builds a graphical representation of the results of the identification of weaknesses by drawing a square at the coordinate (source_line, target_line) using the color associated to its class. To illustrate this, consider the function `aes_addRoundKey_cpy` of an implementation of AES-256 [22] in C, used later in experiments, given in Figure 3. The distinguisher considers as *bad* any execution producing incorrect encrypted data, representing the attacker's ability to disrupt the encryption. The visualization of the weaknesses is illustrated in the right part of Figure 3. All except one forward jump generate a *bad* case (orange squares correspond to a jump size of one statement, red squares to a larger jump distance). Analyzing statements impacted by these harmful attacks shows that the whole loop body, hence the whole function, must be secured.

```
#define DECL_INIT(cnt, x) int cnt; if ((cnt = x) != x) killcard();
#define CHECK_INCR(cnt, x) cnt = (cnt == x ? cnt +1 : killcard());
#define CHECK_INCR_FUNC(cnt1, x1, cnt2, x2) cnt1 = ((cnt1 == x1) && (cnt2 == x2) ? cnt1 +
      1 : killcard());
#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();
#define CHECK_INCR_COND(b, cnt, val, cond) (b = (((cnt)++ != val) ? killcard() : cond))
#define RESET_CNT(cnt_while, val) cnt_while = !(cnt_while == 0 || cnt_while == val) ?
     killcard() : 0;
#define CHECK_LOOP_INCR(cnt, x, b) cnt = (b && cnt == x ? cnt +1 : killcard());
#define CHECK_END_LOOP(cnt_while, b, val) if ( ! (cnt_while == val && !b) ) killcard();
```

Fig. 4. Security macros used for control flow securing

5 Countermeasure for C code securing

In this section, we present the countermeasures we have designed to detect jump attacks with a distance of at least two C statements. These countermeasures deal with the different high-level control-flow constructs such as straight line flow, if-then-else and loops. Countermeasures are presented in C-style in Appendix A and use the macros shown in Figure 4. Note that all macros are expanded to only one line of source code.

5.1 Protection of a function and straight-line flow of statements

To secure the control flow integrity of a whole function or a whole block of straight-line statements, our securing scheme uses a dedicated counter. Each function or each block of sequential code has its own counter to ensure its control flow integrity. Counters are incremented after each C statements of the original source code using the `CHECK_INCR` macro. Before any incrementation, a check of the expected value of the counter is performed. When a check fails, a handler named, `killcard()` as the one used in smart card community, stops the execution.

To ensure control flow integrity, checks and incrementations of counters need to be nested. Consider the example in Figure 5 that illustrates the countermeasure for a function `g` with a straight-line control flow composed of N statements. The dedicated statement 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 to the counter declaration is *surrounded* by two checks and incrementations of the counter `cnt_f` dedicated to the block of the function `f` where `g` is called. Moreover, the initialization value is different for each function which enables the detection of any call to another function as shown in the experiments. A reference to the counter `cnt_g` is passed to `g` as an extra parameter. Upon return from `g`, a check of the values of both counters `cnt_f` and `cnt_g` is performed in order to detect any corruption of the flow inside the function `g`.

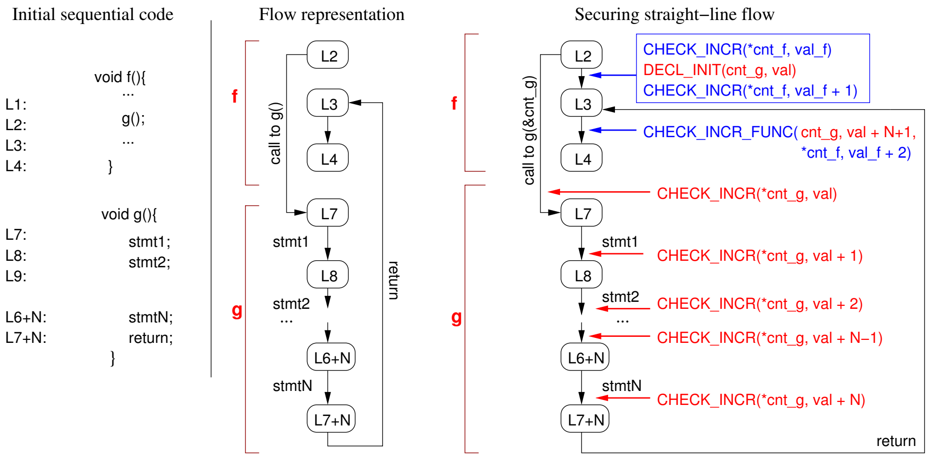

Fig. 5. Securing function call and straight-line flow

This way, any jump to the beginning of the function is detected inside the called function `g`. Any jump to the end of a function is caught when the control flow returns to the calling function. The nesting of counter checks is at the core of our countermeasure scheme ensuring control flow integrity.

5.2 Conditional if-then and if-then-else constructs

High level conditional control flow refers to if-then or if-then-else constructs, illustrated by the example on the left part of Figure 6. The securing scheme for conditional flow is illustrated in the right part of the Figure. For such a construct, our securing scheme requires 2 counters `cnt_then` and `cnt_else` (for the control flow integrity of each branch of the conditional construct) and one extra variable `b` to hold the value of the condition of the conditional flow. 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 interleaved with checks and incrementations of the counter `cnt` used for the control flow of the surrounding block. This is performed by the additionnal statements in the red box on the upper right part 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, in order to be able to distinguish, after the execution of the if-then-else construct, which branch has been taken. Both counters dedicated to the conditional branches are then checked according to the value of `b`. This is performed by the code corresponding to the `CHECK_END_IF_ELSE`

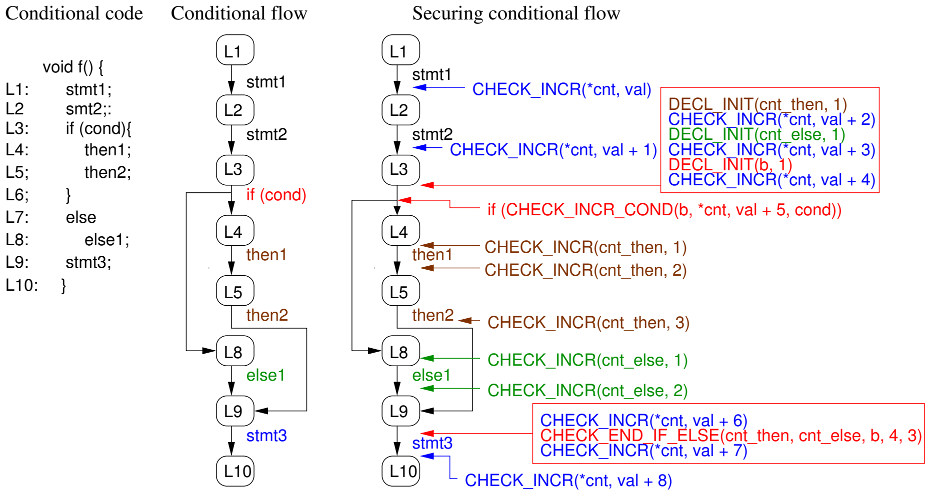

Fig. 6. Securing conditional control flow

macro inserted between two checks of the counter `cnt`. Again, this nesting of counter checks is at the core of the effectiveness of our countermeasure scheme.

5.3 Loop constructs

We have also designed a countermeasure scheme for loops. Due to lack of space, we only present `while` loops. Any other loop constructs (`for`, `do while`) can be rewritten into a `while` construct. The left part of Figure 7 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. The variable `b` is needed at the end of the loop to verify correct execution of the loop body and correct termination of the loop. This is performed by the `CHECK_END_LOOP` macro which is surrounded by `CHECK_INCR` of the counter `cnt`. The `b` variable is declared and initialized outside the loop as for 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 initial iteration using the `RESET_CNT(cnt_while, val)` macro with `val` being the final value of the counter after one iteration. The reset is performed only if `cnt_while` is equal to 0 or to the value `val` that is expected after one complete iteration. 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 a jump attack leading to an additional

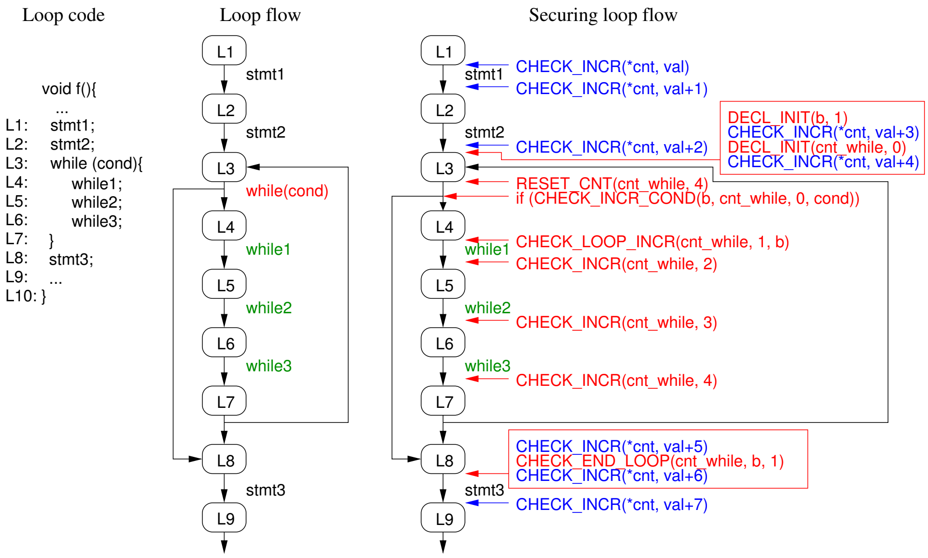

Fig. 7. Securing loop control flow

iteration of the loop. Moreover, the evaluation of the condition (that may update an induction variable) is performed along with a check and an incrementation 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.

We have also designed a countermeasure scheme for other C constructs such as `switch case`, `break`, multiple `returns`, `goto`. Due to space limitations and their absence in applications we have considered, they are not presented here.

6 Formal verification of countermeasures

Formal verification of our securing scheme helped us designing effective countermeasures and gives strong confidence in their effectiveness against attacks. In this section, we present the models used for program execution from a control flow point of view and for jump attacks, as well as properties to check to ensure the control flow integrity of a secured program execution even in presence of attacks. The verification of the correctness of the secured code is based on an equivalence checking with the original code.

6.1 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 variables of the program (contents of the memory) and of the program counter 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 program counter and potentially changing variables or the contents of memory. Figure 8 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 or if-then-else). Sequential execution of these regions guarantees that, if the control flow integrity is ensured at the end of a code region, the following code region starts with a correct input from a control flow point of view. Thus, the integrity of the control flow of both code regions can be proven by proving the control flow integrity of each code region. Our countermeasure scheme relies on securing each control flow construct (function call/sequential code, if-then-else, while constructs) nested with few straight-line statements of the surrounding block. Then, our approach consists in verifying separately for each control flow construct enclosed with straight-line statements of the surrounding block that all possible executions of the secured version are stopped by a countermeasure in 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 which is correctly executed or not. Thus, in the models used for verification 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.

6.2 Models for verification of control flow integrity

State machine model. To model and verify the integrity of the control flow, we associate to each statement `stmt_i` of the original code of a function α a dedicated verification counter denoted `cntv_`α`i`. In the remainder of the paper, we refer to such counters as *statement counter*. 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 the set of transitions $T : S \to S$, 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 program counter and the value of statement counters associated to every statement of the initial code. L is a set of labels corresponding to the possible values of the program counter, *i.e.* line numbers in the source code. Initial states are states with a program counter value equal to the first line of the modeled code and where all statement counters hold 0. Any transition from T is defined by the effect of the statement `stmt_i` associated to the program counter value. Transitions change the program counter value to the next line number to be executed and increment the

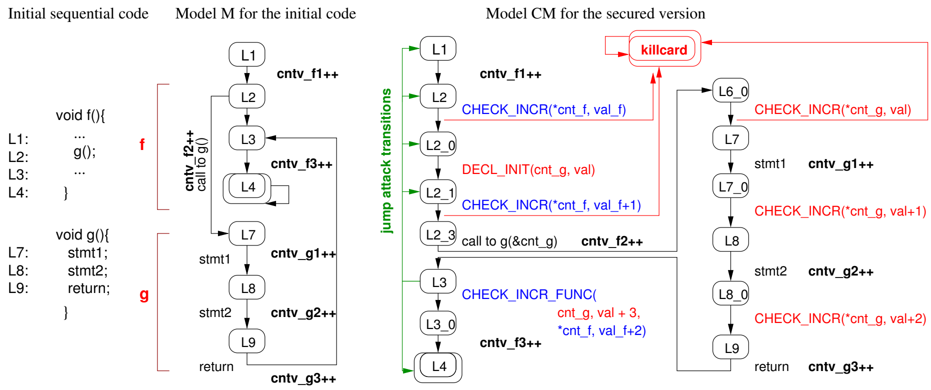

Fig. 8. Compact representation of TS for a function call and straight-line statements

statement counter `cntv_`α`i` associated to `stmt_i` of function α. A jump attack, as considered, can only corrupt the program counter. Thus, for such attacks, modeling the memory and other registers is not relevant.

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 build two transition systems: one for the initial control-flow construct named `M(c)` and another one for the version including countermeasures named `CM(c)`. Figure 8 illustrates a compact representation of both transition systems for a generic example code with a call to a function composed of straight-line statements. In a secured version, checks may result in a call to `killcard()`. Hence, there is an additional program counter value denoted `killcard` in any `CM(c)`. All states with this program counter value are final. All transitions labeled with a countermeasure macro may change the program counter to `killcard`. Due to the high number of such transitions, only a subset is represented in order to keep the Figure readable.

Jump attack model. A jump attack is equivalent to the modification of the program counter with an unexpected value. As our countermeasures are effective against attacks that jump at least two lines, we add faulty transitions between every pair of states of `CM(c)` separated by at least one line of C. These transitions only update the program counter. The green arrows in `CM(c)` in Figure 8 illustrate all possible jump attacks occurring at line 3 of the code. As we assume that only a single fault can occur, every fault transition is guarded with a boolean indicating that a fault has already occurred.

6.3 Specification of control flow integrity and equivalence checking

To perform the verification of the control flow integrity and the correctness of a secured code, we connect the two transition systems `M(c)` and `CM(c)` in order to force the input (such as condition value, iteration counts), if any, to be identical. The model checker builds a product of both models. We explain in this section, the properties to verify on this product.

To verify the correctness of the secured code, in the presence or absence of an attack, we need to prove that:

1. Any path in `M(c)` or `CM(c)` reaches a final absorbing state.
2. The statement counter values in any final correct state in `CM(c)` (with a program counter value different from `killcard`) are equal to the statement counter values in final states of `M(c)`.
3. In `CM(c)` at any time and in any path, counters `cntv_`α`i` and `cntv_`α`(i+1)` for two adjacent statements `stmt_i` and `stmt_i+1` in a straight-line flow respects $1 \geq$ `cntv_`α`i` $\geq$ `cntv_`α`(i+1)` ≥ 0 or execution will reach a final state with the `killcard` value for the program counter.

Property 2 ensures the right execution counts of statements in `CM(c)` if the execution reaches a correct final state. Property 3 checks that line `i` is always executed before line `i+1` and after line `i-1` or the execution will reach a killcard state. By transitivity, this property, if verified by all statement counters, ensures that in a straight-line flow, the statement of line `k` is always executed after line `i` and before line `l` with `i` $<$ `k` $<$ `l` both in `M(c)` and `CM(c)`. Hence, Property 3 ensures the right order of execution of statements.

For a conditional flow or for a loop, Properties 1 and 2 are the same but the Property 3 changes slightly. For a conditional flow, such as the example in Figure 6, Property 3 specifies that at any time in an execution path that reaches a correct final state 1) the straight-line flow before and after the branches, condition included, is correct and 2) inside both branches, condition included, the straight-line flow is correct. Property 2 ensures that only one branch is executed. For a while loop as in Figure 7, Property 3 says that at any time in an execution that reaches a correct final state 1) the statements before and after the loop (condition excluded) are executed only once and in order; 2) the condition is never executed before its preceding statements, 3) statements inside the loop, condition included, are executed in order but their execution counts is not limited. Property 2 ensures the right number of iterations and Property 3 ensures the right control flow during execution.

We have chosen the Vis model checker [11] to prove the effectiveness of our countermeasure scheme. After modeling all the constructs given in this section and expressing the properties in CTL, all properties hold.

7 Experimental results

We implemented all software components[4] presented in Figure 1. The countermeasures for control flow securing as well as the jump attacks for the detection of weaknesses are injected using a python C parser that manipulates the C instructions. For the experiments, we considered three well-known encryption algorithms available in C: AES [22], SHA [17] and Blowfish [17].

[4] A demonstration video is available at: `http://dai.ly/x205n3x`

Table 1. Jump attack classification for original and secured version (+ CM)

	BAD size > 1	BAD size = 1	GOOD	KILLCARD	ERROR	TOTAL
C JUMP ATTACKS	Attacking all functions at C level for all transient rounds					
AES	7786 29%	1104 4.2%	17 372 65%		108 0.4%	26 370
AES + CM	0	528 0.2%	18 015 5.3%	318 972 94%	1 0.0%	337 516
SHA	32 818 75%	1528 3.5%	8516 19%		412 1.0%	43 274
SHA + CM	0	1149 0.3%	5080 1.2%	421 200 98%	261 0.1%	427 690
Blowfish	70 086 32%	3550 1.7%	134 360 62%		5725 2.7%	213 721
Blowfish + CM	0	2470 0.2%	331 664 23%	1 060 156 75%	6065 0.4%	1 400 355
ASM JUMP ATTACKS	Attacking the aes_encrypt function at ASM level for the first transient round					
aes_encrypt	1566 82.8%	36 1.9%	179 9.4%		111 5.9%	1892
aes_encrypt + CM	627 0.2%	21 0%	63 040 20.2%	239 303 78.4%	2264 0.7%	305 255
ASM CALL ATTACKS	Attacking all function calls at ASM level for the first transient round					
AES	249 59.3%		139 33.1%		32 5%	420
AES + CM	0		21 5%	398 94.8%	1 0.2%	420
SHA	35 48.7%		13 18%		24 33.3%	72
SHA + CM	0		8 11.1%	61 84.7%	3 4.2%	72
Blowfish	9 21.4%		18 42.9%		15 35.7%	42
Blowfish + CM	0		18 42.9%	17 40.5%	7 16.6%	42

First, we simulated all the intra-procedural jump attacks at C level for each function (such as in Figure 2). A simulated attack is transient and triggered once during execution. However, using the `gcov` tool to determine how many times each line is executed, we simulated all possible instances of jump attacks from a Line i to a Line j. In all our experiments, the distinguisher classifies as *bad* any attack that provokes program termination with corrupted output. The second column of Table 1 shows that all attacks with a jump distance greater than or equal to two C statements are captured by our countermeasures. For example, 32 818 jump attacks were harmful for SHA whereas none was for its secured version (SHA + CM). The number of attacks jumping only one C statements is also reduced (third column). More important, the ratio of the remaining jump attacks of size one becomes very low ($\leq$ 0.3%). For example, for AES the *bad* cases of size one decrease from 33.2% to 0.2%.

Also, we simulated all possible intra-procedural jump attacks at assembly level targeting the `aes_encrypt` function of AES executed by an ARM Cortex-M3 processor. We used the Keil ARM-MDK compiler and Keil simulator [19] for the replacement of any instruction by a jump anywhere into the same function. We considered only one function due to a very long simulation time (3 weeks), highlighting the benefits to perform the attack simulation at source level. Results are presented in the ASM JUMP ATTACKS section of Table 1. The harmful attacks in the secured version represent only 40% of the ones in the original code: our countermeasures enable to defeat 60% of the attacks on this example. Moreover, only 0.2% of attacks give advantage to the attacker while 78.4% are detected. Thanks to the frequent checks added by our countermeasures, the harmful attacks are much harder to perform on the secured code. It shows our countermeasures are effective while being implemented at source code level.

Finally, we simulated attacks that call an unexpected function instead of the expected one for all the benchmarks using the Keil simulator. Results, presented

Table 2. Size and overhead for original and secured version (+ CM)

	x86					ARM-V7M			
	Simulation time	Size bytes	overhead	Execution time time	overhead	Size bytes	overhead	Execution time time	overhead
AES	27m	17 996		1.27 ms		4216		38.3 ms	
AES + CM	9h 46m	30 284	(+68%)	2.61 ms	(+106%)	15 696	(+272%)	191.7 ms	(+400.5%)
SHA	1h 18m	13 235		1.47 µs		3184		106.5µs	
SHA + CM	16h 52m	21 702	(+64%)	2.81 µs	(+91%)	7752	(+143%)	499.1µs	(+368%)
Blowfish	5h 52m	30 103		47.6 µs		6292		3.02 ms	
Blowfish + CM	3d 6h 19m	46 680	(+55%)	70.6 µs	(+48%)	16 396	(+161%)	6.3 ms	(+109%)

in the ASM CALL ATTACKS section of Table 1, show that all harmful attacks are captured and many harmless attacks are also detected. Thus, our countermeasures are also very effective against unexpected function calls.

Table 2 reports code sizes as well as execution times of both the original version and the secured one, for a x86 target machine and a cortex-M3 processor. For the x86 platform, the execution time overhead ranges from +59% (blowfish) up to +106% (AES). For the embedded ARM processor, the overhead is higher as the simpler processor does not exploit instruction level parallelism. The highest overhead is also achieved for AES (+400%). As all functions of our benchmarks exhibit vulnerabilities, they were all fully secured by our methodology. We will consider in future work how to achieve at least the same level of security without fully securing sensitive functions. However, as a smart card is primarily the host of sensitive operations, ensuring the required security level is crucial. Full code securing often implies such a high overhead [23, 25].

8 Conclusion

This paper has presented a methodology to automatically secure any C application with formally verified countermeasures at source level. Results has shown that these countermeasures defeat 100% of C jump attacks with a distance of two statements or beyond. Moreover, our countermeasures are able to capture all unexpected function calls. They also have been able to reduce significantly the number of attacks injected at assembly level: for the studied function, 60% of the assembly jump attacks were eliminated. Future work will address the optimization of countermeasure injection according to the weaknesses detection step. If harmless attacks are found inside a function, countermeasures might be adapted accordingly to reduce their cost while preserving their effectiveness.

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: Atluri, V., Meadows, C., Juels, A. (eds.) 12th ACM conference on Computer and communications security. pp. 340–353. ACM Press, Alexandria, USA (Nov 2005)
2. Balasch, J., Gierlichs, B., Verbauwhede, I.: An in-depth and black-box characterization of the effects of clock glitches on 8-bit MCUs. In: Breveglieri, L., Guilley,

S., Koren, I., Naccache, D., Takahashi, J. (eds.) The 8th Workshop on Fault Diagnosis and Tolerance in Cryptography. pp. 105–114. IEEE Computer Society, Nara, Japan (Sep 2011)

3. Barbu, G., Duc, G., Hoogvorst, P.: Java card operand stack: fault attacks, combined attacks and countermeasures. In: Prouff, E. (ed.) 10th Smart Card Research and Advanced Application IFIP Conference. LNCS, vol. 7079, pp. 297–313. Springer, Heidelberg, Leuven, Belgium (Sep 2011)
4. Barbu, G., Thiebeauld, H., Guerin, V.: Attacks on java card 3.0 combining fault and logical attacks. In: Gollmann, D., Lanet, J.L., Iguchi-Cartigny, J. (eds.) 9th Smart Card Research and Advanced Application IFIP Conference. LNCS, vol. 6035, pp. 148–163. Springer, Heidelberg, Passau, Germany (Apr 2010)
5. Barenghi, A., Breveglieri, L., Koren, I., Naccache, D.: Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures. Proceedings of the IEEE 100(11), 3056–3076 (Nov 2012)
6. Barenghi, A., Trichina, E.: Fault attacks on stream ciphers. In: Joye, M., Tunstall, M. (eds.) Fault Analysis in Cryptography, pp. 239–255. Information Security and Cryptography, Springer, Heidelberg (2012)
7. Berthomé, P., Heydemann, K., Kauffmann-Tourkestansky, X., Lalande, J.F.: High level model of control flow attacks for smart card functional security. In: 7th International Conference on Availability, Reliability and Security AReS 2012. pp. 224–229. IEEE Computer Society, Prague, Czech Republic (Aug 2012)
8. Bletsch, T., Jiang, X., Freeh, V.: Mitigating code-reuse attacks with control-flow locking. In: Zakon, R.H., McDermott, J.P., Locasto, M.E. (eds.) 27th Annual Computer Security Applications Conference. pp. 353–362. ACM Press, Orlando, FL, USA (Dec 2011)
9. Bouffard, G., Iguchi-Cartigny, J., Lanet, J.L.: Combined software and hardware attacks on the java card control flow. In: Prouff, E. (ed.) 10th Smart Card Research and Advanced Application IFIP Conference. LNCS, vol. 7079, pp. 283–296. Springer, Heidelberg, Leuven, Belgium (Sep 2011)
10. Bouffard, G., Thampi, B.N., Lanet, J.L.: Detecting laser fault injection for smart cards using security automata. In: Thampi, S.M., Atrey, P.K., Fan, C.I., Pérez, G.M. (eds.) International Symposium on Security in Computing and Communications. CCIS, vol. 377, pp. 18–29. Springer, Heidelberg, Mysore, India (Aug 2013)
11. Brayton, R., Hachtel, G., Sangiovanni-Vincentelli, A., Somenzi, F., Aziz, A., Cheng, S.T., Edwards, S., Khatri, S., Kukimoto, Y., Pardo, A., Qadeer, S., Ranjan, R., Sarwary, S., Staple, T., Swamy, G., Villa, T.: Vis: A system for verification and synthesis. In: Alur, R., Henzinger, T. (eds.) Computer Aided Verification, Lecture Notes in Computer Science, vol. 1102, pp. 428–432. Springer, Heidelberg (1996), `http://vlsi.colorado.edu/~vis/`
12. Ceara, D.: Detecting Software Vulnerabilities - Static Taint Analysis. Bsc thesis, Universitatea Politehnica Bucuresti, Verimag (2009)
13. Chen, Y., Venkatesan, R., Cary, M., Pang, R., Sinha, S., Jakubowski, M.H.: Oblivious hashing: A stealthy software integrity verification primitive. In: Petitcolas, F.A.P. (ed.) 5th International Workshop on Information Hiding. LNCS, vol. 2578, pp. 400–414. Springer, Heidelberg, Noordwijkerhout, The Netherlands (Oct 2002)
14. Dehbaoui, A., Mirbaha, A.P., Moro, N., Dutertre, J.M., Tria, A.: Electromagnetic glitch on the AES round counter. In: Prouff, E. (ed.) 4th International conference on Constructive Side-Channel Analysis and Secure Design. LNCS, vol. 7864, pp. 17–31. Springer, Heidelberg, Paris, France (Mar 2013)

15. Fiskiran, A.M., Lee, R.B.: Runtime execution monitoring (REM) to detect and prevent malicious code execution. In: IEEE International Conference on Computer Design: VLSI in Computers and Processors. pp. 452–457. IEEE Computer Society, San Jose, CA, USA (Oct 2004)
16. Goloubeva, O., Rebaudengo, M., Reorda, M.S., Violante, M.: Soft-error detection using control flow assertions. In: 18th International Symposium on Defect and Fault Tolerance in VLSI Systems. pp. 581–588. IEEE Computer Society, Boston, MA, USA (Nov 2003)
17. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: MiBench: A free, commercially representative embedded benchmark suite. In: 4th Annual Workshop on Workload Characterization. pp. 3–14. IEEE Computer Society, Austin, TX, USA (Dec 2001), `http://www.eecs.umich.edu/mibench/`
18. Iguchi-cartigny, J., Lanet, J.L.: Evaluation of Countermeasures Against Fault Attacks on Smart Cards. International Journal of Security and Its Applications 5(2), 49–60 (2011)
19. Keil: Keil uVision for ARM processors (2012), `http://www.keil.com/support/man_arm.htm`
20. Lackner, M., Berlach, R., Raschke, W., Weiss, R., Steger, C.: A defensive virtual machine layer to counteract fault attacks on java cards. In: Cavallaro, L., Gollmann, D. (eds.) Workshop in Information Security Theory and Practice Series. LNCS, vol. 7886, pp. 82–97. Springer, Heidelberg, Heraklion, Crete, Greece (May 2013)
21. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Morrisett, J.G., Jones, S.L.P. (eds.) 33rd ACM symposium on Principles of Programming Languages. pp. 42–54. ACM Press, Charleston, SC, USA (2006)
22. Levin, I.: A byte-oriented AES-256 implementation (2007), `http://www.literatecode.com/aes256`
23. Moro, N., Heydemann, K., Encrenaz, E., Robisson, B.: Formal verification of a software countermeasure against instruction skip attacks. Journal of Cryptographic Engineering pp. 1–12 (2014)
24. Moro, N., Dehbaoui, A., Heydemann, K., Robisson, B., Encrenaz, E.: Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller. In: Workshop on Fault Diagnosis and Tolerance in Cryptography. pp. 77–88. IEEE Computer Society, Santa Barbara, CA, USA (Aug 2013)
25. Nicolescu, B., Savaria, Y., Velazco, R.: SIED: Software implemented error detection. In: 18th International Symposium on Defect and Fault Tolerance in VLSI Systems. pp. 589–596. IEEE Computer Society, Boston, MA, USA (Nov 2003)
26. Oh, N., Shirvani, P., McCluskey, E.: Control-flow checking by software signatures. IEEE Transactions on Reliability 51(1), 111–122 (Mar 2002)
27. Verbauwhede, I., Karaklajić, D., Schmidt, J.M.: The fault attack jungle - a classification model to guide you. In: Breveglieri, L., Guilley, S., Koren, I., Naccache, D., Takahashi, J. (eds.) 8th Workshop on Fault Diagnosis and Tolerance in Cryptography. pp. 3–8. IEEE Computer Society, Nara, Japan (Sep 2011)
28. Xia, Y., Liu, Y., Chen, H., Zang, B.: CFIMon: Detecting violation of control flow integrity using performance counters. In: Swarz, R.S., Koopman, P., Cukier, M. (eds.) IEEE/IFIP International Conference on Dependable Systems and Networks. pp. 1–12. IEEE Computer Society, Boston, MA, USA (Jun 2012)
29. Yamaguchi, F., Wressnegger, C., Gascon, H., Rieck, K.: Chucky: exposing missing checks in source code for vulnerability discovery. In: Sadeghi, A.R., Gligor, V.D., Yung, M. (eds.) ACM Conference on Computer and Communications Security. pp. 499–510. Berlin, Germany (Nov 2013)

A Countermeasures securing codes

This section presents the implementation of the countermeasures of Figure 5, 6 and 7 using the macros of Figure 4.

The implementation of countermeasures for function calls and sequence of statements are shown in the two listings below. The statement counter `cnt_g` for a function `g` must be initialized with a value different that the one for all other functions in order to capture jump attacks that try to call another function. Moreover, the range of values taken by the statement counter of a function must be different from the ones of other functions in order to detect inter-procedural jumps.

```
int g(int n, int m, int * cnt_g){
CHECK_INCR(*cnt_g, 8)
statement;
CHECK_INCR(*cnt_g, 9)
...
CHECK_INCR(*cnt_g, 10)
statement;
CHECK_INCR(*cnt_g, 11)
return res;
}
```

```
int f(int * cnt_f){
...
CHECK_INCR(*cnt_f, 15)
// initialization value ≠ for each func.
DECL_INIT(cnt_g, 8)
CHECK_INCR(*cnt_f, 16)
x = g(p, q, &cnt_g)
CHECK_INCR_FUNC(*cnt_f, 17, cnt_g, 12);
...
}
```

The listings below illustrate the implementation of the countermeasures for an if construct (left) and for a while construct (right). The while construct example contains the statements to be inserted to handle a `for` construct for the initialization of the induction variable and its incrementation.

```
...
CHECK_INCR(*cnt, 8)
statement;
CHECK_INCR(*cnt, 9)
DECL_INIT(cnt_then, 1)
CHECK_INCR(*cnt, 10)
DECL_INIT(cnt_else, 1)
CHECK_INCR(*cnt, 11)
DECL_INIT(b, 1)
CHECK_INCR(*cnt, 12)
if (CHECK_INCR_COND(b, *cnt, 13, cond))
{
  CHECK_INCR(cnt_then, 1)
  statement;
  CHECK_INCR(cnt_then, 2)
  ...
  CHECK_INCR(cnt_then, 4)
}
else
{
  CHECK_INCR(cnt_else, 1)
  statement;
  CHECK_INCR(cnt_else, 2)
  ...
  CHECK_INCR(cnt_else, 6)
}
CHECK_INCR(*cnt, 14)
CHECK_END_IF_ELSE(cnt_then, cnt_else, b, 5,
      7)
CHECK_INCR(*cnt, 15)
statement;
```

```
...
CHECK_INCR(*cnt, 8)
statement;
CHECK_INCR(*cnt, 9)
DECL_INIT(b, 1)
CHECK_INCR(*cnt, 10)
DECL_INIT(cnt_while, 1)
CHECK_INCR(*cnt, 11)
// optional induction variable
// initialization statement for a for
CHECK_INCR(cnt, 12)
while: {
  RESET_CNT(cnt_while, 8)
  if (! CHECK_INCR_COND(b, cnt_while, 0,
        cond)) goto next;
  CHECK_LOOP_INCR(cnt_while, 1, b)
  statement;
  CHECK_INCR(cnt_while, 2)
  statement;
  CHECK_INCR(cnt_while, 3)
  ...
  CHECK_INCR(cnt_while, 6)
  // optional incrementation statement
  // for a for
  CHECK_INCR(cnt_while, 7)
  goto while;
}
next:
CHECK_INCR(*cnt, 13)
CHECK_END_LOOP(cnt_while, b, 1)
CHECK_INCR(*cnt, 14)
statement;
```