Efficient Symbolic Execution of Software under Fault Attacks

Yuzhou Fang
University of Southern California, Los Angeles, USA

Chenyu Zhou
University of Southern California, Los Angeles, USA

Jingbo Wang
Purdue University, West Lafayette, USA

Chao Wang
University of Southern California, Los Angeles, USA

Abstract

We propose a symbolic execution method for analyzing the safety of software under fault attacks both accurately and efficiently. Fault attacks leverage physically injected hardware faults in an embedded system to break the safety of a software program. While there are existing methods for analyzing the impact of maliciously injected hardware faults on the embedded software, they suffer from inaccurate fault modeling and inefficient fault analysis. To overcome these limitations, we propose two novel techniques. First, we propose a new fault modeling technique that leverages automated program transformation to add symbolic variables to the original program, to accurately model the new program behavior induced by the injected faults. This new fault modeling approach has two advantages over existing techniques: (a) the fault-induced program behavior is closely related to what attackers exploit in practice and (b) the automatically transformed program may be analyzed by any downstream fault analysis algorithm. Second, we propose an efficient symbolic execution algorithm that is designed specifically for conducting fault analysis on the transformed program. It leverages two pruning techniques to mitigate path explosion, which is the main performance bottleneck of symbolic execution in general and, in this particular application, is exacerbated by the additional fault-induced program behavior. We have implemented the proposed method and evaluated it on a variety of benchmark programs. The experimental results show that our method significantly outperforms the state-of-the-art techniques. Specifically, our method not only drastically reduces the overall running time of symbolic execution but also retains its error detection capabilities. Compared to the current state-of-the-art, it is able to detect previously-missed safety violations and at the same time avoid bogus violations. Furthermore, compared to the baseline algorithm, our optimized symbolic execution algorithm can be orders-of-magnitude faster.

2012 ACM Subject Classification Theory of computation → Program verification; Software and its engineering → Software testing and debugging

Keywords and phrases Symbolic Execution, Safety Verification, Fault Attack, Embedded Software

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2026.16

Funding This work was supported in part by the NSF grant CCF-2220345.

1 Introduction

Fault injection is a technique that attackers often use in the context of embedded systems to violate the safety property of a software program by injecting hardware faults into the underlying CPU hardware. In practice, this has been accomplished using various physical mechanisms including clock glitching [29], voltage glitching [5], electromagnetic (EM) waves [30], and laser beams [35]. Prior studies [1, 42, 14, 25] have shown that such maliciously injected hardware faults can lead to abnormal behaviors in a software program

that runs on top of the hardware. Furthermore, these abnormal program behaviors are often statistically predictable and, in some cases, are even somewhat controllable by attackers, e.g., in the form of targeted instruction skipping, where an if-statement that guards access to some sensitive data is skipped, thus leading to safety violations.

Conventional techniques for assessing the potential impact of such hardware fault injection attacks on the embedded software rely primarily on simulation techniques. Broadly speaking, these techniques include hardware simulation, software simulation, and hardware-software co-simulation. However, these simulation-based techniques tend to be slow. For example, hardware simulation requires the availability of a detailed model of the CPU, and the modeling of fault injection/propagation, often at the register-transfer level (RTL). RTL simulation is extremely slow. Furthermore, all simulation techniques suffer from limited behavior coverage, because they run the software program on the hardware model for one specific program input at a time. In contrast, symbolic execution-based techniques *have the potential* to be more efficient by simultaneously covering all program inputs and execution paths, including all of the fault-induced execution paths, provided that the fault impact is modeled accurately. However, existing symbolic execution-based techniques for fault analysis tend to suffer from inaccurate fault modeling and inefficient fault analysis.

Fault Modeling. Current state-of-the-art techniques for modeling the impact of hardware faults on a software program have two limitations. First, the fault model is not realistic in the sense that it does not model what actually happens at the hardware level. Second, the fault model is not accurate in the sense that it may miss fault-induced program behaviors that can show up in practice. For example, to model the impact of a hardware fault on the program's control flow, an existing technique is called *test inversion* [9]. In general, a program may implement a branching statement, such as an if-else statement, using a pair of comparison and jump operations. Specifically, the comparison sets a CPU flag and the jump operation tests the CPU flag, based on which the control is transferred to different program locations (see Section 2.1.1 for details). Test inversion assumes that, right before testing, the CPU flag may be inverted from *true* to $false$ or vice versa, depending on whether a hardware fault is injected at that moment. Thus, the impact of the hardware fault is modeled by flipping the two conditions guarding the two branches of an if-else statement.

While this may appear to be a reasonable approach at first glance, since it does capture some of the fault-induced execution paths, the model does not reflect what actually happens inside the CPU hardware. For example, it would be extremely difficult in practice for an attacker to flip a CPU flag in a predictable and targeted manner. Therefore, it is not how a real attack is carried out. Instead, what typically happens in practice is that a jump instruction associated with the if-else statement is turned into a *nop* (e.g., due to bit-flips in the jump instruction's encoding, which often occur during faulty instruction fetching). As a result, the jump instruction is skipped. We will show later in this paper (both in Section 4 and during the experimental evaluation in Section 6) that instruction skipping is significantly more accurate than test inversion: it can detect realistic safety violations that otherwise would have been overlooked.

Fault Analysis. Another limitation of the current state-of-the-art techniques is the inefficiency of their fault analysis algorithms. Since the impact of a hardware fault may show up at any moment during the execution of a software program, all possible fault-induced execution paths must be analyzed to check if any of them may lead to the violation of a safety or security property. While symbolic execution is capable of exploring all program paths for all program inputs, including the fault-induced execution paths that otherwise

would not have been feasible, it is also well known that symbolic execution suffers from the *path explosion* problem. When hardware faults are injected, the path explosion problem is further exacerbated. While there are existing techniques for mitigating the path explosion problem, they are not designed specifically for fault analysis.

The transformed program subjected to a fault analysis has some unique characteristics. First, the symbolic variables that we introduce for modeling the impact of hardware faults are significantly different from ordinary program variables; treating them in the same manner during symbolic execution can make it difficult for pruning algorithms to identify redundant execution paths. Furthermore, there can be subtle interactions between the two types of variables. Furthermore, existing methods often impose a bound on the maximum number of faults that an execution can activate using a technique called *fault saturation* [9], which avoids exploring paths where the number of faults exceeds the bound. While this technique can reduce the number of explored paths, by itself, it is not sufficient for taming the often astronomically large number of execution paths. Instead, its tight integration with techniques for pruning redundant paths must be considered.

Proposed Method. To overcome the two limitations mentioned above, we propose a new symbolic execution method to analyze the impact of hardware faults on a software program both accurately and efficiently. Our method has two innovations. The first one is a program transformation technique for accurately modeling the fault-induced program behavior. The second one is a symbolic execution algorithm for efficient fault analysis, equipped with a combination of fault saturation and redundancy pruning.

Unlike existing methods where fault modeling is part of the fault analysis procedure, our method implements fault modeling as a standalone program transformation. It leverages a compiler to automatically insert auxiliary variables into the program at the intermediate representation (IR) level, to model the impact of faults on the program's control flow. Our fault modeling technique is realistic in the sense that it faithfully implements the observations reported by prior empirical studies [42, 14, 25], which show that the primary source of fault attacks in practice is due to the fact that hardware faults can lead to jump instructions being skipped. Thus, for each jump instruction in the original program, we add some additional code to create an auxiliary variable at run time, called the *fault flag*. Details will be presented in Section 4. For now, it suffices to understand that, if the jump instruction is inside a loop, each time the jump instruction is executed, a separate *fault flag* is created. When all of the *fault flags* are disabled, for example, the transformed program behaves exactly the same as the original program. However, when some *fault flags* are enabled, the transformed program may have additional behaviors; they correspond to some jump instructions being skipped during program execution.

Our second innovation is an efficient symbolic execution algorithm for conducting fault analysis. Its efficiency comes from a new technique for mitigating the path explosion problem. The technique combines a path summary-based analysis with fault saturation to identify and then avoid redundant execution paths. In this context, our main observation is that multiple execution paths often share the same suffix. While such sharing is common among fault-free execution paths, it is even more common among fault-induced execution paths, as well as between fault-free execution paths and fault-induced execution paths. In such cases, the shared suffix may only need to be symbolically executed once, instead of being symbolically executed again along different paths (prefixes).

Specifically, the weakest precondition (WP [8]) computed along a previously-explored suffix can serve as a sound and effective summary of the partial path (the suffix). We leverage this path summary to define a sufficient condition, under which the symbolic execution of

this shared suffix will not lead to any previously-unseen violations. Thus, when the shared suffix is encountered again, e.g., following the symbolic execution of another program path (another prefix), we can safely skip the symbolic execution of the shared suffix, because it is deemed redundant. In other words, skipping the partial path would not negatively affect the symbolic execution algorithm's ability to detect safety violations.

Evaluation. To find out whether the proposed method is effective in practice, we have implemented the method in a software tool by leveraging three existing tools: the LLVM compiler [23], the KLEE symbolic execution engine [4], and the Z3 SMT solver [6]. The software tool is designed to analyze C programs where the safety property is specified using an assertion, which is expected to hold for all program paths and program inputs. Alternatively, the safety property may be regarded as a program location named ERROR, which should never be reached. Given such a C program as input, our tool leverages the LLVM compiler's existing capability to compile C code to LLVM bitcode, and then applies our fault modeling to LLVM bitcode. Next, it leverages our extension of KLEE to conduct efficient symbolic execution, to detect violations of the safety property represented as an assertion embedded in the program. The tool also leverages Z3 to perform the symbolic analysis needed for identifying and pruning the redundant execution paths.

For experimental evaluation, we have implemented both our method and a state-of-the-art method from [9] in the same tool. We have applied both methods to 112 benchmark programs. These benchmark programs come from two sources. The first source is the VerifyPIN benchmark suite used by the existing method [9]. The second source is the *array-crafted*, *bitvector*, and *bitvector-loops* categories of the SV-COMP suite [3]. We have also conducted the ablation study by applying our method with and without the new redundancy pruning technique. The experimental results show that our method significantly outperforms the current state-of-the-art. Specifically, our method can not only drastically reduce the running time taken by fault analysis, but also can detect previously-missed safety violations. Compared to the baseline symbolic execution algorithm, our new symbolic execution method, augmented with redundancy pruning, can be orders of magnitude faster.

To summarize, this paper makes the following technical contributions:

- We propose a symbolic execution method for accurately and efficiently analyzing the safety property of a software program running on a hardware platform that is subjected to fault attacks.
- We design an automated program transformation to add auxiliary symbolic variables and encode fault-induced control flows, to allow the transformed program to model fault-induced program behaviors.
- We design a redundancy pruning technique, which leverages both fault saturation and a weakest precondition based path summary to identify and eliminate more redundant paths during symbolic execution.
- We implement the method and demonstrate its superiority over the current state-of-the-art on two sets of widely used benchmark programs.

The remainder of this paper is organized as follows. We provide the technical background in Section 2. Then, we present the top-level procedure of our symbolic execution method in Section 3. This is followed by our fault modeling technique in Section 4 and our new redundancy pruning technique in Section 5. After that, we present the experimental results in Section 6 and review the related work in Section 7. Finally, we give our conclusion in Section 8.

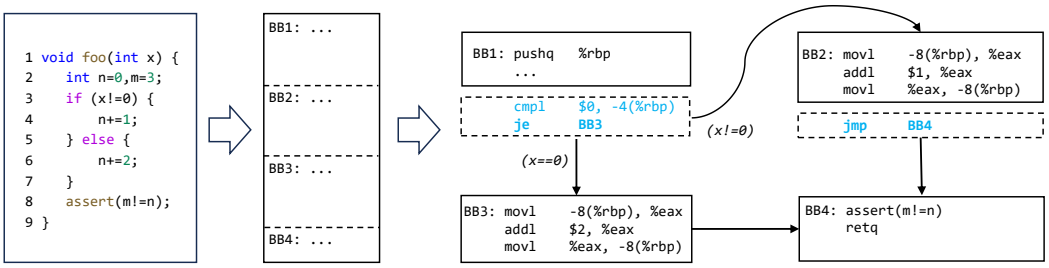


Figure 1 An example C program (left) and its x86 assembly code shown as a sequence of basic blocks (middle) and the CFG (right). Note that the assertion at Line 8 always holds.

2 Background

We first use an example software program to illustrate the limitations of existing methods, and then present the threat model of hardware fault attacks manifested on embedded software.

2.1 The Example Program

Fig. 1 shows a victim software program, where the C code is on the left-hand side. Depending on the value of input variable `x`, the function `foo(x)` may execute either of the two branches in the if-else statement. For all program paths and program inputs, the assertion at Line 8 always holds. In the middle are the four basic blocks shown in the program's x86-64 assembly code, produced by the Clang/LLVM compiler. They are also the same basic blocks `BB1-BB4` shown in the control flow graph (CFG) on the right-hand side of Fig. 1.

Specifically, the conditional jump instruction (`je BB3`) corresponds to `if(x!=0)` in the C program. Depending on the value of `x`, the CPU will execute either `BB2` or `BB3`. The unconditional jump instruction (`jmp BB4`) redirects the control from `BB2` to `BB4`.

2.1.1 Fault Modeling

The example program in Fig. 1 illustrates the inaccuracy of fault modeling in the current state-of-the-art [9], which relies on *test inversion*. That is, the fault is modeled by inverting the CPU flag used by the conditional jump instruction. When the fault is injected at the end of `BB1`, in particular, the resulting CPU flag set by the comparison instruction `cmpl $0,-4(%rbp)` will be inverted, which has the effect of flipping the if-else branches in the C program. However, this does not lead to the violation of the assertion in Line 8, because as we have mentioned earlier, the assertion always holds, regardless of which of the two branches is executed.

In contrast, our fault modeling allows the assertion in Line 8 to be violated, and thus matches what may actually happen in practice (the ground truth). This is because our technique relies on *branch instruction skipping* to model the fault. When the fault is injected at the end of `BB2`, it turns the jump instruction `jmp BB4` into a null operation (*nop*). Consequently, the program enters `BB3` right after `BB2`. This has the effect of executing *both branches* of the if-else statement. As a result of our fault modeling, the value of n becomes 3 at Line 8, which violates the assertion condition `(m!=n)`.

Note that the violation would not have been detected by *test inversion*, since it relies on inverting the CPU flag, but the unconditional jump instruction `jmp BB4` does not check the CPU flag.

2.1.2 Path Pruning

The example program in Fig. 1 also illustrates the inefficiency of existing fault analysis algorithms. For the control flow graph (CFG) on the right-hand side of Fig. 1, there will be five distinct execution paths using our fault modeling with transformation (the five paths will be shown in Fig. 4). Thus, a baseline symbolic execution algorithm would explore all five paths. Some existing methods rely on *fault saturation* [9] to reduce the number of explored paths, but the technique is not always effective. In fact, it is largely ineffective for the example program shown in Fig. 1: even with *fault saturation*, symbolic execution would still completely execute four paths if one fault is allowed, while only one path (Path#5) that requires two faults is partially pruned.

In contrast, our symbolic execution algorithm will explore only two complete paths (they are the first two paths shown in Fig. 4). One path has an assertion violation, and the other path does not. The remaining three paths are only partially explored before our method identifies them as *redundant* with respect to the already explored paths, and thus skips the remaining paths (suffixes).

While in this small example program, a reduction from five paths to two paths (together with three partially explored paths) may seem modest, the reduction will become more significant on larger programs. This is because the number of paths in a program tends to grow exponentially as the size of the program increases. During the experimental evaluation in Section 6, we will show that, in practice, the reduction can be as high as several orders of magnitude.

2.2 The Threat Model

Regarding hardware fault attacks manifested on embedded software, the threat model includes the goal and the capability of the adversary, as well as the fault-induced behavior of the victim software program.

2.2.1 The Victim Software Program

Let P be the victim software program and ϕ be a safety property of program P, represented as an assertion statement embedded in P. Furthermore, during a fault-free execution of the program P, the property ϕ is expected to hold for all program paths and for all program inputs.

2.2.2 The Goal and Capability of the Adversary

The goal of the adversary is to violate the property ϕ by physically injecting faults into the underlying CPU hardware, thus changing the victim program's behavior. Specifically, the goal is to violate the safety property ϕ. While faults may be injected in various ways according to the literature [5, 29, 35], due to physical limitations, there is often a bound (β) on the maximum number of faults that can be injected during each execution of the victim program. In this paper, the bound β is called the *fault budget*.

2.2.3 Fault-Induced Program Behavior

Although maliciously injected hardware faults may lead to various abnormal behaviors in the victim software program, not all of them are predictable or controllable. If they are not predictable and controllable, they will not be easily exploited and thus will not be practically

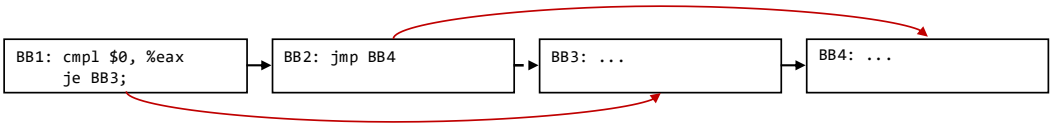


Figure 2 The control flow of an x86 instruction sequence, where the red arrows indicate the control flow brought by the jump instructions and the dashed arrows indicate the possibly sequential execution under a fault.

useful to an adversary. Furthermore, if the abnormal behavior is a moderate deviation, it is generally more useful than a drastic deviation. For example, if the deviation is so drastic that it even leads to the crash of the victim program (or the crash of the entire computer), it may become ineffective for the purpose of stealing information or secretly gaining control.

With this in mind, prior studies of hardware fault-based security attacks [42, 14, 25] tend to focus more on fault-induced *branch instruction skipping*. In contrast, they do not focus on arbitrary data corruption since such corruption is less predictable or controllable. It is worth noting that (silent) data corruption is a significant problem in large-scale infrastructure systems, high-performance computing applications, and cloud data centers. However, it is out of the scope of this paper.

2.2.4 Branch Instruction Skipping

Due to maliciously injected faults, the CPU hardware may interpret a jump instruction (`jmp`) as a null operation (*nop*). It has the effect of completely skipping the jump instruction. Consider the following sequence of x86 assembly instructions `{BB1:cmpl $0,%eax; je BB3; BB2:jmp BB4; BB3:...; BB4:...}` as shown in Fig. 2. In a fault-free environment, the CPU is expected to execute `BB1;BB3` when `%eax` is equal to 0, and execute `BB1;BB2` otherwise. However, in the presence of hardware faults, either (or both) of the jump instructions (`je BB3` and `jmp BB4`) may be turned into a *nop*, thus leading to new (and abnormal) execution paths.

We call it *branch instruction skipping*. It is a transient error instead of a permanent error. For example, if the jump instruction is inside a loop, it is possible to skip the jump instruction during one iteration but not during another iteration. Furthermore, branch instruction skipping is possible not only for the unconditional jump instruction (`jmp`), but also for conditional jump instructions such as `je` and `jne`. In all these cases, the jump instruction is interpreted as a null operation (*nop*).

3 Our Method

We now present the top-level procedure of our method in this section and highlight how it differs from the baseline symbolic execution algorithm. Detailed algorithms of the subroutines will be presented in the next two sections.

3.1 The Notations

Before presenting the top-level procedure, we define the relevant notations. We consider a victim program P with a set L of program locations, where locations l_{init} and l_{end} are the start and end of the program, respectively. Furthermore, $l_{bad} \in L$ represents the error location. In other words, reaching l_{bad} is a safety violation. This way of representing a safety

Algorithm 1 Symbolic execution with (and without) our redundancy pruning techniques.

```
1: Initialize: Run Explore(s_0) with the initial state stack S ← {s_0}
2: Explore( state s ) {
3:     if ( PruningCondition(s) is satisfied)   //use fault count and path summary to skip paths
4:         return;
5:     S.push(s);
6:     if  (s is a branching point)
7:         foreach  (t ∈ s.branch)
8:             s' ← NextState(s, t)
9:             Explore(s');
10:    else if (s is an internal node)
11:        s' ← NextState(s, t)
12:        Explore(s');
13:    else //end of an execution path
14:        compute a program input;
15:        UpdateSuffixSummary(S);          //summarize the path (suffix)
16:    S.pop();
17: }
18: NextState( state s, event t ) {
19:    ⟨l, pcon, mem⟩ ← the state s
20:    ⟨l, inst, l'⟩ ← the event t
21:    if ( inst is assume(c) )
22:        UpdateFaultCounter(s, t);        //compute fault count
23:        s' ← ⟨l', pcon ∧ c, mem⟩
24:    else if ( inst is assignment v := expr )
25:        s' ← ⟨l', pcon, mem[v ↦ expr]⟩
26:    return s';
27: }
```

violation is without loss of generality, since in a Turing-complete programming language, a safety violation can always be expressed as an embedded assertion. Furthermore, `assert(c)` may be modeled as `if(!c){`l_{bad}`}`.

In the victim program's control flow graph (CFG), the nodes represent locations in L, and the edges represent events. Each event is a tuple $t = (l, inst, l')$ where $l, l' \in L$ and $inst$ is an instruction. An execution path is a sequence of events $t_1, t_2, \ldots, t_n$. If $t_n.l' = l_{bad}$ (the last location is an error location), it means that the execution path ends with a safety violation. In contrast, $t_n.l' = l_{end}$ means that the execution path ends without a safety violation.

The instruction $inst$ may be one of two types: it is either an *assignment* or a *branching statement*. This assumption is also made without loss of generality because any event encountered during an execution may be modeled by instructions of these two types. Given a set V of program variables, an assignment is denoted $v := expr$, where $v \in V$ is the left-hand-side (lhs) variable and $expr$ is the right-hand-side (rhs) expression, defined over the variables in V. A branching statement is denoted $assume(c)$, where c is a conditional expression, also defined over the variables in V. For example, `if(x>5){}else{}` in a C program corresponds to two branching statements in our formalism, denoted by $assume(x > 5)$ and $assume(x \leq 5)$, respectively.

3.2 Baseline Symbolic Execution

Our new symbolic execution algorithm is implemented as an extension of KLEE [4], which is a widely used symbolic execution engine for C/C++ programs built on the intermediate representation (IR) of the popular LLVM compiler [23]. Specifically, if we ignore the highlighted lines (3-4, 15, and 22) in Algorithm 1, then it describes the *baseline* symbolic execution procedure as implemented in KLEE. Thus, in the remainder of this section, we

closely follow the description used by KLEE and other symbolic execution tools [34, 13].

As shown in Line 1 of Algorithm 1, starting from the initial state s_0 of the program P, the recursive function EXPLORE(s_0) performs a depth-first-search (DFS) of the feasible program paths. At each step, EXPLORE(s) picks an instruction *inst* that can be executed at the current state s, computes the next state s' (Lines 8 and 11), and invokes EXPLORE on s' recursively (Lines 9 and 12).

Let the execution path be $\pi = s_0, s_1, \ldots, s_n$. During symbolic execution, π is stored in the state stack S, which in turn is leveraged to explore all execution paths systematically. After the procedure finishes executing an execution path (Lines 13–15), it computes a program input. This is useful because, if the path ends with a safety violation, the program input will be able to reproduce the violation.

The state is a tuple $s = (l, pcon, mem)$ where $pcon$ is the path condition that must be satisfied for the execution to reach location $l \in L$, and mem is the symbolic memory map. That is, $mem[v]$ stores, for each variable $v \in V$, its corresponding value expression. Given the current state s, and more importantly, its program location $s.l$, we know all the information needed to compute the next state s'. This includes which events are enabled at s. These events are represented by the outgoing edges of the node $s.l$ in the program's control flow graph.

Finally, in the subroutine NEXTSTATE(s), there are two cases. For an assignment $v := expr$ (Line 24), the new path condition $pcon$ will remain the same, while the symbolic memory map will be updated such that $mem[v]$ holds $expr$. For a branching statement $assume(c)$ (Line 23), the new path condition will be updated to $pcon \wedge c$ while the new symbolic memory map (mem) will remain the same.

3.3 Our New Symbolic Execution Procedure

If we include the highlighted lines in Algorithm 1, then Algorithm 1 describes our new symbolic execution algorithm, which leverages two important techniques for mitigating the path explosion problem. Specifically, Lines 15 and 22 compute the information needed for identifying redundant executions based on the current execution path π, while Lines 3–4 utilize this information to avoid the redundant executions.

3.3.1 Pruning Based on Fault Saturation

Our first redundancy removal technique relies on the fact that the number of activated faults in any execution path must be less than or equal to the fault budget β. Thus, if at any moment during symbolic execution, the currently executed path (stored in the state stack S) has already activated more than β faults, symbolic execution is terminated. Before applying this technique, however, we must perform a program transformation, denoted $P' \leftarrow$ FAULTMODELING(P, β) where the newly transformed program P' is symbolically executed instead of the original program P.

Details of the program transformation will be presented in Section 4. Here, it suffices to say that, for each branching statement $br \in Br$ in the original program P, we add a new branch br' in the transformed program P', such that br' is controlled by a *fault flag* denoted `bFT`, whose value is either *true* or $false$. We also maintain a *fault counter* denoted `FC`, whose value is increased every time a fault flag such as `bFT` is activated (set to *true*).

Together, these auxiliary variables (`bFT`, `FC` and β) model the aforementioned requirements that (1) the original branch br is skipped (turned to *nop*) if and only if its corresponding

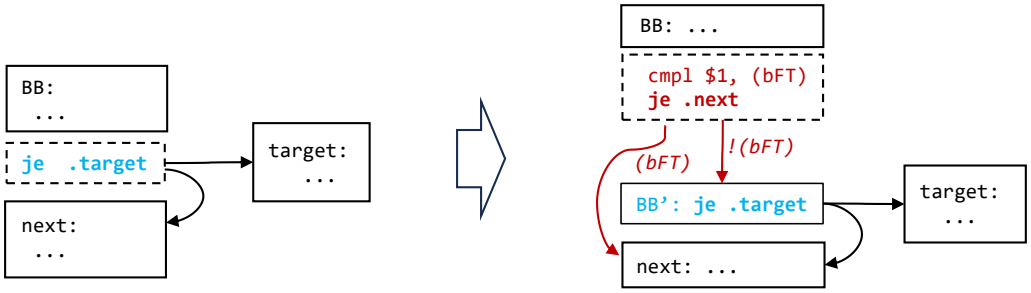


Figure 3 Illustrating how Lines 6-7 of Algorithm 2 work. Given a basic block `BB` shown on the left-hand side, our program transformation algorithm automatically adds the new control flow edge br':= **if**(`bFT`) {**goto** `next`} shown on the right-hand side, right before br := `je .target`.

fault flag `bFT` is set to *true* and (2) an execution path is explored if and only if `FC` $\leq \beta$ holds, meaning that the fault count does not exceed the fault budget.

3.3.2 Pruning Based on Path Summary

Our second redundancy removal technique relies on the fact that an execution path $\pi := \pi_{pre}\pi_{post}$ is the concatenation of a prefix π_{pre} and a suffix π_{post}. Given another path $\pi' := \pi'_{pre}\pi'_{post}$, it is possible for π and π' to have different prefixes ($\pi_{pre} \neq \pi'_{pre}$) but a common suffix ($\pi_{post} = \pi'_{post}$). In such a case, if the common suffix π_{post} has been symbolically executed following π_{pre}, does it need to be symbolically executed again following π'_{pre}? The answer depends on whether symbolically executing π_{post} again can lead to *previously-unexplored* program behavior.

For example, a sufficient condition for answering *no* to the above question is when ($s = s'$). Here, s and s' are the two states reached by the two prefixes π_{pre} and π'_{pre}, respectively. This technique is known as state hashing/merging, but it has limited success in practice, for two reasons. First, since s and s' are symbolic expressions, checking if they are semantically equivalent is expensive, meaning that the overhead may not be compensated by the time saved. More importantly, it is rare for the two states (s and s') to be exactly the same.

In contrast, we use a sufficient condition that is significantly different and more general than state hashing/merging. As shown in Lines 15 and 3 of Algorithm 1, the technique consists of two parts. First, we compute (Line 15) a succinct path summary for the *explored* suffix starting from each program location l, stored in a map entry `WP`[l]. Then, we check if the pruning condition $pcon \rightarrow$ `WP`[l] holds (Line 3). If the pruning condition holds, it means $pcon \subseteq$ `WP`[l]. Since the path condition for the prefix π'_{pre}, denoted $pcon$, is fully covered by the path summary `WP`[l], we can safely avoid executing the common suffix again.

In the next two sections, we present the detailed algorithms of our new techniques for fault modeling and path pruning, respectively.

4 Accurate Fault Modeling via Program Transformation

We propose a program transformation technique for accurately modeling the impact of fault attacks on the victim program. Given the original program P, we construct a new program P' such that P' is semantically equivalent to P in a fault-free execution environment, but has all of the additional behaviors under fault attacks.

4.1 The Fault Modeling Algorithm

Algorithm 2 shows our fault modeling subroutine, which takes a program P as input and returns a new program P' as output. Initially, P' is set to be the same as P. Then, for each branching instruction br in P, we model the impact of *instruction skipping* in two steps. First, we add a *fault flag* `bFT` to the new program P'. Second, we create a new branching instruction br' controlled by `bFT`, with `next` as the target basic block. That is, br' := **if**(`bFT`) {**goto** `next`}. Finally, we add br' to the new program P', at the program location right before the original br, inside the basic block `BB` that holds br.

Algorithm 2 Subroutine $P' \leftarrow$ FaultModeling (P) for our new program transformation.

```
1: Initialize: new program P' ← a copy of program P
2: for each ( branching instruction br ∈ Br in program P ) do
3:     Add a fault flag, denoted bFT, to program P'
4:     BB ← the basic block that holds br
5:     next ← the basic block that immediately follows br in the program code
6:     Create br' := if(bFT) {goto next}
7:     Add br' to program P', right before br inside BB
8: end for
9: return P';
```

Fig. 3 illustrates how Algorithm 2 works, where the basic block of P containing the branch br is shown on the left-hand side. The corresponding basic blocks of the newly transformed program P' are shown on the right-hand side, where the added instructions and edges are highlighted in red. Without loss of generality, in this figure, we only illustrate how the conditional jump instruction is handled. The unconditional jump instruction can be regarded as a special case of conditional jump where the transition from `BB` to `next` is always disabled.

As illustrated in Fig. 3, regardless of whether br is conditional (e.g., `je target`) or unconditional (e.g., `jmp target`), it must have `BB`, `target` and `next`. Here, `BB` is the basic block to which br belongs, `target` is the basic block where br jumps to, and `next` is the basic block immediately following br in the program code. For a concrete example of the `next` basic block, please refer to `BB3` in the middle subfigure of Fig. 1: `BB3` is the basic block immediately following `BB2` in the program code.

4.2 Applied to the Running Example

We now use another example, i.e., the program in Fig. 1, to explain our fault modeling algorithm in detail. Recall that in Fig. 1, the right-hand side shows the control flow graph of the original program P. Accordingly, in Fig. 4, the left-hand side shows the control flow graph of the new program P'.

Specifically, for the conditional branch `je BB3` colored in *blue* in Fig. 4, we add the new branch `je BB2`, which is controlled by the fault flag `bFT1`. When `bFT1`= *true*, meaning that a fault is injected, the control transfers to `BB2` without checking the original condition `(x!=0)`. Similarly, for the unconditional branch `jmp BB4`, we add the new branch `je BB3`, which is controlled by the fault flag `bFT2`. When `bFT2`= *true*, meaning that another fault is injected, the control transfers to `BB3` instead of the original destination `BB4`.

Overall, the new program P' shown on the left-hand side of Fig. 4 has five paths. They are listed on the right-hand side of this figure. In contrast, the original program P has only two paths; they are the first and the second of the five paths listed in Fig. 4.

Advantage over the Prior Work. To summarize, our fault modeling technique is significantly more accurate than the current state-of-the-art [9], which models the impact

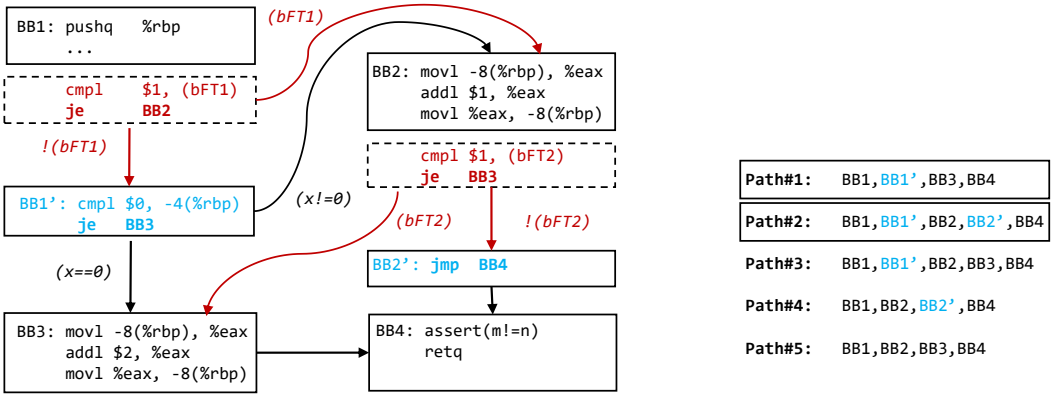


Figure 4 Illustrating the advantages of our new method on the running example: on the left-hand side is the transformed program P' for the original program P in Fig. 1, where fault-induced control flows are highlighted as red instructions and edges. The five paths explored by baseline symbolic execution (without pruning) are shown on the right-hand side; only the first two paths will be fully explored by our new algorithm (with pruning).

of a fault by inverting the test flag. For the running example, in particular, the existing technique would include only 4 of the 5 paths, but would miss the path BB1→BB2→BB3→BB4. Arguably, this is the most interesting path, since it visits both the then-branch and the else-branch of the if-else statement during one execution of the program. While it may be hard to imagine for a novice developer, as we have mentioned earlier, it may occur in practice due to *branch instruction skipping*. The current state-of-the-art technique will miss this violation, as well as many other violations that can be detected by our new method (see the experimental results in Section 6).

We now state the soundness and completeness of our technique.

▶ **Theorem 1.** *Our fault modeling technique as presented in Algorithm 2 is both sound and complete in that the transformed program (1) includes all real violations and (2) does not include bogus violations.*

Proof. The soundness and completeness of our fault modeling technique follow directly from the program transformation steps presented in Algorithm 2. As we have illustrated in Fig. 3, the algorithm guarantees that all existing control flow paths of the program are retained in the transformed program. Furthermore, in the transformed program, all of the newly added control flow paths correspond to faults that lead to skipping the branch instructions. Therefore, the transformed program includes all real violations and no bogus violations. ◀

5 Efficient Symbolic Execution for Fault Analysis

We now present the two pruning techniques for speeding up fault analysis based on symbolic execution. As mentioned earlier, one technique is based on bounding the maximal number of activated faults during an execution, and the other technique is based on skipping previously-explored suffixes. In Algorithm 1, these two techniques correspond to the subroutines UpdateFaultCounter, UpdateSuffixSummary, and PruningCondition.

5.1 Pruning Based on the Fault Count

If we ignore Lines 8–9 in Algorithm 3, then Algorithm 3 implements the pruning technique based on bounding the maximum number of activated faults during an execution. Specifically, the subroutine UpdateFaultCounter updates the fault counter during the symbolic execution of a program path, while the subroutine PruningCondition checks the fault bounding condition ($s.\texttt{FC} > \beta$) to decide if symbolic execution can terminate early for the program path.

Algorithm 3 Subroutines that support efficient fault bounding and redundancy pruning.

```
1: UpdateFaultCounter( state s, event t ) {
2:     if ( instruction t.inst is of the type assume(bFT = true) )
3:         s.FC ← s.FC + 1
4: }
5: PruningCondition( state s ) {
6:     if ( s.FC > the fault budget β )        //pruning based on the fault count s.FC
7:         return true;
8:     else if ( s.pcon ∧ ¬WP[s.l] is unsatisfiable )        //pruning based on the path summary WP[s.l]
9:         return true;
10:    else
11:        return false;
12: }
```

Recall that the state stack S stores the current execution path as a sequence $s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \ldots \xrightarrow{t} s$. At the initial state s_0, the value of the fault counter $s_0.\texttt{FC}$ is set to 0. Inside UpdateFaultCounter(s, t), every time a fault is activated, by executing the special branch statement **assume**($\texttt{bFT} = true$), the value of the fault counter $s.\texttt{FC}$ increases by 1. Thus, $s.\texttt{FC}$ is the total number of activated faults along the current path.

For the reason stated above, inside PruningCondition(s), we can end the symbolic execution of the current path when $s.\texttt{FC} > \beta$ (Lines 6–7), meaning that the total number of activated faults exceeds the fault budget. Thus, pruning is guaranteed to be sound (or safe), meaning it never skips a path that should not be skipped.

Another advantage of our method over existing methods is that, when there are multiple ways of injecting faults to violate a safety property, our method tries to return the one with fewer faults as follows. On top of the baseline symbolic execution algorithm (Section 3.2), which aims at exploring the program paths in a strictly depth-first search (DFS) order, at every branching point, our method symbolically explores the fault-free path (where $\texttt{bFT} = false$) before symbolically exploring the faulty path (where $\texttt{bFT} = true$). For each fault-controlled branching instruction, it means that our method always explores the inactive branch **assume**(!bFT) before it explores the active branch **assume**(bFT).

5.2 Pruning Based on the Path Summary

If we include Lines 8–9 in Algorithm 3, then the pruning condition leverages both the fault counter and the path summary. Specifically, the path condition $s.pcon$ captures the set of states that can be reached by executing the new prefix (from s_0 to s). At the same time, the suffix summary $\texttt{WP}[s.l]$ captures the weakest precondition computed from *all the explored suffixes starting from* s. Thus, if $s.pcon \wedge \neg\texttt{WP}[s.l]$ is unsatisfiable, it means that the set of states captured by $s.pcon$ is a subset of the set of states captured by $\texttt{WP}[s.l]$. Consequently, continuing the execution from s cannot lead to any previously unexplored behavior.

Fig. 5 illustrates the situation on the left-hand side. Depending on whether $s.pcon$ is fully included in $\texttt{WP}[s.l]$, there are two cases, as shown on the right-hand side of Fig. 5. While both

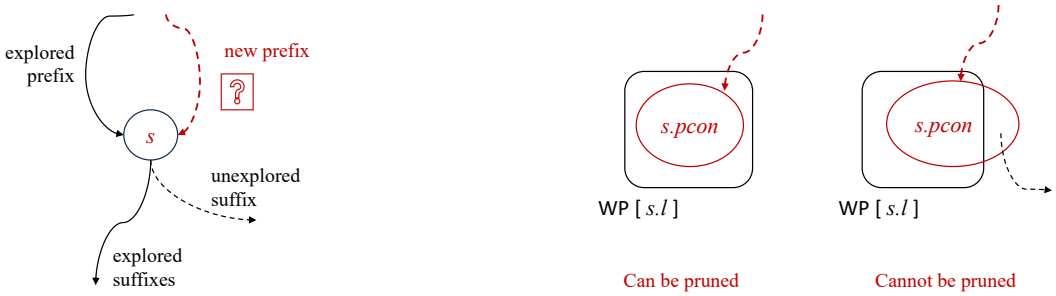


Figure 5 Illustrating how our pruning algorithm works: to decide if we can safely prune the current execution (new prefix) shown on the left, we check the pruning condition $s.pcon \rightarrow \texttt{WP}[s.l]$, which may lead to one of the two cases shown on the right.

$s.pcon$ and $\texttt{WP}[s.l]$ are symbolic expressions (or logical formulas represented in the format of SMT constraints), they capture two sets of program states. That is, program state belongs to $s.pcon$ if and only if that state makes $s.pcon$ evaluate to $true$.

We use Z3 to check if $s.pcon \wedge \neg\texttt{WP}[s.l]$ is unsatisfiable. Being unsatisfiable means that the set of program states captured by $s.pcon$ is indeed a subset of the set of program states captured by $\texttt{WP}[s.l]$. In other words, if we continue the current execution from state s, we will only explore the known behaviors. Since our goal is to avoid exploring the previously-explored behaviors, we can safely end the current execution early. This is the reason why pruning is guaranteed to be sound, meaning that it never skips executions that should not be skipped.

5.3 Updating the Path Summary

We now present the subroutine UpdateSuffixSummary. Internally, the subroutine has two steps: (1) computing the weakest precondition (wp) along each explored path, and (2) combining them to form a summary (WP) of all explored paths.

5.3.1 Weakest Precondition (wp)

Following Dijkstra [8], we compute the *weakest precondition* of an execution path by traversing the path backwardly. Let the path be $\pi = s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} s_n$. The last program location $s_n.l$ is either l_{end} (meaning the path ends without violation) or l_{bad} (meaning the path ends with a violation). Results of the weakest precondition computation are stored in a map named wp, where each entry $wp[s_n.l]$ is a symbolic expression for the program location $s_n.l$ associated with the state s_n.

- If s_n is the last state in π, then set $wp[s_n.l] = true$.
- For any other state s_i where $0 \leq i < n$, if the corresponding instruction $t_i.inst$ is of the type $\textbf{assume}(c)$, then set $wp[s_i.l] = wp[s_{i+1}.l] \wedge c$.
- For any other state s_i where $0 \leq i < n$, if the corresponding instruction $t_i.inst$ is of the type $v := expr$, then set $wp[s_i.l] = wp[s_{i+1}.l](v \mapsto expr)$.

Here, $(v \mapsto expr)$ means that, inside $wp[s_{i+1}.l]$, we replace the left-hand-side (lhs) variable v with the right-hand-side (rhs) expression $expr$.

5.3.2 From wp to the Union (WP)

Given a set of execution paths, we first compute wp for each path, and then union them together. The result, denoted WP, is a summary of all paths. Note that the union is performed

at each program location $s_i.l \in L$, not at each state $s_i \in S$. Thus, the resulting `WP` is a map where `WP`$[s_i.l]$ stores the union of all the individual $wp[s_i.l]$ expressions.

Specifically, after computing $wp_1, \cdots, wp_k$ for k paths, we combine them to create the union `WP` by applying

$$\texttt{WP}[l] \equiv \bigvee_{1 \le j \le k} wp_j[l]$$

for every program location $l \in L$ in the program.

5.3.3 Applied to the Running Example

We now illustrate how wp is computed along a path, and how wp's are combined to compute `WP` using the example program in Fig. 1. For ease of presentation, we assume that the fault budget β is set to 2. Under this assumption, there are five paths in the CFG of the transformed program, shown on the right-hand side of Fig. 4.

To compute wp_1 for Path #1, for example, we first initialize wp_1 to *true* at the end of the path. Then, we traverse the path backward. After encountering **assume**(c), we apply the rule $wp[s_i.l] = wp[s_{i+1}.l] \wedge c$ to obtain

$$wp_1[\texttt{assert}] \equiv true \wedge (m \neq n).$$

Next, we encounter `n+=2` in `BB3`. By applying the rule $wp[s_i.l] = wp[s_{i+1}.l](v \mapsto expr)$, we obtain

$$wp_1[\texttt{n+=2}] \equiv wp_1[\texttt{assert}](n \mapsto n+2) \equiv (m \neq n+2).$$

Next, we encounter `BB1':je BB3` (i.e., **assume**$(x = 0)$), for which we obtain

$$wp_1[\texttt{BB1':je BB3}] \equiv wp_1[\texttt{n+=2}] \wedge (x = 0) \equiv (m \neq n+2) \wedge (x = 0).$$

Finally, we obtain the entire wp_1 map as follows:

$$\begin{aligned} wp_1[\texttt{assert}] &\equiv (m \neq n), \\ wp_1[\texttt{n+=2}] &\equiv (m \neq n+2), \\ wp_1[\texttt{BB1':je BB3}] &\equiv (m \neq n+2) \wedge (x = 0), \\ wp_1[\texttt{je BB2}] &\equiv (m \neq n+2) \wedge (x = 0) \wedge \neg bFT1. \end{aligned}$$

Since there is only one explored path for the moment, we set `WP` $= wp_1$.

Next, the symbolic execution executes Path #2, starting at where it is forked at `je BB3`. When executing the assertion statement in `BB4` again at the end of Path #2, we apply the existing $WP[assert]$ to check against the current path condition. At this point, we have

$$s.pcon \equiv (m = 3) \wedge (n = 1) \wedge \neg bFT1 \wedge (x \neq 0) \wedge \neg bFT2,$$

and

$$\texttt{WP}[assert] \equiv (m \neq n).$$

We further check if the new path condition,

$$s.pcon \wedge \neg \texttt{WP},$$

which simplifies to

$$(\neg bFT1 \wedge \neg bFT2 \wedge x \neq 0) \wedge \neg(3 \neq 1).$$

Since the above constraint equals $false$, we stop executing Path #2 at the assertion statement and mark it as pruned. We then repeat the same process to compute the path summary for Path #2, starting with

$$wp_2[\texttt{assert}] \equiv true \wedge (m \neq n).$$

Eventually, we have $\texttt{WP}[l] \equiv wp_1[l] \vee \ldots \vee wp_5[l]$, and Paths 2, 4, and 5 are partially executed, due to the path summary based redundancy pruning in our method.

We now state the soundness of our technique.

▶ **Theorem 2.** *Our path pruning technique presented in this section is sound. That is, paths that are pruned away by our technique are guaranteed to be redundant.*

Proof. The soundness of our path pruning technique can be established as follows. First, recall that our path pruning technique relies on two methods: fault saturation and weakest precondition. For fault saturation, by definition, our pruning technique would only eliminate paths where the number of activated faults exceeds a predefined threshold. For weakest precondition (WP), whether a path should be pruned away is determined by checking if the WP-based path summary is covered by the current symbolic program state: if the answer is yes, it means that all possible extensions of the current path prefix have been covered by the summary of previously explored paths. Therefore, in both cases, paths that are pruned away by our technique are guaranteed to be redundant. ◀

6 Experiments

We have implemented our method as a software tool by leveraging the LLVM compiler [23], the KLEE symbolic execution engine [4], and the Z3 SMT solver [6]. After compiling a C program to LLVM bit-code, the tool first conducts fault modeling and then applies symbolic execution. Our fault modeling technique was implemented as an LLVM optimization pass, which transforms the original program P to the new program P' at the LLVM IR level. Our redundancy pruning technique was implemented as an extension of KLEE, where we first compute fault count and WP-based path summary, and then use Z3 to check the pruning conditions.

To allow a fair comparison with the state-of-the-art method, we have re-implemented the fault modeling and pruning techniques of [9] using KLEE. To allow an ablation study, we have also implemented a baseline of our symbolic execution method, which includes our fault modeling and fault bounding techniques, but excludes our summary-based pruning technique. In total, our implementation adds 4,000 lines of C++ code to LLVM and KLEE.

6.1 Benchmark Programs

We have evaluated our tool on 112 benchmark C programs. They include all of the 8 versions of VerifyPIN [9], which is a widely-used benchmark suite for fault attack analysis [39, 10, 38, 11, 33]. In addition, they also include three *complete* sets of benchmark programs from SV-COMP 2025's *ReachSafety* category, namely *bitvector*, *bitvector-loops*, and *array-crafted*. These three sets of benchmarks were designed to challenge verification tools on handling low-level bit-accurate computations (such as integer overflow and bitwise operations) crucial to

Table 1 Statistics of the control-intensive benchmark C programs used in the evaluation.

Category Name	Num. Programs	Min. LoC	Max. LoC	Avg. LoC	Benchmark Description
VerifyPIN	8	80	141	100.9	a set of benchmark programs widely used to evaluate fault attacks
array-crafted	43	44	86	54.5	SV-COMP benchmarks: programs with integer and array operations
bitvector	20	24	143	80.5	SV-COMP benchmarks: programs with bounded integer operations
bitvector (unbounded)	41	66	733	524.5	SV-COMP benchmarks: programs with bounded integers and unbounded loops

embedded systems, instead of the less accurate, standard mathematical integer verification for general-purpose programs. Thus, they align well with our research objectives of determining if hardware fault attacks cause abnormal control flows in embedded software, leading to the reachability of an error state.

Note that nine programs from the *bitvector* category (*jain_*.c*) were excluded because they contain infinite loops without branching or assertions, and thus are outside our research scope. Moreover, since KLEE is designed for test case generation, instead of proving that an assertion always holds, its capability of handling unbounded loops is limited. Therefore, we set a limit on the maximal number of symbolic execution steps for KLEE. Our goal is to ensure that KLEE terminates reasonably quickly, so we can have a baseline for comparing our pruning method with the current state-of-the-art method.

The statistics of the 112 benchmark programs are shown in Table 1. Column 1 shows the category name and Column 2 shows the total number of programs in the category. Columns 3–5 show the minimum, maximum, and average number of lines of code (LoC) for all programs in a category. Column 6 shows a brief description of the benchmark programs. Note that the LoC span a wide range, indicating that the programs have varying complexity; they include small programs with <50 LoC as well as larger programs with >700 LoC.

6.2 Evaluation Method

The experiments were designed to answer the following questions:

- Is our fault modeling technique more effective than the current state-of-the-art, thus allowing us to detect more real violations and fewer bogus violations?
- Is our new pruning technique effective in reducing the overall computational cost of the symbolic execution based fault analysis?

To evaluate the quality of our fault modeling technique, we compared it with the existing method on all benchmark programs. We set the maximum execution depth to 50 when exploring a single symbolic path for the benchmarks in *bitvector (unbounded)* to bound the execution of unbounded loops. The only exceptions are for programs *soft_float_3.c.cil.c* and *soft_float_3a.c.cil.c*, where we lowered the maximum depth to 30. This is because the two programs are more complex than others: the baseline method in KLEE cannot finish analyzing them when the execution depth is too large.

Our experiments were conducted on a computer with a 4.7 GHz AMD R9 7900X CPU and 32 GB of RAM running the Ubuntu 22.04 LTS Linux system. We set a 90-minute timeout per program. In the remainder of this section, we present the experimental results obtained to answer these questions.

Table 2 Results on fault modeling: the total number of programs for which safety violations were found by the existing method [9], our baseline method, and our optimized method, respectively. # No Violation is the number of programs on which a violation was not found, and # Found Violation is the number of programs on which a violation was found.

Category Name	# Programs	Existing Method [9]		Our Method	
		# No Violation	# Found Violation	# No Violation	# Found Violation
VerifyPIN	8	3	5	3	5
array-crafted	43	2	41	1	42
bitvector	20	3	17	1	19
bitvector (unbound)	41	28	13	17	24
Total	112	36	**76**	22	**90**

6.3 Results on Fault Modeling

We first present a summary of the safety violations found by different methods in Table 2, where Column 1 shows the category name and Column 2 shows the total number of programs in the category. Columns 3-4 show the results of the existing method, including the number of programs where a violation was not found, and the number of programs where the violation was found. Similarly, Columns 5-6 show the results of our method. The last row shows the total number of programs, the ones without violations, and the ones with violations. Overall, our method found 14 more programs with violations than the existing method (90 violations versus 76 violations); furthermore, a careful analysis shows that all of the 14 violations are real violations.

For example, our method found a violation in *bAnd1.c* while the existing method found no violation. A closer look at the program input computed by KLEE showed that it is a realistic violation. Conversely, our method found no violation in *gcd_2.c* while the existing method found a violation. However, a closer look at the program input computed by KLEE showed that it is a bogus violation. In both cases, the differences support the claim that our method more accurately models the impact of fault attacks on a victim program.

We now present detailed results for the VerifyPIN programs in Table 3, where Columns 1–2 show the name of each program and the number of injected faults, Column 3 shows the result of the existing method, and Column 4 shows the result of our method. For each method, we indicate whether a violation is detected (✗ means no and ✓ means yes). For these eight programs, our method detected the same number of violations as the existing method. That is, with at most one injected fault, a violation was found in the first five of the eight programs. When the number of injected faults was increased to 2, both our method and the existing method detected violations in all programs, including the last three of the eight programs.

We present detailed results for a subset of SV-COMP benchmarks in Table 4. It only includes programs where a violation was found by one method but not by the other method. Furthermore, ✓ means the found violation was real, whereas (✓) means the found violation was bogus (in the sense that it would not have occurred in practice). Overall, our method found a violation in 15 of the 16 programs, whereas the existing method found a violation in only 1 of the 16 programs.

Furthermore, a closer look (using the code snippet in Fig. 6) shows that the violation found by the existing method was bogus, whereas the violations found by our method were all real. The bogus violation was due to *test inversion*, an inaccurate fault modeling technique.

If inside the function `gcd_test()` the condition `(b!=0)` were inverted, the while-loop would be skipped entirely. Furthermore, if this occurred when `a=12` and `b=8`, the resulting

Table 3 Detailed results for VerifyPIN benchmarks, where the two methods found the same number of violations. Here, ✗ means violation was not found, and ✓ means violation was found.

Program Name	# Faults	Violation Found by Existing Method	Violation Found by Our Method
VerifyPIN_0.c	1	✓	✓
VerifyPIN_1.c	1	✓	✓
VerifyPIN_2.c	1	✓	✓
VerifyPIN_3.c	1	✓	✓
VerifyPIN_4.c	1	✓	✓
VerifyPIN_5.c	1	✗	✗
VerifyPIN_6.c	1	✗	✗
VerifyPIN_7.c	1	✗	✗
VerifyPIN_5.c	2	✓	✓
VerifyPIN_6.c	2	✓	✓
VerifyPIN_7.c	2	✓	✓

Table 4 Detailed results for the subset of SV-COMP benchmarks where a violation was found by one method but not by the other. Here, ✗ means no violation was found, and ✓ means a violation was found. Furthermore, (✓) means the violation was bogus (would not occur in practice).

Program Name	# Faults	Violation Found by Existing Method	Violation Found by Our Method
bAnd1.c	1	✗	✓
gcd_2.c	1	(✓)	✗
gcd_3.c	1	✗	✓
interleave_bits.c	1	✗	✓
s3_clnt_1.BV.c.cil-1a.c	1	✗	✓
s3_clnt_2.BV.c.cil-1.c	1	✗	✓
s3_clnt_2.BV.c.cil-1a.c	1	✗	✓
s3_clnt_3.BV.c.cil-1.c	1	✗	✓
s3_clnt_3.BV.c.cil-1a.c	1	✗	✓
s3_clnt_3.BV.c.cil-2.c	1	✗	✓
s3_clnt_3.BV.c.cil-2a.c	1	✗	✓
soft_float_2.c.cil.c	1	✗	✓
soft_float_2a.c.cil.c	1	✗	✓
soft_float_5.c.cil.c	1	✗	✓
sum02-1.c	1	✗	✓
bor2.c	2	✗	✓

`a=12` would violate `assert(b>a)`. However, this bogus violation cannot occur in the real world. With our more accurate fault modeling, the while-loop would never be skipped entirely. After executing the loop body at least once, the assertion would never be violated.

This is because, after the while-loop is compiled to machine code, the loop body is guarded by a conditional jump to the loop exit, e.g., `jz .exit` for `while(b!=0)`. When the loop condition `(b!=0)` is not satisfied, the jump will be taken to skip the loop body; but when the loop condition is satisfied, the jump will not be taken and the execution will fall through to the loop body. When a hardware fault turns the jump instruction into *nop*, the execution will also fall through to the loop body. Therefore, an attacker cannot physically force the while-loop to be skipped when the loop condition `(b!=0)` is satisfied.

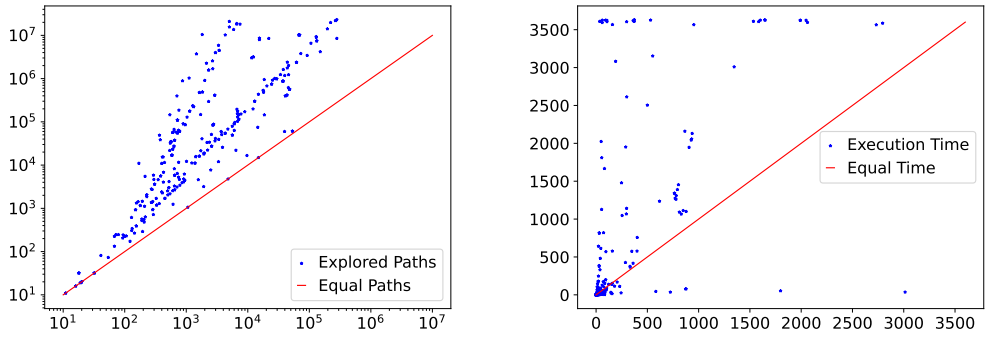
To summarize, these experimental results show that our new fault modeling technique is significantly more effective than the existing method.

6.4 Results on Redundancy Pruning

To evaluate the effectiveness of our redundancy pruning technique, we compared two aspects of the experimental results between our baseline and optimized methods: the number of explored paths and the execution time. To obtain more data, we increased the maximum

```
signed char gcd_test (signed char a, signed char b) {
    signed char t;
    if (a < (signed char)0)
        a = -a;
    if (b < (signed char)0)
        b = -b;
    while (b != (signed char)0) {
        t = b;
        b = a % b;
        a = t;
    }
    return a;
}
```

Figure 6 The code snippet (in gcd_2.c) shows that, while a bogus violation is reported due to test inversion (an inaccurate fault modeling technique), the bogus violation is avoided by using our new fault modeling technique.



(a) Number of explored paths: with our pruning technique (x-axis) versus the baseline (y-axis)

(b) Execution time in seconds: with our pruning technique (x-axis) versus the baseline (y-axis)

Figure 7 Experimental results that demonstrate the advantage of our pruning technique over baseline symbolic execution: the number of explored paths (left) and the execution time (right) of our symbolic execution method, with and without the new pruning technique. In both scatter plots, blue points above the red diagonal lines are winning cases for our new pruning technique.

fault budget from 1 to 4 for benchmarks in the categories *VerifyPIN*, *array-crafted*, and *bitvector*. During this process, we stopped increasing the number of injected faults whenever timeouts occurred. The reason why we excluded *bitvector (unbounded)* from this step is explained below.

First, the optimized method already outperformed the baseline method in terms of execution time across nearly all cases in *bitvector (unbounded)* when the fault budget was set to one. Therefore, increasing the number of faults would only yield limited additional insights. Second, the benchmarks in *bitvector (unbounded)* are more complex, and using a higher fault budget would require additional computational resources and time, which may not be feasible.

We present the number of explored paths and the running time in the scatter plots in Fig. 7. Since the number of explored paths falls in a wide range, we present the results on a log scale. Here, the number of explored paths is shown in Fig. 7a, where the baseline is on the y-axis and our optimized method is on the x-axis; the running time (in seconds) is

shown in Fig. 7b, where the baseline is on the y-axis and our optimized method is on the x-axis. In both sub-figures, blue points above the red diagonal lines are winning cases for our optimized method.

The results in Fig. 7a show that our optimized method almost always explores fewer paths than the baseline. Furthermore, in many cases, it explores significantly fewer paths. The results in Fig. 7b show that our optimized method almost always takes less time, although there are exceptions that show up as dots below the diagonal line. This is due to the overhead of WP-based pruning. Nevertheless, they typically occur when the running time is short; when the running time is long, our optimized method wins.

There are some cases in which our optimized method takes notably longer than the baseline, particularly for the benchmarks *gcd_2.c* and *gcd_3.c* when the number of faults was set to 3 and 4; the corresponding data points are located in the bottom-right corner of Fig. 7b. We inspected these benchmark programs and found that most of the time was spent on invoking solvers to determine the satisfiability of WP-based pruning constraints. But, overall, the results in Fig. 7b show that our pruning technique is effective in reducing the computational cost.

7 Related Work

Our method relies on symbolic execution to accurately and efficiently analyze the impact of hardware faults on software. While in theory, any software program may become the victim of hardware fault attacks, in practice, the most popular targets are embedded software programs [1, 42] and cryptographic software programs [7, 2]. These programs are the targets of our method. As mentioned earlier in this paper, BINSEC [9] is the most closely related prior work, but it is not as effective as our method in terms of fault modeling, as demonstrated by our experimental evaluation.

Besides BINSEC [9], which represents the current state-of-the-art, there are various earlier works on fault analysis [31, 24, 12, 20, 21, 22]. However, they leverage existing techniques like static analysis and concolic execution in a more or less straightforward fashion; in particular, they do not focus on improving the underlying symbolic analysis procedures using redundancy pruning techniques.

Although symbolic execution is a popular technique for analyzing both sequential and concurrent software [16, 17, 18] and it has been implemented in many existing tools including CUTE [34], KLEE [4] and SAGE [13], it is known to suffer from the path explosion problem. Various techniques have been developed to mitigate path explosion, including forward analysis techniques such as state merging [19] and backward analysis techniques such as post-conditioned pruning [41, 40, 15]. However, none of these existing techniques specifically target software programs under fault attacks.

The reason why we focus on symbolic execution is because our goal in this work is to detect safety violations in software programs, for which symbolic execution is particularly strong. If the goal were to generate proofs of no safety violations, other techniques that focus primarily on verification (instead of falsification) could be used instead. Specifically, such techniques can be used to prove fault-resistance [26, 39, 36, 32]. For example, Tollec et al. [39] applied bounded model checking to software programs running on a RISC-V processor to analyze the impact of faults injected at the micro-architectural level, while Tan et al. [36] used SAT solvers to evaluate the fault-resistance of cryptographic circuits. Our method is different in that it is geared toward detecting violations and, more importantly, it is a software-only solution where the impact of hardware faults is abstracted into the *branch*

instruction skipping threat model.

Beyond software-only solutions and tools, there is a body of work on physically conducting fault injection using power [5] and clock glitching [29], laser beams [35] on hardware boards, as well as studying the impact of the injected faults on software programs using hardware-software co-simulation [14, 25]. These empirical studies have led to the *branch instruction skipping* threat model used in our method [1, 42] as well as other threat models including *memory bit flipping* [37, 27, 28]. While we focus exclusively on modeling instruction skipping in this work, we foresee no technical difficulty in extending our method to the *memory bit flipping* threat model.

8 Conclusion

We have presented a symbolic execution based method to analyze the impact of hardware faults on the safety of a software program both accurately and efficiently. The method has two new techniques. The first one is a compiler-based program transformation technique for accurate fault modeling. The second one is a redundancy pruning technique that leverages fault saturation and weakest precondition to avoid symbolically executing redundant program paths. Our experimental evaluation on two sets of benchmark programs shows that the method significantly outperforms the current state-of-the-art method, in that it can detect previously missed violations and can significantly reduce the overall computational cost.

References

1 Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *Proc. IEEE*, 94(2):370–382, 2006. doi:10.1109/JPROC.2005.862424.

2 Ali Asghar Beigizad, Hadi Soleimany, Sara Zarei, and Hamed Ramzanipour. Linked fault analysis. *IEEE Trans. Inf. Forensics Secur.*, 19:632–645, 2024. doi:10.1109/TIFS.2023.3327658.

3 Dirk Beyer. State of the art in software verification and witness validation: SV-COMP 2024. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part III*, Lecture Notes in Computer Science, pages 299–329. Springer, 2024. doi:10.1007/978-3-031-57256-2_15.

4 Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008. URL: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.

5 Hamid Choukri and Michael Tunstall. Round reduction using faults. *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2005.

6 Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, Lecture Notes in Computer Science, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

7 Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, and Assia Tria. Electromagnetic transient faults injection on a hardware and a software implementations of AES. In Guido

Bertoni and Benedikt Gierlichs, editors, *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography, Leuven, Belgium, September 9, 2012*, pages 7–15. IEEE Computer Society, 2012. doi:10.1109/FDTC.2012.15.

8 Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. URL: https://www.worldcat.org/oclc/01958445.

9 Soline Ducousso, Sébastien Bardin, and Marie-Laure Potet. Adversarial reachability for program-level security analysis. In Thomas Wies, editor, *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings*, Lecture Notes in Computer Science, pages 59–89. Springer, 2023. doi:10.1007/978-3-031-30044-8_3.

10 Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen, and Philippe de Choudens. FISSC: A fault injection and simulation secure collection. In Amund Skavhaug, Jérémie Guiochet, and Friedemann Bitsch, editors, *Computer Safety, Reliability, and Security - 35th International Conference, SAFECOMP 2016, Trondheim, Norway, September 21-23, 2016, Proceedings*, Lecture Notes in Computer Science, pages 3–11. Springer, 2016. doi:10.1007/978-3-319-45477-1_1.

11 Guillaume Girol, Guilhem Lacombe, and Sébastien Bardin. Quantitative robustness for vulnerability assessment. *Proc. ACM Program. Lang.*, 8(PLDI):741–765, 2024. doi:10.1145/3656407.

12 Thomas Given-Wilson, Nisrine Jafri, and Axel Legay. Combined software and hardware fault injection vulnerability detection. *Innov. Syst. Softw. Eng.*, 16(2):101–120, 2020. URL: https://doi.org/10.1007/s11334-020-00364-5, doi:10.1007/S11334-020-00364-5.

13 Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, 2012. doi:10.1145/2093548.2093564.

14 Jacob T. Grycel and Patrick Schaumont. Simplifi: Hardware simulation of embedded software fault attacks. *Cryptogr.*, 5(2):15, 2021. URL: https://doi.org/10.3390/cryptography5020015, doi:10.3390/CRYPTOGRAPHY5020015.

15 Shengjian Guo, Markus Kusano, and Chao Wang. Conc-iSE: incremental symbolic execution of concurrent software. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 531–542. ACM, 2016. doi:10.1145/2970276.2970332.

16 Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, and Aarti Gupta. Assertion guided symbolic execution of multithreaded programs. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 854–865. ACM, 2015. doi:10.1145/2786805.2786841.

17 Shengjian Guo, Meng Wu, and Chao Wang. Symbolic execution of programmable logic controller code. In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 326–336. ACM, 2017. doi:10.1145/3106237.3106245.

18 Shengjian Guo, Meng Wu, and Chao Wang. Adversarial symbolic execution for detecting concurrency-related cache timing leaks. In Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu, editors, *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 377–388. ACM, 2018. doi:10.1145/3236024.3236028.

19 Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12,*

Beijing, China - June 11 - 16, 2012, pages 193–204. ACM, 2012. doi:10.1145/2254064.2254088.

20 Guilhem Lacombe, David Féliot, Etienne Boespflug, and Marie-Laure Potet. Combining static analysis and dynamic symbolic execution in a toolchain to detect fault injection vulnerabilities. *J. Cryptogr. Eng.*, 14(1):147–164, 2024. URL: https://doi.org/10.1007/s13389-023-00310-8, doi:10.1007/S13389-023-00310-8.

21 Julien Lancia. Detecting fault injection vulnerabilities in binaries with symbolic execution. In *14th International Conference on Electronics, Computers and Artificial Intelligence, ECAI 2022, Ploiesti, Romania, June 30 - July 1, 2022*, pages 1–8. IEEE, 2022. doi:10.1109/ECAI54874.2022.9847500.

22 Daniel Larsson and Reiner Hähnle. Symbolic fault injection. In Bernhard Beckert, editor, *Proceedings of 4th International Verification Workshop in connection with CADE-21, Bremen, Germany, July 15-16, 2007*, CEUR Workshop Proceedings. CEUR-WS.org, 2007. URL: https://ceur-ws.org/Vol-259/paper09.pdf.

23 Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004. doi:10.1109/CGO.2004.1281665.

24 Hoang M. Le, Vladimir Herdt, Daniel Große, and Rolf Drechsler. Resilience evaluation via symbolic fault injection on intermediate code. In Jan Madsen and Ayse K. Coskun, editors, *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*, pages 845–850. IEEE, 2018. doi:10.23919/DATE.2018.8342123.

25 Zhenyuan Liu, Dillibabu Shanmugam, and Patrick Schaumont. Faultdetective explainable to a fault, from the design layout to the software. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(4):610–632, 2024. URL: https://doi.org/10.46586/tches.v2024.i4.610-632, doi:10.46586/TCHES.V2024.I4.610-632.

26 Nicolas Moro, Karine Heydemann, Emmanuelle Encrenaz, and Bruno Robisson. Formal verification of a software countermeasure against instruction skip attacks. *J. Cryptogr. Eng.*, 4(3):145–156, 2014. URL: https://doi.org/10.1007/s13389-014-0077-7, doi:10.1007/S13389-014-0077-7.

27 Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel SGX. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1466–1482. IEEE, 2020. doi:10.1109/SP40000.2020.00057.

28 Onur Mutlu and Jeremie S. Kim. Rowhammer: A retrospective. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 39(8):1555–1571, 2020. doi:10.1109/TCAD.2019.2915318.

29 Bo Ning and Qiang Liu. Modeling and efficiency analysis of clock glitch fault injection attack. In *Asian Hardware Oriented Security and Trust Symposium, AsianHOST 2018, Hong Kong, China, December 17-18, 2018*, pages 13–18. IEEE, 2018. URL: https://doi.org/10.1109/AsianHOST.2018.8607175, doi:10.1109/ASIANHOST.2018.8607175.

30 Colin O'Flynn. BAM BAM!! on reliability of EMFI for in-situ automotive ECU attacks. *IACR Cryptol. ePrint Arch.*, 2020:937, 2020. URL: https://eprint.iacr.org/2020/937.

31 Karthik Pattabiraman, Nithin Nakka, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. SymPLFIED: Symbolic program-level fault injection and error detection framework. *IEEE Trans. Computers*, 62(11):2292–2307, 2013. doi:10.1109/TC.2012.219.

32 Basile Pesin, Sylvain Boulmé, David Monniaux, and Marie-Laure Potet. Formally verified hardening of C programs against hardware fault injection. In Kathrin Stark, Amin Timany, Sandrine Blazy, and Nicolas Tabareau, editors, *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2025, Denver, CO, USA, January 20-21, 2025*, pages 140–155. ACM, 2025. doi:10.1145/3703595.3705880.

33 Yanis Sellami, Guillaume Girol, Frédéric Recoules, Damien Couroussé, and Sébastien Bardin. Inference of robust reachability constraints. *Proc. ACM Program. Lang.*, 8(POPL):2731–2760, 2024. doi:10.1145/3632933.

34 Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In Michel Wermelinger and Harald C. Gall, editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 263–272. ACM, 2005. doi:10.1145/1081706.1081750.

35 Sergei P. Skorobogatov and Ross J. Anderson. Optical fault induction attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, Lecture Notes in Computer Science, pages 2–12. Springer, 2002. doi:10.1007/3-540-36400-5_2.

36 Huiyu Tan, Pengfei Gao, Fu Song, Taolue Chen, and Zhilin Wu. Sat-based formal verification of fault injection countermeasures for cryptographic circuits. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(4):1–39, 2024. URL: https://doi.org/10.46586/tches.v2024.i4.1-39, doi:10.46586/TCHES.V2024.I4.1-39.

37 Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. CLKSCREW: exposing the perils of security-oblivious energy management. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 1057–1074. USENIX Association, 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang.

38 Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydemann, and Mathieu Jan. Exploration of fault effects on formal RISC-V microarchitecture models. In *Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2022, Virtual Event / Italy, September 16, 2022*, pages 73–83. IEEE, 2022. doi:10.1109/FDTC57191.2022.00017.

39 Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydemann, and Mathieu Jan. μarchifi: Formal modeling and verification strategies for microarchitectural fault injections. In Alexander Nadel and Kristin Yvonne Rozier, editors, *Formal Methods in Computer-Aided Design, FMCAD 2023, Ames, IA, USA, October 24-27, 2023*, pages 101–109. IEEE, 2023. URL: https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_18, doi:10.34727/2023/ISBN.978-3-85448-060-0_18.

40 Qiuping Yi, Zijiang Yang, Shengjian Guo, Chao Wang, Jian Liu, and Chen Zhao. Postconditioned symbolic execution. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, pages 1–10. IEEE Computer Society, 2015. doi:10.1109/ICST.2015.7102601.

41 Qiuping Yi, Zijiang Yang, Shengjian Guo, Chao Wang, Jian Liu, and Chen Zhao. Eliminating path redundancy via postconditioned symbolic execution. *IEEE Trans. Software Eng.*, 44(1):25–43, 2018. doi:10.1109/TSE.2017.2659751.

42 Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. Fault attacks on secure embedded software: Threats, design, and evaluation. *J. Hardw. Syst. Secur.*, 2(2):111–130, 2018. URL: https://doi.org/10.1007/s41635-018-0038-1, doi:10.1007/S41635-018-0038-1.