# SAMVA: Static Analysis for Multi-Fault Attack Paths Determination

Antoine Gicquel, Damien Hardy, Karine Heydemann, Erven Rohou

# SAMVA: Static Analysis for Multi-Fault Attack Paths Determination

Antoine Gicquel[1], Damien Hardy[1], Karine Heydemann[2], and Erven Rohou[1]

[1] Univ Rennes, Inria, CNRS, IRISA, France
firstname.lastname@{inria,irisa}.fr
[2] Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
Thales DIS, France
firstname.lastname@lip6.fr

**Abstract.** Multi-fault injection attacks are powerful since they allow to bypass software security mechanisms of embedded devices. Assessing the vulnerability of an application while considering multiple faults with various effects is an open problem due to the size of the fault space to explore. We propose SAMVA, a framework for efficiently searching vulnerabilities of applications in presence of multiple instruction-skip faults with various widths. SAMVA relies solely on static analysis to determine attack paths in a binary code. It is configurable with the fault injection capacity of the attacker and the attacker's objective. We evaluate the proposed approach on eight PIN verification programs containing various software countermeasures. Our framework finds numerous attack paths, even for the most hardened version, in very limited time.

**Keywords:** Fault Injection Attack · Multi-Fault · Static Analysis

## 1 Introduction

Fault injection attacks are a major concern for embedded systems since they allow an attacker to overcome security mechanisms in order to retrieve secret data or take over a device. To inject a fault, a physical perturbation must be introduced in the circuit during the execution of the target program. Literature covers various means of injection [26], such as laser beams, electromagnetic pulses, voltage or clock glitching. Throughout a fault propagation mechanism, perturbations introduced at the hardware level impact the nominal execution of the running program, corrupting variables, control-flow of the program, or both.

To protect against fault injections, several countermeasures have been proposed. At software level, they often rely on redundancy [23]: sensitive checks or computations are duplicated; constant values are encoded such that it is difficult to change their value consistently. Also, some variables are added in order to monitor the executed path and check its validity with respect to the original program [13]. As a consequence, attackers must inject multiple faults and/or faults that impact several consecutive instructions in order to bypass countermeasures

and reach their objectives. Recent works have shown that it is possible to inject multiple faults [7] (i.e. at different instants) and to corrupt several consecutive instructions [20,5,11,15,8,6]. Faults can have a width varying from a few up to more than one hundred consecutively executed instructions. Therefore, multiple and wide faults are now considered as a real threat and system security against fault attacks must be evaluated considering such attacker capacity.

Security assessment eventually relies on real fault injection campaigns. However, some analysis dedicated to the discovery of attack paths is often used as an early security evaluation process, i.e. before the final system is available. Moreover, concerning real fault injection campaigns, there is also a need to determine potential attack paths in order to reduce the time needed to prepare an attack. While there exist several approaches to help designers and evaluators to find attack paths, they are often limited by the combinatorial explosion that arises when considering either large applications, or multiple faults with variable widths or different effects. Existing approaches typically make use of fault simulation [22], symbolic execution [18] or model checking [4]. As a consequence, we believe there is a need for a new kind of approaches able to scale with the multiplicity and width of faults as well as the size of the target application.

In this paper, we go in this direction by proposing an approach only based on static analysis to determine the possible attack paths when considering multiple faults with various widths. Our framework named SAMVA implements this analysis, it quantifies the vulnerability of a binary code, for example, on the basis of the minimum number of faults necessary to perform an exploit or the characteristics of the required faults.

Our approach works at the binary-level. We currently supports Arm binaries and instruction-skip like faults. In addition to the binary, SAMVA takes as input the attacker's capacity as well as their goal. The goal is expressed with a list of code addresses – mandatory steps to reach their objective, that must be executed – and a set of code addresses that must never be executed – corresponding to attack detection. Attacker capacity describes the possible number of faults as well as their possible widths. The static analysis is based on a path search heuristic in a graph representing the program and the effect of potential faults. The found candidate paths are analyzed in order to determine when and which faults to inject in order to make the attack path feasible. The analysis outputs the set of paths that meet user-specified fault injection constraints. We evaluate SAMVA on eight variations of PIN verification from the FISSC suite [9] while considering different attacker capacities. We verify the validity of the attacks paths found by SAMVA with a fault simulator based on the `gem5` [2]. We show that SAMVA is able to find in all implementations, even the most hardened, when and which faults to inject in order to reach an objective. Furthermore, we show that the required time to find attack paths is kept low even when considering hardened applications and a large set of potential effects of fault injections.

The threat model is introduced in Sec. 2. Sec. 3 depicts the core of our analysis for the search of attack paths. Our experimental setup and results are discussed in Sec. 4. We review related work in Sec. 5. Sec. 6 concludes.

## 2 Threat model

Since the seminal paper of Boneh et al. in 1997 [3], a lot of research has been conducted around fault injection. While some research works demonstrate the feasibility to retrieve sensitive data or to take over a device, some others aim at characterizing fault injection effects in order to better harden target systems. Fault effects can be modeled at different levels (logical level, RTL, assembly code, source code) using a bottom-up approach. A lot of research works have focused on the modelling of fault injection effects at ISA-level. Fault injections can lead to several effects at this level such as an instruction replacement or the frequent special case of an instruction skip [1,16,25]. While these papers report single fault effects, recent works show that one fault injection or complex fault injection means can lead to the corruption of several consecutive instructions. Electromagnetic pulses can lead to the replay or the skip of several consecutive instructions, from two up to a dozen [20,5,19,15]. Laser-based fault injection techniques can also lead to the skip of few chosen instructions [8] or of a variable number of consecutive instructions, from 1 to 300 depending on the laser pulse duration [11]. Multiple instruction skips, from a few chosen ones up to almost one hundred, can also be achieved using cheaper injection means such as clock glitching [6]. Instruction skip is a fault model that encompasses many fault effects, such as instruction replacement with another one that does not alter the execution, the replay of idempotent instructions, the replacement of the destination register of an instruction with a dead register, etc. It is powerful as it allows to easily corrupt the control flow of the execution. Moreover, injecting multiple instruction-skip faults allows an adversary to combine their effects to realize even more powerful attacks: Péneau et al. [17] show that if precise and numerous instruction skips can be injected, a binary program can be attacked in many ways. They show that NOP-oriented programming is Turing-complete. In this paper, we consider an attacker able to inject multiple and precise faults that finally result in skipping the execution of one or several consecutive instructions. The distance between two fault injections, the minimal and maximal number of instructions that are skipped with one fault injection all depend on the injection mean. We then consider them as input of the proposed analysis.

## 3 Method

We first provide an overview of the approach implemented in SAMVA. Then, we detail the modeling of faults effects on the binary and finally the proposed static analysis to determine the location of faults to be injected at run-time.

### 3.1 Overview

Fig. 1 gives an overview of the whole analysis dedicated to the search of attack paths. The analysis takes as inputs the binary, the objective of the attacker and the attacker's capacity. The output of SAMVA is a list of up to `N` attack paths,

`N` being defined by the user. An attack path contains the position of the required faults with their corresponding width (thereafter denoted *fw*) that need to be dynamically injected at run-time to achieve the attacker objective.

The objective of the attacker, denoted as *exploit specifications* in Fig. 1, are composed of (1) an ordered list of code locations that must all be reached during the execution in the specified order. This list composed of start addresses of straight-line code is referred to as the *targeted basic blocks*; (2) a set of code locations that must not be executed, for example it can correspond to code related to fault attack detection. This set is referred to as *forbidden basic blocks*.

The capacity of the attacker is expressed using three fault parameters (cf. Fig. 1): (1) `fw_min` indicates the minimal number of instructions skipped by one fault injection; (2) `fw_max` gives the maximal number of instructions skipped; (3) `f_min_dist` expresses the minimal number of instructions executed between two fault injections as imposed by the injection means. As an exemple, the setup of Dutertre et al. [11] (cf. Sec. 2) would be reflected by setting `fw_min=1` and `fw_max=300`. The `f_min_dist` would be set according to the frequency of the targeted processor and the reloading time of the fault injection setup.

First, the analysis automatically generates the control-flow graph (CFG) of the binary. A CFG is composed of basic blocks (BB) defined as a maximal length sequence of straight-line (i.e. branch-free) code. Basic blocks are linked with oriented edges to represent all possible execution paths of the program. The CFG is extended and annotated to reflect the effects of possible fault injections, noted hereafter ECFG. Then, potential attack paths are computed using the ECFG as well as the attacker objective. Finally, the analysis infers a set of attack paths that meet the attacker capacity. Each output attack path takes the form of a list of BB with the faults to inject (location, width) in the instruction trace generated by the execution of all the instructions of the BB list.

### 3.2 Fault effects modeling

Our approach implemented in SAMVA starts with the CFG of the binary program which characterizes all the possible execution paths in the absence of attack. We call it the nominal CFG in the remainder. It can be obtained by static analysis or a combination of static and symbolic analysis. In SAMVA, we use the angr framework [21] to build it. The ability to skip the execution of chosen instructions allows an attacker to alter the control-flow of a program in a way to force an existing execution path, or to create a new one. We model such potential effects of instruction skips by generating an ECFG from the nominal CFG. This step is independent of the attacker as it models all potential fault effects without considering the attacker capacity. In the following, we detail the two transformations performed on the nominal CFG to generate the ECFG that is later used by our attack paths finding heuristic.

**Hijacked control-flow modeling.** Being able to skip the execution of branch instructions enables an attacker to force the execution of the instructions which are located in memory right after these branch instructions.

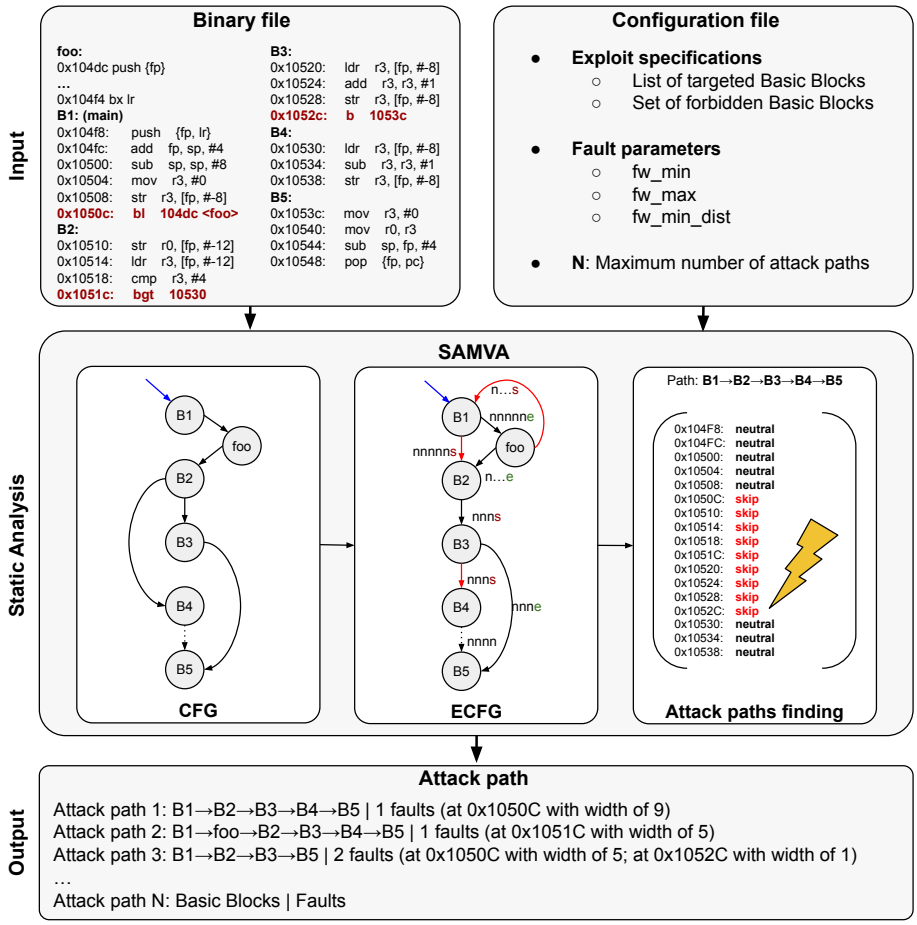



Fig. 1: Platform overview. In the code example, the targeted BB are [B1, B2, B5] and the set of forbidden BB is empty. Some found attack paths are given to illustrate the output format.

For the case of unconditional jump, we can choose to execute or to skip it. This allows an attacker to continue the execution with the instruction that comes after the jump instruction, according to the memory layout. The previously impossible control-flow is illustrated in Fig. 1 by the insertion of a new edge between `B1` and `B2` in the ECFG.

Concerning the skipping of a conditional branch, it forces the execution of the instruction that comes after the branch instruction, which corresponds to the case where the condition does not hold. As SAMVA currently does not rely on any dataflow analysis, and as each branch outcome must be statically known to compute feasible paths, the conditional jumps must always be skipped. As a consequence, the edges corresponding to the taken branches are removed of the ECFG. In the ECFG example in Fig. 1, the edge from `B2` to `B4` has been removed. Nonetheless, it is still possible, using several instruction skips, to execute the

target BB of the conditional branch if it is placed further in the memory layout. In the ECFG in Fig. 1, an attacker would have to skip the branch at the end of both `B2` and `B3` to reach `B4` from `B2`.

A limitation of our approach only based on control flow analysis and instruction skipping is that we cannot manage backward conditional branches, i.e. forcing the execution of the target BB of a conditional branch when this BB is at a lower address in the memory layout. This requires a data-flow analysis, to determine if it is feasible using only instruction-skip faults to force the condition to hold, or if this requires a fault outside our fault model (e.g. branch condition inversion). We keep as future work the study of data-flow analysis in presence of instruction-skip faults to force a backward conditional jump.

**Edges annotations.** This second step annotates the ECFG's edges to reflect if the corresponding control flow results from a fault injection on the branch instruction of the source BB of the edge or not. To define the edge annotation, we consider the following instruction types:

- *execute* (`e`) is the type of the instructions that must be executed;
- *skip* (`s`) is the type of the instructions that must be skipped;
- *neutral* (`n`) is the type of instructions that can either be skipped or executed without affecting the control flow at the end of their basic block.

Every instruction of a BB is typed. The neutral type leaves room for positioning the fault injection according to the attacker capacity. Based on this instruction type, an edge annotation can be derived by typing, in order, each instruction of its source BB. Branch instructions are always typed as either *skip* or *execute* to reflect the condition under which the edge must be followed during the execution. By default, all other instructions of a basic block are typed as *neutral*. These edge annotations are also illustrated in Fig. 1.

While this is enough for our attack paths finding heuristic, we refine the typing strategy to avoid source of crashes when performing an attack. In fact, inconsistent stack pointer updates during the execution may lead to a crash of the attacked program. Moreover, inconsistent return address can make the execution deviate from the expected execution path. As a consequence, we add the two following typing rules:

- *`R1:` Execution of stack pointer updates.* Instructions writing into the stack pointer register (`SP` register), such as `push` and `pop` instructions, are always typed as *execute*. This guarantees that the memory allocated for the stack is subsequently deallocated. For Arm architectures, the return from a callee function to its caller uses a unique instruction, a `pop pc` instruction or equivalent, to retrieve the return address on the stack, to update the stack pointer and finally to return to the caller function. As the execution of `pop` instructions is forced by this typing rule, this ensures that a function call is either skipped or the return to the caller will be correctly executed.
- *`R2:` Execution of function returns using the link register.* In case of leaf function, the link register `lr`, set by the call instruction `bl`, may be directly used for returning to the caller, i.e. using a `bx lr` instruction or equivalent. This

happens when the link register `lr` is not saved on the stack due to low register pressure. This additional rule types as *execute* all the `bx lr` instructions. As a consequence, a leaf function that does not save the link register on the stack is either skipped or the return to the caller will be correctly executed.

### 3.3 Attack paths finding

In this section, we present our heuristic for finding attack paths. We first explore the ECFG to generate a set of paths that are compliant with the attacker objective. These paths must contain the target basic blocks in the order specified by the user and must not contain any forbidden basic blocks. Edge annotations present on each path are then used to determine the position and the width of fault injections to perform while conforming with the instruction types. The position and the width of fault injections must be valid according to the attacker capacity given as input to the analysis.

**Candidate paths generation.** A set of candidate paths is generated by exploring the ECFG. Such paths must reach, in the correct order, the BB specified in the attacker objective and avoid the forbidden BB. Additionally, an ideal candidate path should allow an easy fault injection positioning by spacing out the *execute* and *skip* instructions, and reduce the number of fault injections by favoring *neutral* and *execute* instructions. Therefore, we associate to each edge of the ECFG a cost depending on its annotation.

- Cost=1: if instructions are all typed as *neutral*;
- Cost=2: if instructions are only typed as *neutral* or *execute*;
- Cost=3: if instructions are only typed as *neutral* or *skip*;
- Cost=4: if there are some instructions typed as *skip* and some other ones typed as *execute*.

This weighing policy hints the path search at finding more feasible attack paths. Thus, for each pair of successive basic blocks in the list of targeted basic blocks, a temporary set of paths is retrieved using the shortest paths algorithm [24] according to edge weights. The complexity to find the $K$ first shortest paths in a CFG containing $N_{BB}$ basic blocks is then $\mathcal{O}(K{N_{BB}}^3)$.

The final set of complete candidate paths $P_{candidate\_paths}$ passing through all the basic blocks specified in the attacker objective is then generated by making the Cartesian product of the temporary sets. We iteratively combine the sets corresponding to consecutive basic blocks and retain the K paths with the least costs. This final set is composed of candidate paths that do not ensure the possibility of the fault injection positioning according to the attacker capacity. The next step aims at finding a valid set of fault injections to perform to make a candidate path an attack path.

**Fault injection positioning.** The determination of fault injections to perform in order to make feasible a given candidate path is based on the instructions

types retrieved on the edges annotations and the attacker capacity. For a given candidate path, we build a so-called "execution trace" which is a list of pairs $\langle$*instruction address*, *instruction type*$\rangle$. The fault injection positioning aims at finding the position and width of faults to inject such that instructions typed as *skip* are covered by a fault and instructions typed as *execute* are outside of any fault. Instructions typed as *neutral* can be covered by a fault or not.

The width of any injected fault must be included in [`fw_min`, `fw_max`]. As a consequence, there are potentially a lot of possibilities for the fault injection position and width as shown in Fig. 2. Nevertheless, the distance between two consecutive faults must be at least equal to the minimal distance `f_min_dist`. Computing the whole set of possible fault positions and widths is not realistic, as a consequence we use a two-step approach to determine a fixed-size set of solutions: i) we first use simple rules to quickly determine when there is no valid solution for the fault positioning based on the distance between instructions; ii) then, the set of remaining fault configurations (i.e. position and width) is explored using a backtracking algorithm in order to find a valid configuration that make feasible a candidate path.

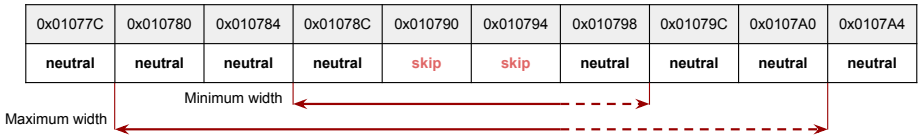



Fig. 2: Example of fault positioning on trace with `fw_min` $= 3$, `fw_max` $= 5$

*Unsolvability verification.* We use the following straightforward rules to quickly detect the unsolvability of the fault positioning problem on an execution trace:

- If there is at least one instruction typed as *skip* between two instructions $i_0$ and $i_1$ typed as *execute*, then the distance between $i_0$ and $i_1$ must be greater than or equal to `fw_min` the minimal width of a fault. Otherwise, any fault covering the instruction typed as *skip* would at least impact $i_0$ or $i_1$, and so the fault positioning problem is unsolvable;
- If there is at least one instruction typed as *execute* between two instructions $i_0$ and $i_1$ typed as *skip*, then the distance between $i_0$ and $i_1$ must be greater than or equal to `f_min_dist` the minimal distance between two faults. Otherwise, the fault positioning problem is unsolvable.

*Backtracking algorithm.* The algorithm attempts to place faults in order to cover all the instructions typed as *skip* in a trace or to prove the invalidity of the attack path. Thus, we try to build a solution, consisting in a list of faults each having a position and a width. Additionally, these faults must respect the fault width constraints (`fw_min`, `fw_max`) and respect the distance between each other (i.e. meet the minimal distance requirement `fw_min_dist`).

A solution is built incrementally with a backtracking approach using recursion. Alg. 1 gives an overview of our implementation. First, the position of the

**Algorithm 1** Fault positioning algorithm using backtracking

```
function FAULT_POSITIONING(trace, f_candidates, f_params)
    next_pos ← find_next_skip(trace, f_candidates)
    if next_pos = ∅ then
        return True
    for fw ∈ range(f_params.fw_max, f_params.fw_min, -1) do
        for pos ∈ range(next_pos, next_pos - fw, -1) do
            fault ← <pos, fw>
            if is_valid(fault, trace, f_candidates, f_params) then
                f_candidates.push(fault)
                if fault_positioning(trace, f_candidates, fault_params) then
                    return True
                else
                    candidate_faults.pop()
    return False
```

next instruction typed as *skip* that is not yet covered by a fault is retrieved in the execution trace. Then, we vary the width and position of the fault in order to find a valid fault configuration.

To determine if a candidate fault configuration is valid, the following properties are verified:

- the fault position must fit in the trace, i.e. taking care of the trace bounds;
- the fault must not cover any instruction typed as *execute*;
- the fault must not overlap with the previous fault (if any) and their distance must be greater than the minimal distance between two faults.

The recursive calls stop when a final solution meeting all the constraints and covering all the instructions typed as *skip* is obtained. This happens when the function `find_next_skip` no longer finds any uncovered instruction typed as *skip*. During this process if we discover that the current solution will not be valid, we backtrack, i.e. we go back to the previous step by removing the last validated fault and try another fault configuration instead. Backtracking algorithms use the depth-first search method. In order to minimize the number of faults necessary to perform the attack, we first explore, as visible in the loops order, the possible positions starting from the one of the instruction to cover and then vary its width, starting with the widest one.

For the sake of performance, we do some optimizations to reduce the space of possible fault configurations. First, when validating a position of a fault, we also check if there is any instruction typed as *skip* further in the trace at a distance of less than `fw_min_dist` that would be covered by a conflicting fault. Consequently, even if the configuration of the fault is valid with the already chosen faults, it is rejected to avoid useless recursive calls.

Additionally, we decompose our execution trace into several sub-traces that are then handled independently. We apply a cut in the execution trace when (1) two instructions typed as *skip* are separated only by instructions typed as

*neutral*, and (2) the distance between these two instructions is larger than twice the maximal fault width plus the minimal distance between two faults.

*Fault trimming.* Our fault positioning algorithm tries to make the faults as large as possible in order to reduce their number. It can therefore find valid solutions that nonetheless cover unnecessarily instructions typed as *neutral*. For this reason, we apply a last pass that shortens the width of the faults when possible. It shifts the beginning and the end of a fault towards the first and the last instruction typed as *skip* while meeting the constraint of the minimal fault width (`fw_min`). We thus obtain smaller faults, potentially easier to achieve, and which reduce the risk of skipping critical instructions.

## 4 Experimentation

In this section, we evaluate the effectiveness of SAMVA. We first present the experimental setup comprising the targeted applications, the considered attacker capacity and our evaluation methodology. Then, we discuss the results.

### 4.1 Experimental setup

**Benchmarks.** We evaluate our analysis on all PIN verification programs from the FISCC project [9]. This software collection contains eight implementations of `VerifyPIN`, one naive implementation, as illustrated in Lst. 1 and seven other implementations containing different set of countermeasures. The PIN code verification programs compare a user-provided PIN and the card PIN using the function `byteArrayCompare`. The variable `g_authenticated` is set according to its result. The number of tries is controlled by the variable `g_ptc`, initially set to 3 and decremented after each failed authentication attempt. Authentication is no longer permitted if `g_ptc` reaches zero, in order to avoid brute-forcing the PIN code. For protected implementations, i.e. version higher than V0, a fault handler is called when an attack is detected by a countermeasure. The fault handler sets to true a variable added to any protected version and named `g_countermeasure`. In the end, the evaluator is able to know afterward if the attack has been detected by the countermeasures. The implemented countermeasures are described below. Table 1 reports the countermeasures implemented in each `VerifyPIN` version as well as the number of instructions, BB and edges in the ECFG considered in the analysis at the binary level.

- Hardened Booleans (**HB**): Booleans are encoded with two constants, instead of 0 and 1, which are less sensitive to fault injection;
- Step counter (**SC**): some variables called step counters are added to the code in order to protect against attacks disrupting the control flow integrity. The number of loop iterations is checked at the loop exit in versions `V2` to `V5`. All the statements and control flow constructs are protected using such variables in version `V7`;

- Inlined calls (**IC**): function calls are inlined in order to prevent the skip of the call. This also reduces the attack surface as there is no more instructions to pass parameters to the calls;
- Backup copy (**BC**): the number of remaining attempts is duplicated to prevent single fault attacks from targeting the attempt counter;
- Double test (**DT**): the call to the function verifying the PIN codes and all the tests are duplicated to prevent a single fault from bypassing them.

The objective of an attacker is to obtain an authentication without knowing the user PIN and without triggering any countermeasures. As a consequence, our analysis searches for the faults enabling to hijack the control-flow of the program in order to execute the authentication code (lines 4 and 5) without executing any attack detection. For the experiments, we manually retrieve the targeted and forbidden basic blocks for each implementation of `VerifyPIN`. The attacks start at the beginning of the verification function and then we define the targeted BBs as a list containing: the BB setting `g_authenticated` at true, possibly the BB setting `g_ptc` at 3 if this code is not included in the previous BB, and finally the BB in then `main` function that comes right after the call to `VerifyPIN`. The set of forbidden BB only includes the BB calling the detection function that sets the `g_countermeasure` variable.

The eight versions of `VerifyPIN` are compiled for Arm Thumb instruction set architecture (ARMv7-M). The cross-compiler used is GNU GCC gnueabi version 8.5.0. We deactivate all compiler optimizations (`-O0`) to avoid the alteration of the software countermeasures, as well as the use of predicated instructions that are not yet supported in SAMVA.

Listing 1: Source code of `VerifyPIN` without countermeasures (V0)

```
1 g_authenticated = 0;
2 if(g_ptc > 0) {
3     if(byteArrayCompare(...)) {
4         g_ptc = 3;
5         g_authenticated = 1;
6     } else {
7         g_ptc--;
8     }
9 }
```

Table 1: `VerifyPIN` suite description with the included countermeasures, their number of instructions, BB and ECFG edges (+ edges added to original CFG) at binary level

| | **HB** | **SC** | **IC** | **DT** | **BC** | #Instr | #BB | #Edges |
|---|---|---|---|---|---|---|---|---|
| V0 | | | | | | 142 | 24 | 46 (+12) |
| V1 | ✓ | | | | | 162 | 30 | 57 (+15) |
| V2 | ✓ | ✓ | | | | 172 | 32 | 58 (+15) |
| V3 | ✓ | ✓ | ✓ | | | 158 | 30 | 54 (+13) |
| V4 | ✓ | ✓ | ✓ | ✓ | | 221 | 41 | 79 (+20) |
| V5 | ✓ | ✓ | | | ✓ | 241 | 47 | 87 (+22) |
| V6 | ✓ | | ✓ | | ✓ | 177 | 36 | 68 (+17) |
| V7 | ✓ | ✓ | ✓ | | ✓ | 306 | 66 | 140 (+38) |

**Fault injection parameters.** For each implementation of `VerifyPIN`, we consider various fault injection parameters corresponding to various attacker capacities. We vary the width of the possible faults (using the fault parameters `fw_min` and `fw_max`) as well as the minimal distance between two consecutive faults (using the parameter named `fw_min_dist`). Our objective is to observe the sensitivity of the included countermeasures to the fault injection parameters required to perform an attack.

Let $W$ be the set of possible fault width values measured in number of instructions. It is defined as: $W := \{1\} \bigcup \{2n : n \in \mathbb{N} \mid n \leq 32\}$. Thus, the minimum width varies over 1 and all even numbers between 2 to 64; the maximum width varies over the minimal width and all even numbers between 2 to 64 as well, such that: $\{(\texttt{fw\_min}, \texttt{fw\_max}) \in W \times W \mid \texttt{fw\_min} \leq \texttt{fw\_max}\}$. Finally, the minimal distance, in number of instructions, between two fault injections varies over all the power of 2, such that: $\texttt{fw\_min\_dist} \in \{2^n : n \in \mathbb{N} \mid n \leq 5\}$.

Moreover, we run SAMVA on all the versions of `VerifyPIN` considering instruction typing strategies (cf. Sec. 3.2). We pick three different strategies as follows: a first default one, denoted `default`, without any additional typing rule; a second one featuring the `R1` rule that forces the execution of stack pointer updates; a third one, denoted `R1 + R2`, that applies both `R1` and `R2` rules.

In summary, we test SAMVA on a total of 3366 distinct fault parameters, on each of the eight binary files, for each of the three instruction typing strategies, with fault trimming enabled and disabled.

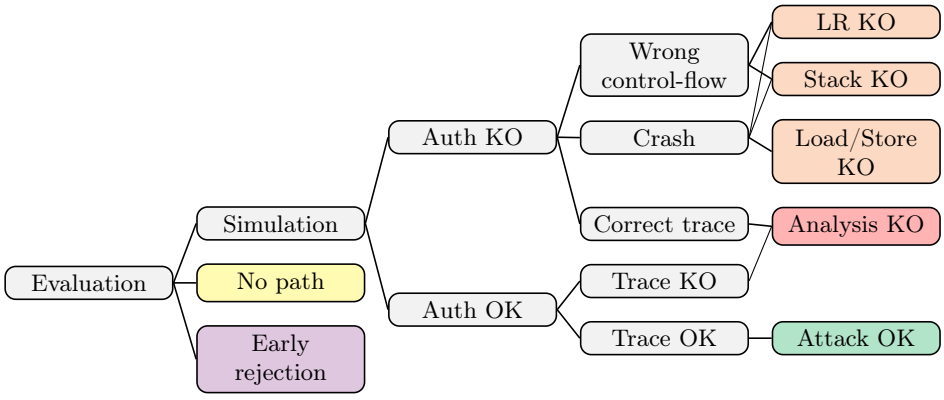



Fig. 3: Decision tree for attack results classification

**Evaluation methodology.** We iterate through the possible fault parameters and binaries as described previously. For a given couple of binary and fault parameters, we strive to generate a set of `N` distinct attack paths. In our experiments, `N` equals 30, meaning that we expect to obtain up to 30 attacks paths, depending on the possibilities offered by the instructions used and the binary layout. The evaluation methodology followed to assess the results is depicted on Fig. 3. We classify the results as explained below.

The set of attack paths found by the analysis can be empty if our analysis does not find any candidate attack path for certain fault parameters (class "No path" □). Additionally, the fault parameters may not fit the binary if `fw_min`, the minimal fault width, is greater than the number of instructions distancing the starting point and the first targeted instruction of the attack (class "Early rejection" □), since the execution of targeted BB is mandatory.

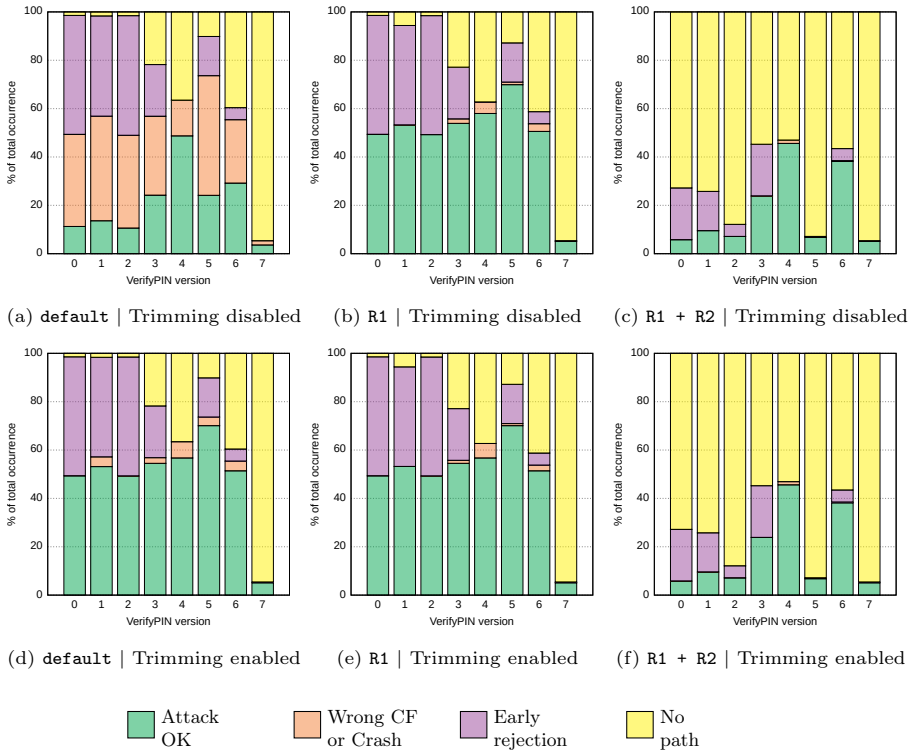
Otherwise, if the resulting set is not empty, the attack paths are validated by simulation of the instruction-skip fault model. The simulator used in our ex-

periment is a modified version of `gem5` [2] which is able to skip the execution of chosen instructions in a specified order of occurrence. It takes as input the binary program under analysis and the faults that must be injected for achieving an attack path. Once the simulation terminates, without reporting a crash, we first check the output containing the `VerifyPIN` variables. If the `g_authenticated` is set to true, `g_ptc` equals 3 and `g_countermeasure` stays at false, the authentication has been granted (node "Auth OK"). In this case, the execution trace resulting from the simulation and the one intended by the analysis are compared in order to make sure they match (class "Attack OK" ). We stop iterating the set of attack paths after the first successful simulated attack. In the case of a simulation crash or if we did not get authenticated at the end of the simulation (node "Auth KO"), the simulated and expected execution traces are also analyzed. When they do not match, we determine the reason of the crash or of the divergence of control-flow respectively (classes "LR KO", "Stack KO" or "Load/-Store KO"). The result is orange-colored (node "Crash" and "Wrong control-flow" ). Finally, we also measure the failure of our analysis for two particular cases (class "Analysis KO" ). The first one is when the authentication fails despite the matching of the execution traces and the expected authentication by the analysis. The second case is when we get authenticated but the traces do not match. These cases are sanity checks only, which should not happen. We did not encounter them in our experiments.

### 4.2 Experimental results

**Attack path evaluation results.** Experiments aim to measure the effectiveness of SAMVA in finding attack paths in the different benchmarks according to the different typing instruction strategies. We consider three strategies with and without fault trimming. Fig. 4 shows the classification of the evaluation outcomes. The first row represents the results for the three strategies without the fault trimming and the second one with trimming.

First of all, we are able to find many attack paths in all cases. The main difference between the default strategy and the other ones preserving the execution of stack pointer updates is the higher number of crashes during the simulation using the default strategy. However, fault trimming seems to sensibly mitigate the number of crashes by reducing the number of instructions that must be skipped, meaning, it reduces the risk of skipping an instruction that is necessary for the execution of the program such as memory allocation for the stack. Nonetheless, fault trimming has only an impact when no instruction typing rule is enabled. Otherwise, the strategies guarantee the execution of some instructions reducing the number of crashes. Finally, the `R1 + R2` strategy finds fewer attack paths. This reduced number of attack paths can be explained by the constraints induced by the instruction typing rules, resulting in a more difficult fault positioning. However, found attack paths lead to fewer crashes. This means that the few attack paths found are more prone to be effective. Remaining crashes are solely caused by invalid memory access due to instruction skips. To load or store a value, the address location is usually stored in a register that is defined before




(a) `default` | Trimming disabled (b) `R1` | Trimming disabled (c) `R1 + R2` | Trimming disabled

(d) `default` | Trimming enabled (e) `R1` | Trimming enabled (f) `R1 + R2` | Trimming enabled

Fig. 4: Results classification of attack path searches

by one or multiple instructions. If one of these instructions is skipped, then an illegal access can cause a crash. Future work will study data-flow analysis to handle these cases.

**Fault injection parameters study.** An alternative manner of representing the results for the eight binaries is depicted on Fig. 5, which shows the classification of the analysis outcomes according to the considered three fault parameters (`fw_min`, `fw_max`, `fw_min_dist`). As expected, the general pattern that we can observe is that the smaller `fw_min` and `fw_min_dist` are, the larger the `fw_max` is, the more possibilities to find attack paths which result in successful attacks. We can see that versions `V1`, `V2` and `V5` have similar results to `V0`, meaning that the implemented countermeasures have only limited effect against multiple skips. We can also see that in `V4` and `V6`, the distance between two faults is the main factor for the realization of the attack. This can be explained by the necessity to make several faults if `fw_max` does not allow to make a sufficiently large fault. Finally for `V7`, we only find few configurations that lead to successful attacks. The fine-grained control-flow integrity countermeasure included in this version forces to skip several small sets of instructions and thus require a higher precision to inject the faults.

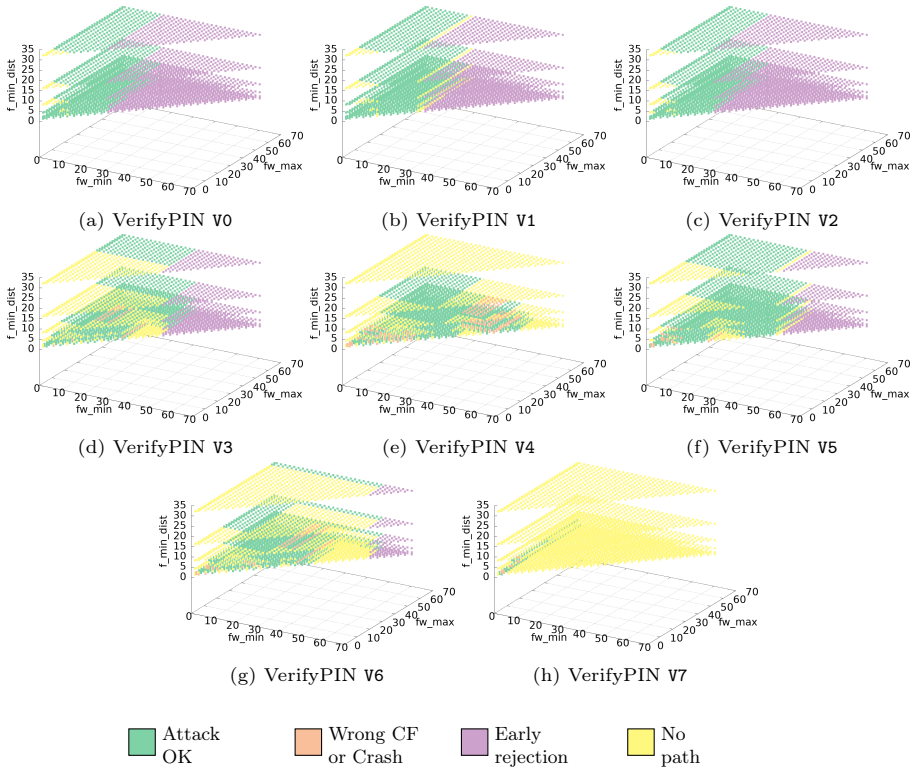
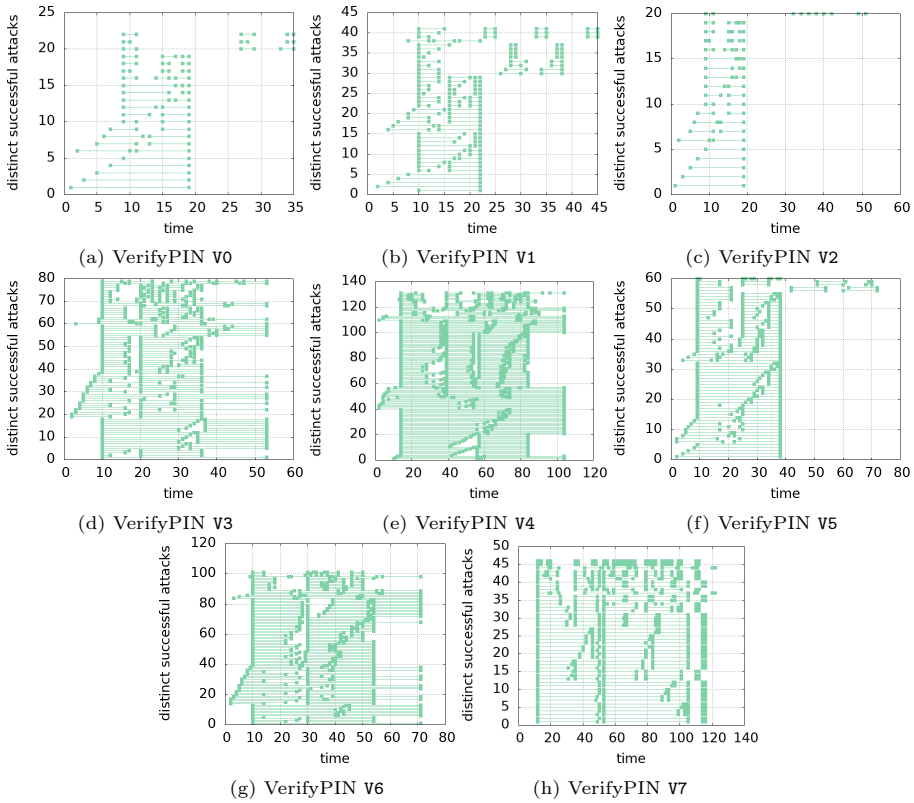



Fig. 5: Outcomes of every tested configuration, per VerifyPIN version, using the `R1` strategy and fault trimming enabled

**Characteristics of successful fault configurations.** We now study the characteristic of the fault configurations that lead to successful attacks. Since the strategy featuring the additional typing rule `R1` along with the fault trimming gives the highest number of successful attack paths, we base our study on its evaluation results. Fig. 7 presents the number of faults required for each successful attacks. Our results show that versions from `V0` to `V3`, `V5` and `V6` can be attacked with a single fault. Version `V4` can be attacked with at least two faults and `V7` requires at least three faults. The instruction typing obtained for a given attack path is responsible for the minimum number of faults. For instance, if two instructions are typed as *skip* with an instruction typed as *execute* in between, then two faults are necessary. Depending on the code layout and instructions induced by the countermeasures, the attack path may contain such constraints, resulting in a higher number of faults required for the `V4` and `V7`.

To better understand the effects of instruction typing on the fault positioning, Fig. 6 represents the characteristic of the faults on an attack path. These different attack paths can report identical control-flow, although we can see some patterns. Taking the `V0` as an example, we can notice that according to the fault injection parameters, SAMVA can choose to make one long fault to cover all the




(a) VerifyPIN `V0` (b) VerifyPIN `V1` (c) VerifyPIN `V2`

(d) VerifyPIN `V3` (e) VerifyPIN `V4` (f) VerifyPIN `V5`

(g) VerifyPIN `V6` (h) VerifyPIN `V7`

Fig. 6: Unique attacks found for each version of VerifyPIN. Each attack is represented horizontally. The x-axis represents time (more precisely consecutively executed instructions). Each segment denotes a fault whose width is the length. For example, `V1` can be attacked with a single fault of width 12 (bottom segment ranging from x=10 to x=22); but also with four narrower faults shown at y=41: a fault ranging from x=10 to x=15 followed by tree faults of width 3 at times x=23, x=33, and x=43. Attacks are sorted vertically by their number of faults: fewer faults at the bottom, more towards the top.

instructions typed *skip* or to make several smaller faults to cover them individually. For some attacks, the BB restoring the variable `g_ptc` to its initial value and the BB turning `g_authenticated` to true may be different. As a result, we get mandatory checkpoints in the control-flow, which graphically manifests as a column in the figures, because no fault is allowed to cover this section of the attack path. Finally, we consider only fully predictable paths in our analysis. Since we do not use data-flow analysis, we hijack conditional jumps that do not necessarily require a fault. For instance, at the beginning of the `VerifyPIN` function the value of `g_ptc` is checked and must be greater than zero, as depicted in Lst. 1 (line 2). As we consider only one try, during the attack this condition always holds. In consequence, the branch instructions related to this check add unnecessary constraints by adding an instruction typed as *skip* and result in more faults than really required.

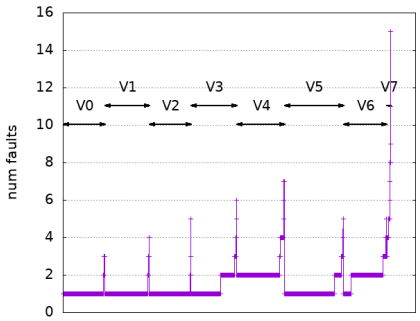



Fig. 7: Number of faults needed for each successful attack found for each version of VerifyPIN, using the strategy `R1` with fault trimming enabled

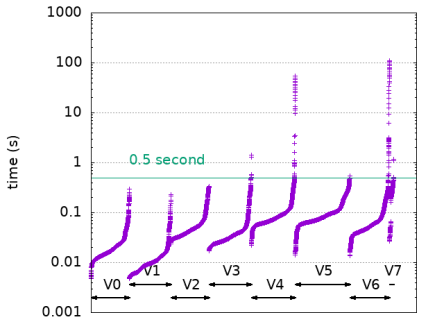



Fig. 8: Time needed to generate the paths, for each considered fault parameter and per VerifyPIN, using the strategy `R1` with fault trimming enabled

**Execution time.** We measure the execution time of our framework in order to assess its performance. Fig. 8 represents the time that our analysis took to find up to 30 distinct attack paths for each version of `VerifyPIN` and for each parameter configuration. These results do not include the simulation time. The strategy used for these measurements is the typing rule `R1` with fault trimming enabled. We ran our benchmarks on a Xeon Gold 5218 CPU at 2.3 GHz featuring 32 physical cores, on which independent instances of SAMVA are launched. Each instance of SAMVA is sequential, hence the times reported are independent of the parallelism of the server. We obtain relatively short analysis times, most of the results are under the threshold of half a second. For the `V4` and `V7` versions, we can notice that the analysis times can rise significantly, up to 109 seconds and this can be explained by the usage of the fault positioning algorithm which uses backtracking. Indeed, the major part of the analysis time is actually spent in this algorithm and according to the typing of the instructions, we may need to backtrack a lot to prove the non-feasibility of an attack path.

## 5 Related work

In order to help both security evaluators and countermeasure designers, different vulnerability and attack path search tools have been proposed.

Potet et al. [18] propose Lazart, a tool based on the modification of the CFG at LLVM-IR level to establish using symbolic execution the absence of attacks only based on multiple branch inversions. While convenient to early analyze the effectiveness of software countermeasures, this solution does not consider the binary layout and so requires a companion analysis at binary level. While the authors do not report the time required by the analysis, this approach is intrinsically limited by the symbolic execution engine that faces path or state explosion in case of complex applications with symbolic inputs which impact memory accesses or control flow.

Bréjon et al. [4] propose the framework `RobustB` that uses formal verification through SMT solving to find vulnerabilities in binary code. The considered faults are either a single instruction skip or a single register corruption. The reported verification times on the same benchmarks range from few minutes up to few hours without details. We can however say that our approach is more efficient as it requires less than two minutes in the worst case, and our attacker model encompasses the single instruction-skip fault model.

Given-Wilson et al. [12] also propose an automated approach based on formal verification to find vulnerabilities against fault attacks at binary code level. The approach only considers permanent faults that are reflected in code mutants that are then given to a model checker. This approach must then produce as many code mutants as the number of fault configurations to explore. This would not scale to multiple faults with various widths.

Werner et al. [22] extend the CELTIC simulation-based framework in order to search for attack paths considering up to two faults. Considered fault models are inferred from real experiments, as previously proposed by Dureuil et al. [10], and the whole approach enables to select fault injection parameters. As other simulation-based approach [14], it is however limited in the number of faults that can be injected. While simulation is better suited than formal approaches for analyzing large applications, the fault configurations space grows exponentially when considering multiple faults with variable width. The convergence towards successful fault configurations is dependent on the exploration strategy of the fault configurations space. To the best of our knowledge, there is currently no simulation-based approach able to consider a large number of such faults.

In summary, we believe that, even if only instruction-skip faults are supported yet, SAMVA, which is only based on static analysis, is the first tool able to search for multiple faults with variable width that leads to successful attacks.

## 6 Conclusion

In this paper, we propose SAMVA, a framework for assessing vulnerabilities of a program binary against multiple instruction-skip attacks. SAMVA is based on purely static analysis. We evaluate our approach by determining the required faults to attack eight versions of PIN code verification programs hardened by various countermeasures against faults. In our experiments, we explore numerous fault injection capabilities and the results show the capacities of SAMVA to find successful attack paths, even for the most hardened implementations. We also report that our approach scales well, making it an effective way to explore a wide range of fault configurations in limited time.

Future work will consider the extension of our threat model by integrating instruction-replay for our fault positioning. Additionally, we plan to link the attacks found by analysis with fault injection means to conduct physical attacks in order to validate experimentally the found attacks. This will make the bridge between our fault analysis and their realizations.

## Acknowledgements

This study is partially funded by the ANR within the framework of the PIA EUR CyberSchool project (ANR-18-EURE-0004) and by Région Bretagne.

## References

1. Balasch, J., Gierlichs, B., Verbauwhede, I.: An in-depth and black-box characterization of the effects of clock glitches on 8-bit MCUs. In: FDTC. IEEE Computer Society (2011)
2. Binkert, N.L., Beckmann, B.M., Black, G., Reinhardt, S.K., Saidi, A.G., Basu, A., Hestness, J., Hower, D., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Altaf, M.S.B., Vaish, N., Hill, M.D., Wood, D.A.: The gem5 simulator. SIGARCH Comput. Archit. News **39**(2) (2011)
3. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults. In: EUROCRYPT. LNCS, vol. 1233. Springer (1997)
4. Bréjon, J.B., Heydemann, K., Encrenaz, E., Meunier, Q., Vu, S.T.: Fault attack vulnerability assessment of binary code. In: CS2. ACM (2019)
5. Bukasa, S.K., Lashermes, R., Lanet, J.L., Legay, A.: Let's shock our IoT's heart: ARMv7-M under (fault) attacks. In: ARES 2018. ACM (2018)
6. Claudepierre, L., Péneau, P.Y., Hardy, D., Rohou, E.: TRAITOR: A low-cost evaluation platform for multifault injection. In: ASSS. ACM (2021)
7. Colombier, B., Grandamme, P., Vernay, J., Chanavat, É., Bossuet, L., de Laulanié, L., Chassagne, B.: Multi-spot laser fault injection setup: New possibilities for fault injection attacks. In: CARDIS. LNCS, vol. 13173. Springer (2021)
8. Colombier, B., Menu, A., Dutertre, J.M., Moëllic, P.A., Rigaud, J.B., Danger, J.L.: Laser-induced single-bit faults in flash memory: Instructions corruption on a 32-bit microcontroller. In: IEEE HOST. IEEE (2019)
9. Dureuil, L., Petiot, G., Potet, M.L., Le, T.H., Crohen, A., de Choudens, P.: FISSC: A fault injection and simulation secure collection. In: Skavhaug, A., Guiochet, J., Bitsch, F. (eds.) Computer Safety, Reliability, and Security - 35th International Conference, SAFECOMP. LNCS, vol. 9922. Springer (2016)
10. Dureuil, L., Potet, M.L., de Choudens, P., Dumas, C., Clédière, J.: From code review to fault injection attacks: Filling the gap using fault model inference. In: CARDIS. LNCS, vol. 9514. Springer (2015)
11. Dutertre, J.M., Riom, T., Potin, O., Rigaud, J.B.: Experimental analysis of the laser-induced instruction skip fault model. In: NordSec. LNCS, vol. 11875. Springer (2019)
12. Given-Wilson, T., Heuser, A., Jafri, N., Legay, A.: An automated and scalable formal process for detecting fault injection vulnerabilities in binaries. Concurr. Comput. Pract. Exp. **31**(23) (2019)
13. Heydemann, K., Lalande, J.F., Berthomé, P.: Formally verified software countermeasures for control-flow integrity of smart card C code. Comput. Secur. **85** (2019)
14. Hoffmann, M., Schellenberg, F., Paar, C.: ARMORY: fully automated and exhaustive fault simulation on ARM-M binaries. IEEE Trans. Inf. Forensics Secur. **16** (2021)
15. Menu, A., Dutertre, J.M., Potin, O., Rigaud, J.B., Danger, J.L.: Experimental analysis of the electromagnetic instruction skip fault model. In: DTIS. IEEE (2020)

16. Moro, N., Dehbaoui, A., Heydemann, K., Robisson, B., Encrenaz, E.: Electromagnetic fault injection: Towards a fault model on a 32-bit microcontroller. In: FDTC. IEEE Computer Society (2013)
17. Péneau, P.Y., Claudepierre, L., Hardy, D., Rohou, E.: NOP-oriented programming: Should we care? In: SILM EuroS&P Workshops. IEEE (2020)
18. Potet, M.L., Mounier, L., Puys, M., Dureuil, L.: Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections. In: ICST. IEEE Computer Society (2014)
19. Proy, J., Heydemann, K., Berzati, A., Majéric, F., Cohen, A.: A first ISA-level characterization of EM pulse effects on superscalar microarchitectures: A secure software perspective. In: Proceedings of the 14th International Conference on Availability, Reliability and Security, ARES. ACM (2019)
20. Rivière, L., Najm, Z., Rauzy, P., Danger, J.L., Bringer, J., Sauvage, L.: High precision fault injections on the instruction cache of ARMv7-M architectures. In: HOST. IEEE Computer Society (2015)
21. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Krügel, C., Vigna, G.: SOK: (state of) the art of war: Offensive techniques in binary analysis. In: IEEE Symposium on Security and Privacy, SP. IEEE Computer Society (2016)
22. Werner, V., Maingault, L., Potet, M.L.: An end-to-end approach for multi-fault attack vulnerability assessment. In: FDTC 2020. IEEE (2020)
23. Witteman, M., Oostdijk, M.: Secure application programming in the presence of side channel attacks (2008), `https://www.riscure.com/publication/secure-application-programming-presence-side-channel-attacks/`
24. Yen, J.Y.: Finding the k shortest loopless paths in a network. Management Science **17**(11) (1971)
25. Yuce, B., Ghalaty, N.F., Santapuri, H., Deshpande, C., Patrick, C., Schaumont, P.: Software fault resistance is futile: Effective single-glitch attacks. In: FDTC. IEEE Computer Society (2016)
26. Yuce, B., Schaumont, P., Witteman, M.: Fault attacks on secure embedded software: Threats, design, and evaluation. J. Hardw. Syst. Secur. **2**(2) (2018)