# Adversarial Reachability for Program-level Security Analysis[⋆]

Soline Ducousso[1], Sébastien Bardin[1], and Marie-Laure Potet[2]

[1] Université Paris-Saclay, CEA, List, Saclay, France
soline.ducousso@cea.fr, sebastien.bardin@cea.fr
[2] Univ. Grenoble Alpes, VERIMAG, Grenoble, France
marie-laure.potet@univ-grenoble-alpes.fr

**Abstract.** Many program analysis tools and techniques have been developed to assess program vulnerability. Yet, they are based on the standard concept of reachability and represent an attacker able to craft smart *legitimate* input, while in practice attackers can be much more powerful, using for instance micro-architectural exploits or fault injection methods. We introduce *adversarial reachability*, a framework allowing to reason about such *advanced attackers* and check whether a system is vulnerable or immune to a particular attacker. As equipping the attacker with new capacities significantly increases the state space of the program under analysis, we present a new symbolic exploration algorithm, namely *adversarial symbolic execution*, injecting faults in a *forkless* manner to prevent path explosion, together with optimizations dedicated to reduce the number of injections to consider while keeping the same attacker power. Experiments on representative benchmarks from fault injection show that our method significantly reduces the number of adversarial paths to explore, allowing to scale up to 10 faults where prior work timeout for 3 faults. In addition, we analyze the well-tested WooKey bootloader, and demonstrate the ability of our analysis to find attacks and evaluate countermeasures in real-life security scenarios. We were especially able to find an attack not mentioned in a previous patch.

**Keywords:** Program analysis · Attacker model · Fault injection · Symbolic execution

## 1 Introduction

**Context.** Major works have delved into program analysis over the last decades, leveraging techniques such as symbolic execution [18, 24, 53], static analysis [43], abstract interpretation [30] or bounded model checking [29], to hunt for software vulnerabilities and bugs in programs, or to prove their absence [35, 60], leading to industrial adoption in some leading companies [6,18,43,60,66]. As bugs are an attack entry point, removing them is a first step towards better software security.

[⋆] Partially supported by grants ANR TAVA, PEPR Secureval and Carnot Flexsecurity.

**Problem.** Yet, stepping back from these successes, it appears that all these methods consider a rather weak threat model, where the attacker can only craft smart "inputs of death" through legitimate input sources of the program, exploiting corner cases in the code itself. Tools only looking for bugs and software vulnerabilities may deem a program secure while the bar remains quite low for an *advanced attacker*, able for example to take advantage of attack vectors such as (physical) hardware fault injections [58], micro-architectural attacks [61, 70], software-based hardware attacks [55, 69, 86] like Rowhammer [70], or any combination of vectors [63]. While previously limited to high-security devices and systems such as smart cards and cryptography modules [13,16], these fault-based attacks can now target a wider spectrum of systems, such as bootloaders [57], firmware update modules [19], security enclaves [69], etc. The reasoning behind automated software-implemented fault injection also applies to Man-At-The-End attacks [3] and is similar to the (manual) reasoning performed in control-flow integrity to evaluate countermeasures [1, 21].

**Goal & Challenges.** *Our goal is to devise a technique to automatically and efficiently reason about the impact of an advanced attacker onto program security properties*, where the standard reachability framework only supports an attacker crafting smart legitimate inputs. The first challenge is to provide a formal framework to study what an advanced attacker can do to attack a program. Interestingly, while such frameworks are routinely used in cryptographic protocol verification [7, 26], none has been studied for program-level analysis. The second challenge is to design an efficient algorithm to assess the vulnerability of a program to a given attacker model, while adding capabilities to the attacker naturally gives rise to a significant path explosion – especially in the case of multiple fault analysis.

The rare prior works in the field, mostly focused on encompassing physical fault injections for high-security devices, rely mostly on *mutant generation* [25,28,49,50,79] or *forking analysis* [15,20,63,76], yielding scalability issues. Moreover, most of them are limited to a few predefined fault models and do not propose any formalization of the underlying problem.

**Proposal.** We propose *adversarial reachability*, a formalism extending standard reachability to reason about a program execution in the presence of an advanced attacker, and we build a new algorithm based on symbolic techniques, named *adversarial symbolic execution*, to address the adversarial reachability problem from the bug finding point of view (bounded verification). Our algorithm prevents path explosion thanks to a new *forkless* encoding of faults. We show it is correct and k-complete with respect to adversarial reachability. To improve the performance further, we design two new optimizations to reduce the number of injected faults: Early Detection of fault Saturation and Injection On Demand.

**Contributions.** As a summary, we claim the following novelties:

– We formalize the adversarial reachability problem (Section 4), extending standard reachability to take into account an advanced attacker, together with the associated correctness and completeness definitions;

– We describe a new symbolic exploration method (Section 5), adversarial symbolic execution, to answer adversarial reachability, featuring a novel forkless fault encoding to prevent path explosion and two optimization strategies to reduce fault injection. We establish their correctness and completeness;
– We propose an implementation of our techniques for binary-level analysis (Section 6), on top of the BINSEC framework [38]. We systematically evaluate its performances against prior work (Section 7), using a standard SWiFI benchmark from physical fault attacks and smart cards. Experiments show a very significant performance gain against prior approaches, for example up to x10 and x215 times on average for 1 and 2 faults respectively – with a similar reduction in the number of adversarial paths. Moreover, our approach scales up to 10 faults whereas the state-of-the-art starts to timeout for 3 faults ;
– We finally perform a security analysis of the WooKey bootloader [1] (Section 8), a very well tested real-life security-focused program. We were able to find known attacks and evaluate the adequacy of some of the countermeasures. Especially, we found an attack not taken into account in a recently proposed patch [63], and proposed a new patch to the developers.

This work is a first step in designing efficient program analysis techniques able to take into account advanced attackers. The approach is generic enough to accommodate many common fault models, including the bit flip from RowHammer, test inversion or arbitrary data modification; still, instruction skips or modifications are currently out of reach. Also, while we investigate the bug finding side of the problem (underapproximation), the verification side (overapproximation) is interesting as well. These are exciting directions for future research.

*Our dataset and benchmark infrastructure are made available through artifact*[2] *for reproducibility purpose, and the code is open-sourced*[3].

## 2 Motivation

We start by motivating the need for adversarial reachability, first with a description of several realistic attack scenarios on software involving advanced attackers (Section 2.1), second with a small example showing the need for dedicated analysis (Section 2.2).

### 2.1 Fault Injection across Security Fields

We describe hereafter several real software-level security scenarios where the attacker goes beyond crafting legitimate input to abuse the system under at-

[1] WooKey [14, 89] is a secure USB mass storage device developed by the French National Security Agency, and has recently served as a recent challenge among French security evaluators.

[2] DOI: 10.5281/zenodo.7507112
https://zenodo.org/record/7507112#.Y7cLsKfMJhE

[3] https://github.com/binsec/binsec-ase

tack. Interestingly, while these scenarios were historically focused on hardware-hardened high-security systems (such as smart cards) and associated with complex physical attack means, many recent scenarios do involve software-only attacks on standard systems, with targets encompassing cryptographic libraries, bootloaders, firmware updaters, security enclaves, etc.

**Hardware Fault Injection Attacks** [58] cause erroneous computations by disturbing signal propagation in the chip with physical means such as electromagnetic pulses [39], laser beams [4, 85], or power [19] and clock glitches. The associated fault models include bit-, byte- or word- set and reset, bit-flips, instructions corruption and instruction skips. State-of-the-art attacks involve multiple fault injections [59], as expected by the high level of attack potential in Common Criteria vulnerability analysis.

**Software-implemented Hardware Attacks** push the hardware into unstable states using software controlled mechanisms, like delays in memory buses inducing bit-flips in data fetched from memory [55] or CPU voltage and frequency manipulations yielding bit-flips in the processor [69,86]. The notorious *Rowhammer* attack [70] abuses memory accesses to induce bit-flips in flash memory.

**Micro-architectural Attacks** use micro-architectural behaviors in unexpected ways. For example: Spectre (version v1) [62] exploits branch predictors in speculative executions, which can be seen as a test inversion followed by a rollback; Load Value Injection [87] injects arbitrary data into transient execution; race attacks [54] corrupt data of other running processes and can be seen as arbitrary data faults.

**Man-At-The-End Attacks** considers an attacker having full observability and control over a software code and its execution [3], with the goal to steal sensitive data or code (reverse engineering attacks). The associated attacker model is hence very powerful, with capabilities such as halting and modifying data and code at any point of the execution.

**CFI Reasoning** In order to assess the power of Control-Flow Integrity (CFI) mechanisms, researchers [1,21] define hypothetical attackers by their capabilities, such as "write anything anywhere" or "write anything somewhere", and manually prove that their countermeasure is indeed able to thwart such an opponent. While not *per se* an applicative security scenario, the techniques developed in this paper could help automate such essential reasoning.

### 2.2 Motivating Example

The motivating example in Figure 1 is a simple unrolled program inspired by the VerifyPIN benchmark [42], from the domain of hardware fault injection and smart cards. The user PIN digits $u1$ to $u4$ are checked against the reference digits $ref1$ to $ref4$, using the accumulator $res$. The attacker seeks to be authenticated (validate the assert l.16) without knowing the right digits (l.14).

Here, the attacker indeed cannot succeed by only crafting legitimate inputs. However, an advanced attacker can leverage more powerful attack vectors to

```
 1 bool g_authenticated;
 2 int u1, u2, u3, u4, ref1 , ref2 , ref3 , ref4;
 3
 4 void verifyPIN() {
 5     int res = 1;
 6     res = res * (u1 == ref1);
 7     res = res * (u2 == ref2);
 8     res = res * (u3 == ref3);
 9     res = res * (u4 == ref4);
10     g_authenticated = res;
11 }
12
13 void main(int argc, char const *argv[]) {
14     assert(u1!=ref1 || u2!=ref2 || u3!=ref3 || u4!=ref4);
15     verifyPIN();
16     assert(g_authenticated == true); /* Security oracle */
17 }
```

Fig. 1: Motivating example, inspired by VerifyPIN [42]

inject faults into the program in order to succeed. For instance, corrupting *g_authenticated* to *true* at l.10 achieves the attacker goal. It could be obtained for example through a physical- or Rowhammer- attack.

**Program Analysis** As expected, standard symbolic execution tools such as Klee [22], angr [84] or BINSEC [38] do not report any violation here, as they consider the simplest possible attacker. We can try to use SWiFI techniques [15, 20, 63, 76] (detailed in Section 3.1) from high-security system evaluation. Yet, the standard *forking* approach does not scale with multiple faults: here, 166 paths are explored in 0.6 seconds for 1 fault, 2994 paths in 11 seconds for 2 faults, and it keeps on adding a factor x10 in explored paths and analysis time for each extra fault, until the analysis timeouts (12 hours) above 4 faults. On the contrary, our *forkless* algorithm presented in Section 5 simulates fault injection without creating new paths and, in this example, shows a constant runtime as the number of faults increases from 1 to 10 – we explore 9 paths in 0.2 seconds in all cases.

## 3 Background

We provide in this section background information on software-implemented fault injection, standard reachability and symbolic execution.

### 3.1 Software-implemented Fault Injection (SWiFI)

SWiFI tools [15, 20, 25, 28, 49, 50, 63, 68, 76, 79] have been developed in the community of high-secure systems to ease hardware fault injection campaigns, which are time consuming and require special equipment. SWiFI evaluates a program with the transformations induced by the effects of hardware faults, in order to find interesting attack paths. We distinguish two main SWiFI techniques.

First, the *Mutant generation* approach [25,28,49,50,79] consists in analyzing slightly modified versions of the program (named mutants), each of them embedding a different faulty instruction. Each mutant is then analyzed on its own. The main limitation of mutant generation is the explosion of mutants, in particular for multiple faults. Also, as the different mutants differ only slightly, analyzing each of them separately wastes lots of time repeating similar reasoning.

```
x := y + z
```

(a) Original statement

```
if (fault_here)
    then x := fault_value
    else x := y + z
```

(b) Forking transformation

Fig. 2: Forking code transformation in pseudo-code

Second, the *forking approach* [15, 20, 63, 76] consists in instrumenting the analysis (or the code, via instrumentation) to add all possible faults as forking points (branches) controlled by boolean values indicating whether a particular fault will be taken or not, plus constraints on the maximal number of faults allowed. A forking data fault is illustrated in Figure 2. A standard program analysis technique is then launched – typically symbolic execution or bounded model checking. Compared with mutant generation, this method allows sharing the analysis between the different possible faults. Still, the number of paths explodes with the number of possible faults (forking points).

**Scalability Issues** These two approaches yield an explosion of the whole search space w.r.t. the number of fault injection points in the program: the mutant approach leads to consider up to $C_k^n$ ($k$ among $n$)[4] mutants for a program under analysis with $n$ possible fault locations and $k$ faults, while the forking approach yields up to $C_k^n$ paths to analyzed for a single original program path with $n$ possible fault locations and $k$ faults.

*In the following, we will consider the forking approach as the baseline – please keep in mind that the mutant approach scales worse.*

**Fault Models** Supported fault models vary for each tool, but they are usually adapted from hardware fault models [47,82]. The most common fault models are (1) *data faults* such as arbitrary data modifications, set and reset of bytes, words or variables, bit-flips; and (2) *instruction corruptions* such as instruction skips and test inversions. Most tools are limited to one (sometimes two) hard-coded fault models. Only few SWiFI tools can handle multiple faults [63,68,76,88] – still with scalability issues.

[4] Remind that $C_k^n = \binom{k}{n} = \frac{n!}{k!(n-k)!}$

### 3.2 Standard Reachability Formalization

Considering a program $P$, we denote $S$ the set of all possible states of $P$. A state is composed of the code memory, the data memory (i.e. the stack and heap), the state of registers and the location of the next instruction to execute. The set of input states of a program $P$ is noted $S_0 \subset S$. The set of transitions (or instructions) of the program is denoted $T$. The execution of an instruction $t$ is represented by a one-step transition relation $\rightarrow_t \in S \times S$. We denote $s \rightarrow s'$ when $s \rightarrow_t s'$ for some $t \in T$. We extend the transition relation over any finite path $\pi \in T^*$ through composition. The transitive reflexive closure of $\rightarrow$ is noted $\rightarrow^*$. Finally, we use $S \rightarrow s'$ as a shortcut for $\exists s \in S.s \rightarrow s'$, and $\rightarrow_{\leq k}$ for reachability in at most $k$ steps.

We consider in the rest of the paper the case of *location reachability*: given a location $l$ (instruction or code address) of the program under analysis, the question is whether we can reach any state $s$ at location $l$. More formally, $L$ is the finite set of locations of $P$, and we consider a mapping $loc : S \mapsto L$ from states to locations. For example, $loc$ may return the program counter value. We write $S \rightarrow^* l$ as a shortcut for $\exists s' \in S.S \rightarrow^* s' \wedge loc(s') = l$.

**Definition 1 (Standard reachability).** *A location $l$ is reachable in a program $P$ if $S_0 \rightarrow^* l$.*

We now define correctness and completeness for a program analyzer.

**Definition 2 (Correctness, completeness).** *Let $\mathcal{V} : (P, l) \mapsto \{1, 0\}$ be a verifier taking as input a program $P$ and a target location $l$.*
- *$\mathcal{V}$ is correct when for all $P$, $l$, if $\mathcal{V}(P, l) = 1$ then $l$ is reachable in $P$ ;*
- *$\mathcal{V}$ is complete when for all $P$, $l$, if $l$ is reachable then $\mathcal{V}(P, l) = 1$ ;*
- *if $\mathcal{V}$ also takes an integer bound $n$ as input, $\mathcal{V}$ is k-complete when for all bound $n$ and $P$,$l$, if $l$ is reachable in at most $n$ steps then $\mathcal{V}(P, l, n) = 1$.*

We want to stress out that while location reachability can be seen as a basic case, we consider it sufficient here for two reasons: first, it keeps the formalism light while still straightforward to generalize to stronger reachability properties (e.g., local predicates of the form $(l, \varphi)$, sets of finite traces, etc.); second, it is already rather powerful on its own, as we can still instrument the code to reduce some stronger forms of reachability to it (e.g., adding local assertions or monitors).

### 3.3 Symbolic Execution

Symbolic execution (SE) [23, 24, 52, 83] is a symbolic exploration technique for standard reachability. Algorithm 1 gives a high-level view of a typical SE algorithm, adapted for location reachability[5]. The analysis follows each possible

[5] More complex properties can be verified with the same principles, such as local predicate reachability, trace properties or hyper-properties [36].

**Algorithm 1:** Standard symbolic execution algorithm, taken from [48]

```
Input: a program $P$, a bound $k$, a target location $l$
Output: Boolean value indicating whether $l$ can be reached within $k$ steps.
for path $\pi$ in GetPaths($k$) do
    if $\pi$ reaches $l$ then
        $\Phi$ := GetPredicate($\pi$)
        if $\Phi$ is satisfiable then
            return true
        end
    end
end
return false
```

path $\pi$ of a program up to a depth bound $k$. If $\pi$ reaches the target, then we check whether $\pi$ is indeed feasible by computing its *path predicate* $\Phi$ – a logical formula representing the path constraints over the input variables along $\pi$, and sending it to a SMT solver [12], that will try to answer whether the formula is satisfiable or not, and provide a model for free variables (e.g. inputs) if it is (omitted here for simplicity). SE is *correct* for location reachability, and even *k-complete* if we assume a perfect encoding of path predicates.

**Algorithm 2:** Assignment evaluation in SE

```
Input: path predicate $\Phi$, assignment instruction $x := expr$
Output: Updated $\Phi$
Function eval_assign($\Phi$, $x$, $expr$) is
    return $\Phi \wedge (x \triangleq expr)$
end
```

In this paper, we will focus on the evaluation of assignments and conditional jumps for SE, detailed in Algorithms 2 and 3 respectively, as this is where our adversarial symbolic execution will mainly differ from the standard one. It requires going slightly deeper into details. In practice, the program paths are explored incrementally. A worklist $WL$ records all pending paths together with their associated path predicate and their next instruction to explore. On conditional branches, the symbolic path is split in two (one for each branch, updating the path constraint accordingly), and each new prefix is added to the worklist (Algorithm 3). Assignments are dealt with straightforwardly, simply adding a new logical variable definition to the path predicate [6] (notation: $x \triangleq y$).

[6] Actually, a symbolic state usually comprises the path predicate itself plus a mapping from program variable names to logical variable names, and assignments involve both

**Algorithm 3:** Conditional jump evaluation in SE

```
Input: path predicate $\Phi$, conditional jump instruction $if\ cdt\ then\ l_t\ else\ \ l_e$
Data: a worklist $WL$ containing the pending path prefixes to explore – list of
      pairs (path predicate, next location)
Output: $WL$ updated in place
1 Function eval_conditional_jump($\Phi$, $cdt$, $l_t$, $l_e$) is
2     if $\Phi \wedge cdt$ is satisfiable then
3         Add $(\Phi \wedge cdt, l_t)$ to $WL$
4     end
5     if $\Phi \wedge (\neg cdt)$ is satisfiable then
6         Add $(\Phi \wedge \neg cdt, l_e)$ to $WL$
7     end
8 end
```

## 4 Adversarial Reachability

In this section, we detail the advanced attacker model we consider and define the adversarial reachability problem. Especially, *advanced attackers can do more than carefully crafting legitimate inputs to trigger vulnerabilities in a software.* They can use a wide variety of attack vectors (e.g. hardware fault injection attacks, software-implemented hardware attacks, micro-architectural attacks, software attacks, etc), in any combination, and multiple times. We suppose attack vectors prerequisites have been met, and only consider the impact of the faults on the program under attack.

Our *attacker model* has three components: (1) a set of attacker actions, equivalent to fault models; (2) the maximum number of actions the attacker can perform; and (3) a goal, expressed here as a location reachability query.

Formally, given a program $P$ with set of states $S$, set of transitions $T$ and set of locations $L$, we extend the transition model described in Section 3.2 to include an adversarial transition $\rightsquigarrow_A \in S \times S$ related to an attacker $A$, i.e. $T_A = T \cup \rightsquigarrow_A$. To specify practical fault models, restrictions are applied onto $\rightsquigarrow_A$, limiting what part of the state can be modified and how. For instance, when considering arbitrary data faults, only the data memory and the register values can be modified. Then, the transition relation of $P$ under attacker $A$ is denoted as $\mapsto_A = \rightarrow \cup \rightsquigarrow_A = (\cup_{t \in T} t) \cup \rightsquigarrow_A$. We extend the notations from Section 3.2 to the relation $\mapsto_A$. Especially, $S \mapsto_A^* s'$ means $\exists s \in S.s \mapsto_A^* s'$, the adversarial transition relation up to $k$ is denoted $\mapsto_{A,\leq k}$.

Still, we need to take into account the maximum number of faults the attacker can perform along an execution. Given a path $\pi$ over $T_A^*$, $\pi$ is said to be *legit* if it does not contain $\rightsquigarrow_A$, and *faulty* otherwise. The number of occurrences of transition $\rightsquigarrow_A$ in $\pi$ is its *number of faults.* Given a bound $m_A$ on the fault

creating new logical names and updating the mapping. We abstract away from these details.

capability of $A$, we define $\mapsto^*_{(A,m_A)}$ by limiting the adversarial reachability relation to paths $\pi$ with less than $m_A$ faults. We consider $m_A$ to be $+\infty$ in case the attacker has no such limitation. For the sake of simplicity, in the following, we will consider $m_A$ as an implicit parameter of $A$, and simply write $\mapsto^*_A$ instead of $\mapsto^*_{(A,m_A)}$.

**Definition 3 (Adversarial reachability).** *Given an attacker $A$ with a $m_A$ faults budget and a program $P$, a location $l \in L$ is adversarially reachable if $S_0 \mapsto^*_A s' \wedge loc(s') = l$ for some $s' \in S$.*

In the following, adversarial reachability of location $l$ from a set of states $S_0$ will be denoted $S_0 \mapsto^*_A l$.

**Proposition 1.** *Standard reachability implies adversarial reachability. The converse does not hold.*

*Proof.* Standard reachability can be viewed as adversarial reachability with an attacker able to perform 0 faults.

We redefine what it means for an analysis answering adversarial reachability to be correct, complete and k-complete.

**Definition 4.** *Let $\mathcal{V}_A : (P, A, l) \mapsto \{1, 0\}$ be a verifier taking as input a program $P$, an attacker $A$ with $m_A$ fault budget and a target location $l$.*

- *$\mathcal{V}_A$ is correct given $A$ when for all $P$, $l$, if $\mathcal{V}_A(P, A, l) = 1$ then $l$ is adversarially reachable in $P$ for attacker $A$;*
- *$\mathcal{V}_A$ is complete given $A$ when for all $P$, $l$, if $l$ is adversarially reachable for attacker $A$ then $\mathcal{V}_A(P, A, l) = 1$ ;*
- *if $\mathcal{V}_A$ also takes an integer bound $n$ as input, $\mathcal{V}_A$ is k-complete given $A$ when for all integer $n$ and $P$,$l$, if $l$ is adversarially reachable in at most $n$ steps then $\mathcal{V}_A(P, A, l, n) = 1$.*

## 5 Forkless Adversarial Symbolic Execution (FASE)

In this section, we present our forkless algorithm for adversarial reachability. The analysis aims to find inputs and a fault sequence compatible with the considered attacker model and reaching the target location. Our primary goal is to deal with the potential path explosion induced by possible faults. Our design guiding principles are the following:

- First, prevent path explosion as much as possible with a forkless fault encoding. Yet, this forkless encoding leads to logical formulas potentially more complex and harder to solve in practice;
- Second, reduce as much as possible the complexity of the created formulas, by avoiding the undue introduction of extra-faults along a path.

### 5.1 Modelling Faults via Forkless Encoding

The forkless encoding aims to address the path explosion induced by the forking treatment of fault injection in prior works. It is designed mainly for data faults and consists of wrapping arithmetically an assignment right-hand side, as shown in Figure 3 for an arbitrary data fault. The activation of this fault location is determined by the symbolic Boolean value $fault\_here$, and the corrupted value of $x$ is the fresh variable $fault\_value$.

The point is to embed the fault injection as an expression inside the logical formula, without any explicit path forking at the analysis top-level, in order to let the analyzer reason about both legit executions and faulty executions at the same time – this is akin to path merging in some ways, except that we do it only for the treatment of fault injection (we could also see the approach as avoiding undue path splits).

Multiple forkless arbitrary data encodings are possible. We chose to use the *ite* expression operator, an inlined form of if-then-else at the expression level. We also tried encodings inspired from branchless programming idioms (e.g.: $(b)\cdot x+(1-b)\cdot y$. for $ite(b,x,y)$ with $b$ a Boolean value) – in our experiments they worked as well as the *ite* operator. Other data fault models are supported, such as set, reset, bit-flips, etc. Test inversion is also supported by applying faults to the condition of conditional jumps. Table 1 illustrates various forkless encodings. Note that the forkless encoding is not designed for instruction corruptions or instruction skips, as these modifications either yield permanent code modification or span several instructions.

```
x := expr
```

(a) Original statement

```
x:=ite fault_here? fault_value:expr
```

(b) Forkless transformation for arbitrary data fault

Fig. 3: Forkless injection technique

Table 1: Forkless encodings for various fault models

| Fault model | original instruction | Forkless encoding |
|---|---|---|
| Arbitrary data | $x := expr$ | $x := ite\ fault\_here\ ?\ fault\_value\ :\ expr$ |
| Variable reset | $x := expr$ | $x := ite\ fault\_here\ ?\ 0x00000000\ :\ expr$ |
| Variable set | $x := expr$ | $x := ite\ fault\_here\ ?\ 0xffffffff\ :\ expr$ |
| Bit-flip | $x := expr$ | $x := ite\ fault\_here\ ?$ $(expr\ xor\ 1 << fault\_value)\ :\ expr$ |
| Test inversion | $if\ cdt\ then\ goto\ 1$ $else\ goto\ 2$ | $if\ (ite\ fault\_here\ ?\ !cdt\ :\ cdt)$ $then\ goto\ 1\ else\ goto\ 2$ |

**Trade-off.** While these sorts of encoding indeed allow a significant path reduction compared to forking approaches, the corresponding path predicates are more complicated than standard path predicates, as they involve lots of extra-symbolic variables for deciding whether the faults occur and for emulating their effect. We show later in this section how to reduce these extra-variables.

### 5.2 Building Adversarial Path Predicates

Adversarial symbolic execution requires modifications to Algorithms 2 and 3, as illustrated in Algorithms 4 and 5 respectively.

**Algorithm 4:** Forkless assignment evaluation

```
Input: path predicate $\Phi$, assignment instruction $x := expr$, current number of
       faults $nb_f$
Output: Updated $\Phi$
1 Function eval_assign($\Phi$, $x$, $expr$) is
2     $\Phi'$, $expr'$, $nb_f$ := FaultEncoding($\Phi$, $expr$, $nb_f$)
3     return $\Phi' \wedge (x \triangleq expr')$
4 end
```

**Algorithm 5:** Forkless conditional jump evaluation

```
Input: path predicate $\Phi$, conditional jump instruction $if\ cdt\ l_t\ else\ l_e$
Data: fault counter $nb_f$, maximal number of faults $max_f$, worklist $WL$
Output: $WL$ updated in place
1 Function eval_conditional_jump($\Phi$, $cdt$, $l_t$, $l_e$) is
2     if $\Phi \wedge cdt \wedge (nb_f \leq max_f)$ is satisfiable then
3         Add $(\Phi \wedge cdt, l_t)$ to $WL$
4     end
      /* Idem for else branch ($\neg cdt$)                                */
5 end
```

The assign evaluation process embeds a wrapper encoding the fault in a forkless manner. Note that $FaultEncoding$ involves the declaration of fresh symbolic variables for fault decisions and fault effects – hence the update of the path predicate $\Phi$. Also, the fault counter $nb_f$ is updated, and a new potentially faulted expression $expr'$ is computed.

Note that checking if the fault counter $nb_f$ does not exceed the maximal number of faults $max_f$ can be performed at different places. We found the best trade-off is to augment the conditional jump queries to check if we could explore each branch without exceeding $max_f$ (see Algorithm 5), as checking at the end of a path often involves exploring many unfeasible faulty paths.

*We refer to this set of modifications as Forkless Adversarial Symbolic Execution (FASE).*

### 5.3 Algorithm Properties

We now consider the properties of the FASE algorithm.

**Proposition 2.** *The FASE algorithm is correct and k-complete for adversarial reachability.*

*Sketch of proof.* If our algorithm finds an adversarial path reaching the target location $l$, by providing specific input values and a fault sequence, then an attacker executing the program with the provided inputs and performing the proposed faults will reach its goal. Our algorithm is based on symbolic execution with bounded path depth and explores all possible attack paths according to the considered attacker model, hence its k-completeness for adversarial reachability.

**Tightness of FASE.** Consider a single path with no branching instruction and an assert statement to be checked at the end, together with $f$ possible fault locations and a maximum of $m$ faults. Then the forking SE yields up to $C_m^f$ paths to analyze, and as many queries to send to the solver. In the same scenario, FASE will analyze only the original path, and *send a single query to the solver.*

Still, the Forkless encoding increases query complexity, as shown in Section 7. We present in the remainder of this section two mitigation techniques.

### 5.4 Optimization via Early Detection of Fault Saturation (FASE-EDS)

**Algorithm 6:** FASE-EDS conditional jump evaluation

```
Input: path predicate $\Phi$, conditional jump instruction $if\ cdt\ then\ l_t\ else\ l_e$
Data: fault counter $nb_f$, maximal number of faults $max_f$, worklist $WL$
Output: $WL$ updated in place
1 Function eval_conditional_jump_EDS($\Phi$, $cdt$, $l_t$, $l_e$) is
2     if $\Phi \wedge cdt \wedge (nb_f < max_f)$ is satisfiable then
3         Add ($\Phi \wedge cdt$, $l_t$) to $WL$
4     else if $\Phi \wedge cdt \wedge (nb_f == max_f)$ is satisfiable then
5         Stop injection in this path
6         Add ($\Phi \wedge cdt$, $l_t$) to $WL$
7     end
      /* Idem for else branch ($\neg cdt$)                              */
8 end
```

The first angle we explore to minimize query complexity is to reduce the number of injection points by *stopping the injection process as soon as possible*. Indeed, fewer injection points mean fewer extra symbolic variables and in general smaller and simpler queries for the SMT solver. We call this optimization *Early Detection of fault Saturation*, and write FASE-EDS when it is activated.

Its difference compared to FASE is in handling conditional jumps, illustrated in Algorithm 6. Instead of checking whether a branch can be explored without exceeding the maximum number of faults, we double the check: (1) first we check whether the branch can be explored with strictly fewer faults than allowed. If the query is satisfiable, the analysis continues down that branch as usual; (2) if not satisfiable, we check whether the branch is feasible with exactly the maximal number of faults allowed. If not, the branch is infeasible and we stop as usual. Yet, if it is feasible, then we know that we have spent all allowed faults. We can thus continue the exploration *without injecting any new fault* in the corresponding search sub-tree, leading to simpler subsequent queries.

**Proposition 3.** *FASE-EDS is correct and k-complete for the adversarial reachability problem.*

*Proof.* FASE-EDS remains correct as it does not modify the path predicate computation, and it remains k-complete as it only prunes fault injections that are actually infeasible – and would have been proven so by the solver, later in the solving process.

## 5.5 Optimization via Injection on Demand (FASE-IOD)

The second angle explored to reduce query complexity through the reduction of injection points is *to inject faults on demand*, only when they are truly needed. We call this optimization *Injection On Demand*, and write FASE-IOD when it is activated.

To inject faults on demand, we now build *two* path predicates along a path: the working path predicate $\Phi$ based on which solver queries are built (where we try to minimize fault injection), and the normal adversarial path predicate $\Phi_F$ computed in previous sections (encompassing all the faults seen so far).

**Algorithm 7:** FASE-IOD assignment evaluation

```
Input: path predicate $\Phi$, faulted path predicate $\Phi_F$, assignment instruction
       $x := expr$, current number of faults (in $\Phi_F$) $nb_f$
Output: Updated $\Phi$, $\Phi_F$
1 Function eval_assign_IOD($\Phi$, $\Phi_F$, $cdt$, $x$, $expr$) is
2     $\Phi'_F$, $expr'$, $nb_f$ := FaultEncoding($\Phi_F$, $expr$, $nb_f$)
3     return $(\Phi \land (x \triangleq expr), \Phi'_F \land (x \triangleq expr'))$
4 end
```

**Algorithm 8:** FASE-IOD conditional jump evaluation

```
Input: path predicate $\Phi$, conditional jump instruction $if\ cdt\ then\ l_t\ else\ l_e$
Data: fault counter $nb_f$, maximal number of faults $max_f$, under
      approximation counter $under\_counter$, worklist $WL$
Output: $WL$ updated in place
1  Function eval_conditional_jump_IOD($\Phi$, $\Phi_F$, $cdt$, $l_t$, $l_e$) is
2      if $\Phi \wedge cdt \wedge (nb_f \leq max_f)$ is satisfiable then
3          Add ($\Phi \wedge cdt$, $\Phi_F \wedge cdt$, $l_t$) to $WL$
4      else if $under\_counter \leq max_f$ then
5          if $\Phi_F \wedge cdt \wedge (nb_f \leq max_f)$ is satisfiable then
6              $\Phi := \Phi_F$
7              $under\_counter := under\_counter + 1$
8              Add ($\Phi \wedge cdt$, $\Phi_F \wedge cdt$, $l_t$) to $WL$
9          end
10     end
       /* Idem for else branch ($\neg cdt$)                              */
11 end
```

Algorithms are updated accordingly. Especially, assignment evaluation is duplicated as shown in Algorithm 7: The normal symbolic assignment, with the original right-hand-side expression *expr*, is added to $\Phi$, while $\Phi_F$ is updated with the fault encoding of the assignment, $expr'$.

The on-demand reasoning takes place in the conditional jump instruction process detailed in Algorithm 8. The basic idea is to first check branch feasibility with the simpler path predicate $\Phi$, encompassing the least number of faults. We continue this way as long as we can, meaning we rely on standard reachability as much as we can.

When encountering a branch infeasible with $\Phi$, we then check whether this branch is feasible with all the possible faults seen so far, i.e. using $\Phi_F$. If no that is a stop, otherwise we know that $\Phi$ does not encompass enough faults to go further. We then replace $\Phi$ by $\Phi_F$ (called a *switch*) at this stage, and thus continue with strictly more faults. Note that this is straightforward as $\Phi_F$ and $\Phi$ only differ on fault injections. Then again, the new $\Phi$ will not accumulate any fault (until a new switch) while $\Phi_F$ continues accumulating all possible faults.

As a bonus, the number of path predicate switches gives us an under-approximation $under\_counter$ of the number of faults already needed in the path under analysis. We use it to stop the injection early, when at least $max_f$ faults have been used.

**Proposition 4.** *FASE-IOD is correct and k-complete for the adversarial reachability problem.*

*Proof.* FASE-IOD explores the same feasible paths as FASE, hence preserving its properties.

### 5.6 Optimizations Combination

**Algorithm 9:** FASE-IOD and FASE-EDS combination, conditional jump evaluation

```
Input: path predicate $\Phi$, faulty path predicate $\Phi_F$, conditional jump
       instruction $if\ cdt\ then\ l_t\ else\ \ l_e$
Data: fault counter $nb_f$, maximal number of faults $max_f$, under
      approximation counter $under\_counter$, worklist $WL$
Output: $WL$ updated in place
Function eval_conditional_jump_EDS_IOD($\Phi$, $\Phi_F$, $cdt$, $l_t$, $l_e$) is
    if $\Phi \wedge cdt \wedge (nb_f \leq max_f)$ is satisfiable then
        Add ($\Phi \wedge cdt$, $\Phi_F \wedge cdt$, $l_t$) to $WL$
    else if $under\_counter \leq max_f$ then
        if $\Phi_F \wedge cdt \wedge (nb_f < max_f)$ is satisfiable then
            $\Phi := \Phi_F$
            $under\_counter := under\_counter + 1$
            Add ($\Phi \wedge cdt$, $\Phi_F \wedge cdt$, $l_t$) to $WL$
        else if $\Phi_F \wedge cdt \wedge (nb_f == max_f)$ is satisfiable then
            $\Phi := \Phi_F$
            Stop $\Phi'$ update and queries
            Add ($\Phi \wedge cdt$, $\Phi_F \wedge cdt$, $l_t$) to $WL$
        end
    end
    /* Idem for else branch ($\neg cdt$)                                  */
end
```

Both optimizations can be combined together as illustrated in Algorithm 9. Taking FASE-IOD as a basis, saturation detection is added in the faulted path predicate $\Phi_F$ queries at conditional branch handling. If the saturation is detected, the main path predicate switch to $\Phi_F$ but $\Phi_F$ stops being updated and queried further down that path, which stops fault injection.

**Proposition 5.** *The combination of FASE-EDS and FASE-IOD is correct and k-complete for the adversarial reachability problem.*

*Proof.* This combination also explores all possible paths for the considered attacker models, like FASE, hence preserving its properties.

## 6 Implementation

We now provide details about our forkless adversarial symbolic execution (FASE) implementation, named BINSEC/ASE, for Adversarial Symbolic Execution. *The code is made open-source*[7].

[7] https://github.com/binsec/binsec-ase

**Binary-level Fault Injection.** While our method works for any program abstraction level, we choose to implement it for the binary level, which makes more sense in many security scenarios. We implement our forkless adversarial symbolic execution on top of the BINSEC symbolic engine [10,38,40]. It has already been used in a number of significant case studies [9, 36, 37, 80, 81], and it is notably able to achieve bounded verification (k-completeness) and to reasonably deal with symbolic pointers [44].

We modified the path predicate computation of BINSEC 0.4.0 as described in Section 5, and implemented our dedicated optimizations FASE-EDS, FASE-IOD and FASE EDS+IOD. BINSEC consists of 60kloc of Ocaml and our modifications add 6kloc. The attacker goal is specified as a local predicate to reach, using BINSEC directives. We currently support data faults such as arbitrary modification, bit-flip and reset. Test inversion is emulated through faulting the condition of conditional jumps. We let the user define an injection target range, made of multiple code address intervals. For large programs, it enables focusing on the security critical sections. Finally, we also provide a blacklist for some memory locations which will never be faulted. The blacklist is mostly used for the stack register (`esp` in x86, which is concretized in the analysis) and the program counter, as our fault model does not include tampering with the stack nor arbitrary control faults.

**Details.** Our exploration strategy is depth first, the underlying SMT solver is Bitwuzla [71]. We constrain the faulted values to differ from the original values in fault encodings, such that only true corruptions are reported as active faults.

## 7 Evaluation

We now evaluate our new algorithm for software verification against multi-fault attacks. We consider the following research questions.

- **RQ1**: is our tool correct and complete? In particular, can we find attacks on vulnerable programs and prove secure resistant programs?
- **RQ2**: can we scale in number of faults without path explosion?
- **RQ3**: what is the impact of our optimizations?

Besides this evaluation, we also show the use of our method in a number of different security scenarios (Section 7.5), and on a larger case study (Section 8).

### 7.1 Experimental Setting

**The Machine Used.** We ran our experiments on a cloud machine with a processor Intel Dual Xeon 4214R with 48 CPU cores and 384GB of RAM. Experiments ran in parallel on the 48 cores, each run using only one core.

**The Attacker Model** chosen in this evaluation can perform a varying number of faults. Its goal is expressed as a security oracle directly written in C for each benchmark, the computation of which is not faulted.

**The Benchmark** used here is a standard set of programs from the SWiFI literature on physical fault injections and high-security devices, characterized in Table 2. First, the 8 versions of VerifyPIN from the FISSC [42] benchmark suite, dedicated to the evaluation of physical fault attack analyses. VerifyPIN is an authentication program. There are one unprotected and 7 different protected versions, some vulnerable, some resistant to one test inversion fault. We added 2 manually unrolled versions of the unprotected VerifyPIN, with a PIN size of 4 and 16, to add diversity in the benchmarks with programs without loops. An oracle is provided by FISSC, checking if the user PIN truly corresponds to the reference PIN. Second, we take the 2 versions of the npo2 program from Le et al. [65], together with their oracles. Npo2 is a program computing an integer's upper power of two. The attacker's goal is to perform a silent data corruption, i.e. change the end result without triggering countermeasures. One version is vulnerable to one arbitrary data fault, the second is resistant due to extra arithmetic checks.

**Compilation.** The benchmarks are written in C and have been compiled with gcc for the Intel x86-32 architecture, using the flag "-O0" to preserve countermeasures. For BINSEC compatibility, we use the "-static" flag to include the necessary library functions directly in the binary.

Table 2: Benchmarks characteristics and statistics of a standard SE analysis

| | | | BINSEC analysis - no fault | | | |
|---|---|---|---|---|---|---|
| Program group (#) | C loc | x86 loc | #instruction (explored) | #paths | #branch in a path | Time |
| Section 7 | | | | | | |
| VerifyPINs (8) | 80-140 | 160-215 | 192-269 | 1 | 17-34 | $< 0.1$s |
| VerifyPIN unrolled (2) | 40-85 | 140-430 | 142-442 | 5-17 | 5-17 | $< 0.1$s |
| npo2 (2) | 50 | 200-220 | 607-653 | 3 | 31-33 | $< 0.1$s |
| Section 8 | | | | | | |
| WooKey bootloader | 3.2k | 2350 | 290k | 17 | 18k | 9s |
| Section 7.5 | | | | | | |
| CRT-RSA (3) | 125-170 | 400-600 | 108k-29M | 1 | 5k-1.3M | 0.4s - 1m27 |
| Secret keeping machine (2) | 100-200 | 240-360 | 1k-1.3k | 1 | 130-150 | $< 0.1$s |
| VerifyPIN_0 with SecSwift | 80 | 430 | 430 | 1 | 22 | $< 0.1$s |

**BINSEC Settings.** We limit the maximal depth of an analysis to the depth necessary to perform an exhaustive non-faulty analysis, rounded to the upper hundred. We exhaustively explore all the possible paths up to this bound and do not stop at the first identified attack, in order to have comparable results. We set the global analysis timeout for 1 day. We fault values and not addresses, we do not directly fault the stack pointer nor the program counter, and we do not fault the status flags unless explicitly specified.

## 7.2 Correctness and Completeness in Practice (RQ1)

We first show that our tool works as expected on several codes with known ground truth. (1) We check that indeed, with no fault allowed, no attack is found in any of the benchmarks; (2) We check that indeed the insecure npo2 program is vulnerable to a single arbitrary data fault while the secure version is not – it can still be exploited with two faults; (3) According to their authors, the VerifyPIN versions 0 to 4 are vulnerable to one test inversion, while VerifyPIN 5 to 7 are resistant to it. We indeed reproduce these results. When allowing two faults, all VerifyPIN become vulnerable; (4) When using one arbitrary data fault against the VerifyPINs, all versions are found vulnerable. We manually check that indeed the identified attack paths make sense; (5) Our manually unrolled versions of VerifyPINs do not contain conditional branching instructions in the targeted function, making them resistant to test inversion. We check that this is the case, while they are still vulnerable to a single arbitrary data fault.

**Conclusion.** Our tool indeed can showcase a program vulnerability to fault injection attacks and prove resistance to fault injection attacks, as expected by the correctness and k-completeness properties of the underlying algorithms.

## 7.3 Scalability (RQ2)

For this evaluation, we focus on an attacker capable of arbitrary data faults, as those weigh the heaviest on the analysis.

We take FASE-IOD as our best performing technique (see Section 7.4). We evaluate here its capability to handle multi-fault and avoid path explosion, compared to the forking technique. Results are illustrated in Figures 4 and 5. Note that all FASE variants explore the same number of paths, and are thus represented as FASE in Figure 5. For each benchmark, we took the arithmetic mean for 100 runs. Values presented here are the geometric mean over the benchmarks.

FASE-IOD is 10x times faster than Forking for 1 fault, and x200 times faster for 2 faults on average. For the best case benchmark, we are x224 times faster for 1 fault and x6121 for 2. Starting from three faults onward, Forking experiences timeouts, rendering values non comparable. Half of the benchmark timeouts for 3 faults, three quarters for 4 faults, 11 over 12 for 6 faults and all of them after that. FASE-IOD never timeouts in this experiment. This scaling is enabled by avoiding path explosion. On average, Forking explores x50 times more paths for 2 faults than for one, while FASE-IOD only explores x3 times more paths. From Figure 4, we see FASE on its own already scales better than Forking, being x3 times faster for 1 fault and x108 times faster for 2, and never experiencing timeouts either.

**Conclusion.** FASE-IOD shows improved scalability in terms of the maximum number of faults allowed, for the arbitrary data fault model, compared to the forking technique.

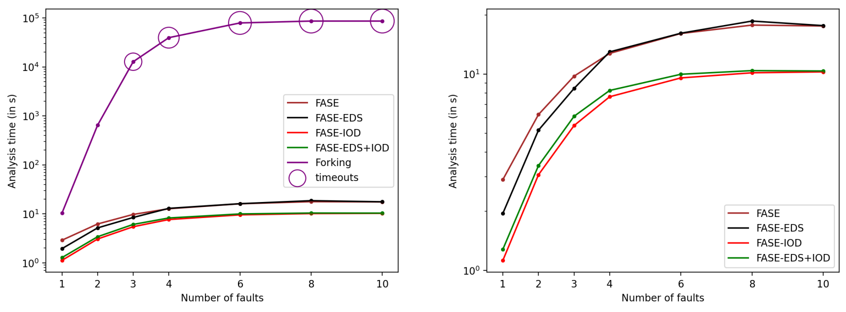

Fig. 4: Analysis time

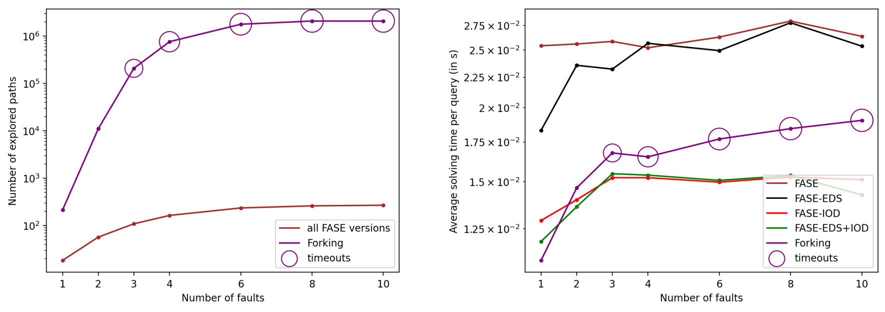

Fig. 5: Average number of explored paths, Average solving time per query

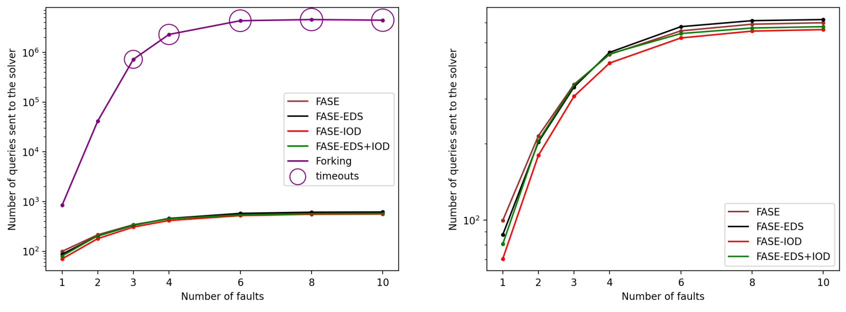

Fig. 6: Number of queries sent to the solver

### 7.4 Performance Optimization (RQ3)

We evaluate our forkless variants: FASE, FASE-EDS, FASE-IOD and FASE EDS+IOD, to determine which performs best for arbitrary data faults. Results are illustrated in Figures 4, 5 and 6.

We vary again the maximum number of faults from 1 to 10. Note that all FASE variants explore the same number of paths for each number of faults, as the optimizations reduce the number of faults injected but do not lose correctness nor k-completeness. FASE indeed generates complex queries[8], taking on average around twice the time necessary for Forking queries to be solved. FASE-EDS then gains a little bit in that regard. FASE queries take only x1.04 longer to solve on average for all fault numbers. The real improvement comes with the On-Demand logic of FASE-IOD (x2.02 times faster on average over all fault numbers) and FASE EDS+IOD (x2.02 also), where query complexity drops to the level of Forking. This improvement in query complexity is achieved algorithmically at the price of query creation. However, due to more queries being arithmetically simplified, fewer queries are sent in the end to the solver for FASE-IOD (x0.88 on average over all fault values compared with FASE) and FASE EDS+IOD (x0.98). FASE-EDS sent approximately the same number of queries as FASE. The number of queries sent to the solver explodes for Forking, correlated with the path explosion experienced. In terms of performance, two trends appear as the number of faults allowed increases. FASE and FASE-EDS tend to be between x2 and x3 times slower than FASE-IOD and FASE EDS+IOD. In the end, FASE-IOD proves to be the fastest optimization (x1.1 times faster than FASE EDS+IOD on average over all number of faults), likely due to the combination of on-demand logic and fewer queries than FASE EDS+IOD.

**Conclusion.** We retain FASE-IOD as our best performing forkless adversarial algorithm, at most x3.06 faster than FASE.

### 7.5 Other Experiments and Fault Models

**CRT-RSA.** Puys et al. [78] describe three versions of CRT-RSA: unprotected, Shamir version and Aumuller version. Only the last one is shown to resist the BellCoRe attack [16] which uses a single reset fault to break the cryptography. We were able to automatically reproduce the attack with 1 reset fault on the unprotected version of CRT-RSA, after 3s of analysis, and we were not able to find attacks on the other two versions in 10 days time.

**Secret-keeping Machine.** Dullien [41] proposes two versions of a secret-keeping machine. The one based on linked lists is manually shown to be exploitable by an attacker able to perform a single bit-flip in the memory (not in registers), while the array version is shown to be secure against that. For this benchmark,

[8] When counting the number of ite operators introduced in queries, from having barely any in a run without faults, we reach around 2,800 ite per query on average for FASE and 1,500 for FASE-IOD for one fault.

we activated faults on variables used as addresses. We were able to reproduce the attack on the linked list implementation with one bit-flip fault and to show the array implementation is secure for this fault model. In addition, if we allow faults in registers too, the array implementation becomes vulnerable.

**SecSwift Countermeasure.** We applied the SecSwift countermeasure, a llvm-level protection developed by STMicroelectronics [27,45], to VerifyPIN version 0. We were able to find attacks yielding an early loop exit on this binary with either a single test inversion or a single arbitrary data fault. These paths belonging to the CFG of the program, these attacks are not unexpected, yet it is still interesting that our method finds them automatically.

## 8 Case Study: the WooKey Bootloader

We now confront our tool to a real-life security system, WooKey.

**Presentation of WooKey.** First presented in 2018 by ANSSI, the French system security agency, the WooKey platform [14, 89] is "a custom STM32-based USB thumb drive with mass storage capabilities designed for user data encryption and protection, with a full-fledged set of in-depth security defenses". Their choice to be open source and open hardware makes WooKey a relevant case study: it is a real-life, complex device, security focused and available for reproducibility. Note also that Wookey has been extensively analyzed, as it was the target of an ANSSI cybersecurity challenge for security professionals [5].

**Security Scenario and Goal of our Study.** We focus on WooKey bootloader, a dual-bank system enabling hot firmware updates. The system is hardened, especially redundant test protections are present in critical sections to protect against test inversion faults. We consider the same attacker model as the ANSSI challenge did [5]: the attacker seeks to manipulate the bootloader logic to boot on the older firmware, more likely to contain security vulnerabilities. We also consider an attacker able to perform a single arbitrary data fault. We see in Table 2 that WooKey bootloader size is orders of magnitude larger than the programs used for evaluation in Section 7. Wookey is available as C code. We compile it like we did for the evaluation benchmarks (Section 7.1).

We conduct the following three analyses:

1. automatically analyze WooKey at binary-level to check whether we are able to find previously known faults [63], and/or new ones: we are indeed able to find the two faults identified by prior work [63] (A1, A2), *as well as an attack they do not mention* (A3);
2. automatically analyze at binary-level the patch version of Wookey proposed by Lacombe et al. [63]: we found that the proposed patch indeed blocks the two known attacks (A1 and A2), but not the new attack (A3);
3. propose a definitive patch by adding a counter-measure for A3 and remove parts of the counter-measures which are shown to be useless here. The patch is proven correct w.r.t. our attack model.

We discuss these results in the following and we present briefly in Section 8 the discovery of two more known faults. Overall, it demonstrates that our technique can scale to binary-level real-size systems.

**Analyze Key Parts of Wookey.** Lacombe et al. find an attack in the *loader_exec_req_selectbank* function (A1) and another in the *loader_exec_req_flashlock* function (A2). They correspond to data corruption in branching conditions. We are able to find both attacks, linking faults back to their locations in the C code with debug information. *We also find an additional attack*, faulting another part of the *loader_exec_req_flashlock* function (A3).

**Analyze a Security Patch of WooKey.** We now evaluate the protection scheme proposed by Lacombe et al. [63] for these attacks. It consists of four extra counter-measures named from CM1 to CM4. We found indeed that the full protection prevents attacks A1 and A2, as claimed by the authors of the patch. Yet, our analysis shows that the protection does not prevent the new attack A3.

**Propose a New Patch and Evaluate It.** We manually inspect these different analysis results to understand what happens. We have especially been able to identify the root cause of A3 and propose a dedicated countermeasure for it (named CMA). Also, by analyzing each counter-measure in isolation, we have been able to understand that counter-measures CM1 and CM3 do not block any attack path as they are redundant with other tests in the code and can be safely removed. Overall, our new patch (CMA + refined former patch) is shown by our tool to protect against all the attacks, for an attacker able to perform one arbitrary data fault.

Table 3: Table summarizing the effects of countermeasures

| Protection scheme | A1 | A2 | A3 (new) |
|---|---|---|---|
| | l.3 | l.31 | l.25 |
| Normal Wookey | ✓ | ✓ | ✓ |
| Prior patch (CM1+CM2+CM3+CM4) | ✗ | ✗ | ✓ |
| Our patch (CM2+CM4+CMA) | ✗ | ✗ | ✗ |

Legend - ✓: attack path found by our tool / ✗: no attack found

**Other Attacks on WooKey.** We were also able to find two other known attacks on Wookey. ***(Attack vector combination)*** The iso8716 library, used in WooKey for secure communication, presents a vulnerability to fault injection which enables a software buffer-overflow in function *SC_get_ATR* [63]. Using an attacker with a single arbitrary data fault, we were able to reproduce this attack; ***(Faulty redundant test)*** Martin et al. [68] shows an incorrect implementation of a redundant test to prevent single test inversion faults in the *loader_set_state* function. We reproduce this result.

## 9 Discussion

**Fault Models.** Our current approach does not support advanced *control* faults such as instruction corruption or instruction skip. Instruction corruption is out of scope as it permanently changes an instruction, while we modify computation results. It is related to self-modification, a notoriously difficult point to address in adversarial binary-level code analysis [17,77]. Instruction skip (or other arbitrary control jumps) could be modeled by local modification of the program counter, yet at the price of a huge path explosion. Also, regarding micro-architectural attacks, modeling Spectre attacks is difficult due to the speculative windows mechanism and its associated rollback.

**Other Formal Methods.** While in the paper we focus on symbolic execution, we believe the main optimization ideas developed here can be used with other formal techniques, e.g. Bounded Model Checking [29,31], Abstract Interpretation [34] or CEGAR [30]. Note that for each of them, fault injection may result either in path explosion or precision loss. Still, our forkless encoding should be able to help at least all approaches based to some extent on path unrolling.

**Other Properties.** The forkless encoding can surely benefit other classes of properties to be achieved by the attacker, especially those known to be supported by (extensions of) symbolic execution, for example: trace properties such as use-after-free, k-hyperreachability properties (secret leakage, privacy leakage, violation of constant-time, etc.) [36], the recent robust reachability proposal [48] for replicable bugs, etc. Our formalism itself is quite generic and can accommodate a wide range of properties, as we mainly keep the property unchanged but modify the underlying transition system. We could for example imagine an attacker willing to activate a non-terminating execution (denial of service).

**Forkless Encoding and Instrumentation.** Several prior works use code-level instrumentation [68] or LLVM-level instrumentation [63,65,76] in order to leverage standard program analyzers as is. The forkless encoding we propose can also be used this way, for more flexibility but without additional optimizations. Actually, we performed some experiments with Klee and a C-level forkless instrumentation, and do observe significant improvement over forking instrumentation.

## 10 Related Work

**SWiFI.** Prior work in SWiFI has already been discussed in Section 3. All methods in this domain consider low-level formalism: C [28,68], LLVM [63,76], binary [15,20,25,50]. Half of the techniques rely on the mutant approach [25,28,49, 50,79], and the other half relies on forking [15,20,63,76]. While most approaches target attack finding (with symbolic execution and bounded model-checking), some do aim at full verification [79], especially with deductive verification [28,68]. Very few works consider multi-faults [63,68,76]. Interestingly, Lacombe et al. [63] propose a static way of reducing injection points on C programs, that is complementary to our own method – still, static analysis at binary-level is known to be hard. Note that a few methods do consider instruction skips [20, 49, 50], yet with path explosion issues.

**Robustness Analysis.** SWiFI is also used for robustness evaluation [32, 56, 64, 65, 72, 74, 88, 90], in order to verify the correct behavior of error handling mechanisms. They rely also on forking or mutant techniques. The fault models are similar to hardware fault injection, yet multi-fault is not really an issue there, as faults are supposed to originate from safety issues (e.g. cosmic rays) and have no reason to accumulate unreasonably.

**Formalizations and Fault Models.** While it is common in the field of automated formal verification of cryptographic protocols to consider models of attackers (typically, extensions of the "Dolev-Yao" model) – either by specifying what the attackers can do [2] or what they cannot do [7], only very few formalizations of software-level attacker capabilities have been proposed so far. In software security, control-flow integrity attacks have been categorized by the capability an attacker needs [21], but these efforts have been restricted to manual reasoning. Interestingly, Given-Wilson et al. [51] propose a formalization of fault injection using Turing machines, but to our knowledge, no algorithm has been built for it. Also, Fournet et al. [46] propose a type system for program-level non-interference, taking into account an active adversary modeled as adversarial components able to perform any action at certain steps of the program.

**Mutation Testing.** Sometimes called software fault injection, mutation testing [33, 75] aims to generate a comprehensive test suite by building test cases discriminating various mutants of a program, and is recognized as a very powerful testing criterion. As it focuses on coverage, mutant explosion cannot be avoided. Dedicated SE techniques [8, 11, 67, 73] have been designed.

## 11 Conclusion

We formalize the concept of adversarial reachability, extending standard reachability to include the presence of an advanced attacker in program analysis, and we propose a dedicated symbolic algorithm for adversarial reachability, integrating a novel forkless encoding of faults together with dedicated optimizations. Our technique is shown to significantly reduce the number of paths to explore, and scales up to 10 faults on a standard SWiFI benchmark, where prior forking attempts timeout for 3 faults. Also, we show that our method scale to realistic size examples, such as the WooKey project where we have been able to replay known fault attacks and to even find a vulnerability not mentioned in a recently proposed countermeasure patch.

## References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Transactions on Information and System Security (TISSEC) **13**(1), 1–40 (2009)
2. Abadi, M., Cortier, V.: Deciding knowledge in security protocols under equational theories. In: International Colloquium on Automata, Languages, and Programming. pp. 46–58. Springer (2004)
3. Akhunzada, A., Sookhak, M., Anuar, N.B., Gani, A., Ahmed, E., Shiraz, M., Furnell, S., Hayat, A., Khan, M.K.: Man-at-the-end attacks: Analysis, taxonomy, human aspects, motivation and future directions. Journal of Network and Computer Applications **48**, 44–57 (2015)
4. Anceau, S., Bleuet, P., Clédière, J., Maingault, L., Rainard, J.l., Tucoulou, R.: Nanofocused X-ray beam to reprogram secure circuits. In: International Conference on Cryptographic Hardware and Embedded Systems. pp. 175–188. Springer (2017)
5. ANSSI, Amossys, EDSI, LETI, Lexfo, Oppida, Quarkslab, SERMA, Synacktiv, Thales, Labs, T.: Inter-cesti: Methodological and technical feedbacks on hardware devices evaluations. In: SSTIC 2020, Symposium sur la sécurité des technologies de l'information et des communications (2020)
6. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In: International Conference on Integrated Formal Methods. pp. 1–20. Springer (2004)
7. Bana, G., Comon-Lundh, H.: A computationally complete symbolic attacker for equivalence properties. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 609–620 (2014)
8. Bardin, S., Chebaro, O., Delahaye, M., Kosmatov, N.: An all-in-one toolkit for automated white-box testing. In: International Conference on Tests and Proofs. pp. 53–60. Springer (2014)
9. Bardin, S., David, R., Marion, J.Y.: Backward-bounded dse: targeting infeasibility questions on obfuscated codes. In: 2017 IEEE Symposium on Security and Privacy (SP). pp. 633–651. IEEE (2017)
10. Bardin, S., Herrmann, P., Leroux, J., Ly, O., Tabary, R., Vincent, A.: The bincoa framework for binary code analysis. In: International Conference on Computer Aided Verification. pp. 165–170. Springer (2011)
11. Bardin, S., Kosmatov, N., Cheynier, F.: Efficient leveraging of symbolic execution to advanced coverage criteria. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation. pp. 173–182. IEEE (2014)
12. Barrett, C., Tinelli, C.: Satisfiability modulo theories. In: Handbook of model checking, pp. 305–343. Springer (2018)
13. Barthe, G., Dupressoir, F., Fouque, P.A., Grégoire, B., Zapalowicz, J.C.: Synthesis of fault attacks on cryptographic implementations. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 1016–1027 (2014)
14. Benadjila, R., Renard, M., Trebuchet, P., Thierry, P., Michelizza, A., Lefaure, J.: Wookey: Usb devices strike back. Proceedings of SSTIC (2018)
15. Berthier, M., Bringer, J., Chabanne, H., Le, T.H., Rivière, L., Servant, V.: Idea: embedded fault injection simulator on smartcard. In: International Symposium on Engineering Secure Software and Systems. pp. 222–229. Springer (2014)
16. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults. In: International conference on the theory and applications of cryptographic techniques. pp. 37–51. Springer (1997)

17. Bonfante, G., Fernandez, J., Marion, J.Y., Rouxel, B., Sabatier, F., Thierry, A.: Codisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 745–756 (2015)
18. Bounimova, E., Godefroid, P., Molnar, D.: Billions and billions of constraints: Whitebox fuzz testing in production. In: 2013 35th International Conference on Software Engineering (ICSE). pp. 122–131. IEEE (2013)
19. Bozzato, C., Focardi, R., Palmarini, F.: Shaping the glitch: optimizing voltage fault injection attacks. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 199–224 (2019)
20. Bréjon, J.B., Heydemann, K., Encrenaz, E., Meunier, Q., Vu, S.T.: Fault attack vulnerability assessment of binary code. In: Proceedings of the Sixth Workshop on Cryptography and Security in Computing Systems. pp. 13–18 (2019)
21. Burow, N., Carr, S.A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., Payer, M.: Control-flow integrity: Precision, security, and performance. ACM Computing Surveys (CSUR) **50**(1), 1–33 (2017)
22. Cadar, C., Dunbar, D., Engler, D.R., et al.: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. vol. 8, pp. 209–224 (2008)
23. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: Automatically generating inputs of death. ACM Transactions on Information and System Security (TISSEC) **12**(2), 1–38 (2008)
24. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Communications of the ACM **56**(2), 82–90 (2013)
25. Carré, S., Desjardins, M., Facon, A., Guilley, S.: Openssl bellcore's protection helps fault attack. In: 2018 21st Euromicro Conference on Digital System Design (DSD). pp. 500–507. IEEE (2018)
26. Cervesato, I.: The dolev-yao intruder is the most powerful attacker. In: 16th Annual Symposium on Logic in Computer Science—LICS. vol. 1, pp. 1–2. Citeseer (2001)
27. Chauvet, H., de Ferrière, F., Bizet, T.: Software fault injection for secswift qualification (2021), https://jaif.io/2021/media/JAIF2021%20-%20deFerriere.pdf
28. Christofi, M., Chetali, B., Goubin, L.: Formal verification of an implementation of crt-rsa vigilant's algorithm. In: PROOFS workshop: pre-proceedings. vol. 28 (2013)
29. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Form. Methods Syst. Des. (2001)
30. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM (2003)
31. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ansi-c programs. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 168–176. Springer (2004)
32. Cotroneo, D., De Simone, L., Liguori, P., Natella, R.: Profipy: Programmable software fault injection as-a-service. In: 2020 50th annual IEEE/IFIP international conference on dependable systems and networks (DSN). pp. 364–372. IEEE (2020)
33. Cotroneo, D., Natella, R.: Fault injection for software certification. IEEE Security & Privacy **11**(4), 38–45 (2013)
34. Cousot, P.: Abstract interpretation. ACM Computing Surveys (CSUR) **28**(2), 324–328 (1996)
35. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The astreé analyzer. In: Programming Languages and Systems (2005)

36. Daniel, L.A., Bardin, S., Rezk, T.: Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 1021–1038. IEEE (2020)
37. Daniel, L.A., Bardin, S., Rezk, T.: Hunting the haunter-efficient relational symbolic execution for spectre with haunted relse. In: NDSS (2021)
38. David, R., Bardin, S., Ta, T.D., Mounier, L., Feist, J., Potet, M.L., Marion, J.Y.: Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis. In: SANER (2016)
39. Dehbaoui, A., Dutertre, J.M., Robisson, B., Tria, A.: Electromagnetic transient faults injection on a hardware and a software implementations of AES. In: 2012 Workshop on Fault Diagnosis and Tolerance in Cryptography. pp. 7–15. IEEE (2012)
40. Djoudi, A., Bardin, S.: Binsec: Binary code analysis with low-level regions. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 212–217. Springer (2015)
41. Dullien, T.: Weird machines, exploitability, and provable unexploitability. IEEE Transactions on Emerging Topics in Computing **8**(2), 391–403 (2017)
42. Dureuil, L., Petiot, G., Potet, M.L., Le, T.H., Crohen, A., Choudens, P.d.: Fissc: A fault injection and simulation secure collection. In: International Conference on Computer Safety, Reliability, and Security. pp. 3–11. Springer (2016)
43. Facebook: Infer static analyzer. https://fbinfer.com/
44. Farinier, B., David, R., Bardin, S., Lemerre, M.: Arrays made simpler: An efficient, scalable and thorough preprocessing. In: LPAR. pp. 363–380 (2018)
45. de Ferrière, F.: Software countermeausres in the llvm risc-v compiler (2021), https://open-src-soc.org/2021-03/media/slides/3rd-RISC-V-Meeting-2021-03-30-15h00-Fran%C3%A7ois-de-Ferri%C3%A8re.pdf
46. Fournet, C., Rezk, T.: Cryptographically sound implementations for typed information-flow security. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008. ACM (2008)
47. Gangolli, A., Mahmoud, Q.H., Azim, A.: A systematic review of fault injection attacks on iot systems. Electronics **11**(13), 2023 (2022)
48. Girol, G., Farinier, B., Bardin, S.: Not all bugs are created equal, but robust reachability can tell the difference. In: International Conference on Computer Aided Verification. pp. 669–693. Springer (2021)
49. Given-Wilson, T., Jafri, N., Lanet, J.L., Legay, A.: An automated formal process for detecting fault injection vulnerabilities in binaries and case study on present. In: 2017 IEEE Trustcom/BigDataSE/ICESS. pp. 293–300. IEEE (2017)
50. Given-Wilson, T., Jafri, N., Legay, A.: Combined software and hardware fault injection vulnerability detection. Innovations in Systems and Software Engineering **16**(2), 101–120 (2020)
51. Given-Wilson, T., Legay, A.: Formalising fault injection and countermeasures. In: Proceedings of the 15th International Conference on Availability, Reliability and Security. pp. 1–11 (2020)
52. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. pp. 213–223 (2005)
53. Godefroid, P., Levin, M.Y., Molnar, D.: Sage: whitebox fuzzing for security testing. Communications of the ACM **55**(3), 40–44 (2012)

54. Goyal, B., Sitaraman, S., Venkatesan, S.: A unified approach to detect binding based race condition attacks. In: Int'l Workshop on Cryptology & Network Security (CANS). p. 16 (2003)
55. Gravellier, J., Dutertre, J.M., Teglia, Y., Moundi, P.L.: Faultline: Software-based fault injection on memory transfers. In: 2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). pp. 46–55. IEEE (2021)
56. Hari, S.K.S., Tsai, T., Stephenson, M., Keckler, S.W., Emer, J.: Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation. In: 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). pp. 249–258. IEEE (2017)
57. Van den Herrewegen, J., Oswald, D., Garcia, F.D., Temeiza, Q.: Fill your boots: Enhanced embedded bootloader exploits via fault injection and binary analysis. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 56–81 (2021)
58. Karaklajić, D., Schmidt, J.M., Verbauwhede, I.: Hardware designer's guide to fault attacks. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **21**(12), 2295–2306 (2013)
59. Kim, C.H., Quisquater, J.J.: Fault attacks for CRT based RSA: New attacks, new results, and new countermeasures. In: IFIP International Workshop on Information Security Theory and Practices. pp. 215–228. Springer (2007)
60. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: A software analysis perspective. Form. Asp. Comput. (2015)
61. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., et al.: Spectre attacks: Exploiting speculative execution. In: SP (2019)
62. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., et al.: Spectre attacks: Exploiting speculative execution. Communications of the ACM **63**(7), 93–101 (2020)
63. Lacombe, G., Feliot, D., Boespflug, E., Potet, M.L.: Combining static analysis and dynamic symbolic execution in a toolchain to detect fault injection vulnerabilities. In: PROOFS WORKSHOP (SECURITY PROOFS FOR EMBEDDED SYSTEMS) (2021)
64. Larsson, D., Hähnle, R.: Symbolic fault injection. In: International Verification Workshop (VERIFY). vol. 259, pp. 85–103. Citeseer (2007)
65. Le, H.M., Herdt, V., Große, D., Drechsler, R.: Resilience evaluation via symbolic fault injection on intermediate code. In: 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 845–850. IEEE (2018)
66. Le, Q.L., Raad, A., Villard, J., Berdine, J., Dreyer, D., O'Hearn, P.W.: Finding real bugs in big programs with incorrectness logic. Proceedings of the ACM on Programming Languages **6**(OOPSLA1), 1–27 (2022)
67. Marcozzi, M., Bardin, S., Kosmatov, N., Papadakis, M., Prevosto, V., Correnson, L.: Time to clean your test objectives. In: Proceedings of the 40th International Conference on Software Engineering. pp. 456–467 (2018)
68. Martin, T., Kosmatov, N., Prevosto, V.: Verifying redundant-check based countermeasures: a case study. In: Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing. pp. 1849–1852 (2022)
69. Murdock, K., Oswald, D., Garcia, F.D., Van Bulck, J., Gruss, D., Piessens, F.: Plundervolt: Software-based fault injection attacks against intel sgx. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 1466–1482. IEEE (2020)

70. Mutlu, O., Kim, J.S.: Rowhammer: A retrospective. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **39**(8), 1555–1571 (2019)
71. Niemetz, A., Preiner, M.: Bitwuzla at the SMT-COMP 2020. CoRR **abs/2006.01621** (2020), https://arxiv.org/abs/2006.01621
72. Palazzi, L., Li, G., Fang, B., Pattabiraman, K.: A tale of two injectors: End-to-end comparison of ir-level and assembly-level fault injection. In: 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE). pp. 151–162. IEEE (2019)
73. Papadakis, M., Malevris, N.: Automatic mutation test case generation via dynamic symbolic execution. In: 2010 IEEE 21st International Symposium on Software Reliability Engineering. pp. 121–130. IEEE (2010)
74. Pattabiraman, K., Nakka, N., Kalbarczyk, Z., Iyer, R.: Symplfied: Symbolic program-level fault injection and error detection framework. In: 2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN). pp. 472–481. IEEE (2008)
75. Petrovic, G., Ivankovic, M., Kurtz, B., Ammann, P., Just, R.: An industrial application of mutation testing: Lessons, challenges, and research directions. In: 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 47–53. IEEE (2018)
76. 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: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation. pp. 213–222. IEEE (2014)
77. Preda, M.D., Giacobazzi, R., Debray, S., Coogan, K., Townsend, G.M.: Modelling metamorphism by abstract interpretation. In: International Static Analysis Symposium. pp. 218–235. Springer (2010)
78. Puys, M., Riviere, L., Bringer, J., Le, T.h.: High-level simulation for multiple fault injection evaluation. In: Data Privacy Management, Autonomous Spontaneous Security, and Security Assurance, pp. 293–308. Springer (2014)
79. Rauzy, P., Guilley, S.: A formal proof of countermeasures against fault injection attacks on crt-rsa. Journal of Cryptographic Engineering **4**(3), 173–185 (2014)
80. Recoules, F., Bardin, S., Bonichon, R., Lemerre, M., Mounier, L., Potet, M.L.: Interface compliance of inline assembly: Automatically check, patch and refine. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 1236–1247. IEEE (2021)
81. Recoules, F., Bardin, S., Bonichon, R., Mounier, L., Potet, M.L.: Get rid of inline assembly through verification-oriented lifting. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 577–589. IEEE (2019)
82. Richter-Brockmann, J., Sasdrich, P., Guneysu, T.: Revisiting fault adversary models–hardware faults in theory and practice. IEEE Transactions on Computers (2022)
83. Sen, K., Marinov, D., Agha, G.: Cute: A concolic unit testing engine for c. ACM SIGSOFT Software Engineering Notes **30**(5), 263–272 (2005)
84. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In: IEEE Symposium on Security and Privacy (2016)

85. Skorobogatov, S.P., Anderson, R.J.: Optical fault induction attacks. In: International workshop on cryptographic hardware and embedded systems. pp. 2–12. Springer (2002)
86. Tang, A., Sethumadhavan, S., Stolfo, S.: {CLKSCREW}: Exposing the perils of {Security-Oblivious} energy management. In: 26th USENIX Security Symposium (USENIX Security 17). pp. 1057–1074 (2017)
87. Van Bulck, J., Moghimi, D., Schwarz, M., Lippi, M., Minkin, M., Genkin, D., Yarom, Y., Sunar, B., Gruss, D., Piessens, F.: Lvi: Hijacking transient execution through microarchitectural load value injection. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 54–72. IEEE (2020)
88. Winter, S., Tretter, M., Sattler, B., Suri, N.: simfi: From single to simultaneous software fault injections. In: 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 1–12. IEEE (2013)
89. https://github.com/wookey-project, accessed july 2021
90. Zavalyshyn, I., Given-Wilson, T., Legay, A., Sadre, R., Riviere, E.: Chaos duck: A tool for automatic iot software fault-tolerance analysis. In: 2021 40th International Symposium on Reliable Distributed Systems (SRDS). pp. 46–55. IEEE (2021)