Resilience Evaluation via Symbolic Fault Injection on Intermediate Code

Hoang M. Le\textsuperscript{1} Vladimir Herdt\textsuperscript{1} Daniel Große\textsuperscript{1,2} Rolf Drechsler\textsuperscript{1,2}

\textsuperscript{1}Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
\textsuperscript{2}Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
\{hle,vherdt,grosse,drechsle\}@informatik.uni-bremen.de

Abstract—There is a growing need for error-resilient software that can tolerate hardware faults as well as for new resilience evaluation techniques. For the latter, a promising direction is to apply formal techniques in fault injection-based evaluations to improve the coverage of evaluation results. Building on the recent development of Software-implemented Fault Injection (SWiFI) techniques on compiler’s intermediate code, this paper proposes a novel resilience evaluation framework combining LLVM-based SWiFI and SMT-based symbolic execution. This novel combination offers significant advantages over state-of-the-art approaches with respect to accuracy and coverage.

I. INTRODUCTION

With the rapidly increasing hardware complexity and stringent power and cost constraints, embedded systems become more and more vulnerable to hardware faults. Moreover, the increased importance of software in embedded system design has made software resilience a major challenge nowadays. There is a growing need for error-resilient software that can tolerate hardware faults as well as for resilience evaluation techniques and tools. For the latter, fault injection (FI) is a widely accepted approach that is also being recommended in safety standards such as ISO26262.

There exists already a vast amount of work on hardware FI at various levels of abstraction. Expectedly, the accuracy of FI-based resilience evaluation decreases as one moves up the abstraction levels. However, physical hardware-based injection can be very expensive, impractical and has limited controllability as well as observability. Even FI campaigns at RTL are too slow for today’s systems and RTL simulation is only possible at a very late stage in the design flow. Therefore, researchers have been looking for alternatives at higher levels of abstraction that allow much faster evaluation in earlier design stages. One currently very active research direction is to apply error effect simulation [1] using Virtual Prototypes that are essentially executable abstract software models of hardware. Another direction, which this paper will focus on, is to emulate hardware faults at the software level through Software-implemented Fault Injection (SWiFI) techniques.

Recently, SWiFI techniques operating on compiler’s intermediate code, in particular the LLVM intermediate representation (LLVM-IR), have been gaining attention in the research community (see e.g. [2]–[4]). LLVM-IR is a hardware-independent assembler-like language, which sits between the source language (e.g. C) and the target assembly language (e.g. X86 or ARM). Injecting fault at the level of LLVM-IR is an attractive trade-off. On the one hand, LLVM-IR is low-level enough to represent many hardware faults. On the other hand, LLVM provides, compared to assembly languages, extensive support for program analysis and transformations that ease the tasks of source-code mapping and understanding the error resilience behavior of the program under test.

The major weakness of traditional FI-based evaluations is the lack of coverage. Typically, before the FI experiments, concrete test cases for the program under test are selected. Then, faults are injected in statistical manner during the execution of these test cases and execution results are analyzed to classify the behavior of the program under faults (e.g. failure, fault detected but tolerated, benign fault, etc.) [5]. The most useful provoked behavior is arguably a failure caused by a fault that escaped detection mechanisms. However, due to the inherent statistical nature, “corner-case” failure-inducing faults can be missed, which can mislead the evaluation to deem the program to be error-resilient while it is not. The second issue is that the evaluation result depends heavily on the quality of the selected test cases. If a part of the program is not executed by the test cases, faults injected in this part cannot be activated. Thus, to improve the confidence of FI-based evaluations, there is a compelling need to apply formal techniques.

Conceptually, both issues of finding “corner-case” faults and “good” test cases can be addressed by symbolic fault injection [6], [7]. In contrast to conventional FI, symbolic FI does not inject a concrete fault but use a symbolic value to represent all possible faulty values. Afterwards, the program is symbolically executed to obtain all possible outcomes induced by the symbolic fault. However, the research area of symbolic FI is still in its infancy. Existing approaches suffer from severe drawbacks w.r.t. accuracy, coverage and also usability (more details in Section II).

In this paper, we propose a novel symbolic LLVM-based SWiFI evaluation framework for resilience evaluation with clear improvements over state-of-the-art. Our framework performs symbolic bit-precise FI and propagation on the LLVM-IR, and thus the evaluation will not produce any false negative. Symbolic bit-precise FI is achieved by representing the faulty value as a symbolic value and adding additional constraints to precisely capture the semantics of the fault. For the symbolic propagation analysis, we employ the state-of-the-art symbolic execution engine KLEE [8]. KLEE operates directly on the LLVM-IR and employs various SMT solvers internally to...
explore the state space of all symbolic values systematically path-wise. Thanks to the combination of symbolic execution paths with symbolic FI, the need for selecting “good” test cases for FI is also eliminated, since each symbolic path represents a distinct equivalence class of concrete test cases. If a failure is detected, a concrete test case together with a concrete fault can be derived from the symbolic representation and presented to the user. If all paths together with all symbolic faults have been explored without detecting a failure, the program is guaranteed to be resilient to these considered faults. To the best of our knowledge, this work is the first to propose such a complete symbolic SWIFI-based evaluation framework, especially for the LLVM-IR.

II. RELATED WORK

For hardware, several resilience evaluation approaches based on symbolic techniques have been proposed in the literature. Without attempting to be comprehensive, we briefly discuss a few notable approaches. To evaluate the coverage of error detection circuits for transient faults, Krautz et al. [9] combine symbolic FI and BDD-based symbolic simulation at RTL. The work by Darbari et al. [10] takes another route to combine symbolic FI and BDD-based symbolic simulation at RTL. The work by Darbari et al. [10] takes another route to combine symbolic FI and BDD-based symbolic simulation at RTL.

III. PRELIMINARIES

A. LLVM Intermediate Representation

Clang – the compiler front-end of LLVM – first compiles C source code into LLVM-IR, which is basically a platform-independent virtual instruction set. LLVM-IR resembles assembly languages and offers the key operations of ordinary processors but abstracts away target-specific constraints (e.g., hardware registers or pipelines). Most operations are performed on virtual registers in Static Single Assignment (SSA) form. Opposed to a real processor, the set of virtual registers is infinite. Load and store operations are used to transfer values between registers and memory. Allocation of objects is done via alloca and malloc instructions. Computation instructions include among others arithmetic operations, bitwise operations and comparison. A set of control flow instructions is also available (e.g., br, switch, ret, etc.) For further details we refer to the reference manual of LLVM 3.4 (http://releases.llvm.org/3.4.2/docs/LangRef.html), which is used in this work.

In the remainder of the paper we use the example shown in Fig. 1, which computes the next power of 2 of a given integer, e.g. npo2(5)=8, npo2(8)=16, etc. Fig. 2 shows an excerpt of the corresponding LLVM-IR. Line 14-18 represent the intermediate code for Line 1 of Fig. 1.

B. SMT-based Symbolic Execution

Let $\mathcal{E} = (IN, INSTR)$ be an execution trace of the program under analysis $P$. $IN = \{i_{n_1}, \ldots, i_{n_m}\}$ are the inputs that lead $P$ to the execution of the ordered instruction list $INSTR = \{instr_1, \ldots, instr_n\}$. While we focus on LLVM instructions, the formulation is not tied to a particular level of abstraction, i.e. an instruction can be an assembly instruction, an LLVM instruction or even a C statement.

---

```c
1 int npo2(int x) {  
2   int result = x;  
3   for (int i = 1; i < sizeof(int) * 8; i *= 2) {  
4     result = result | (result >> i);  
5   }  
6   result = result + 1;  
7   return result;  
8 }
```

---

```c
1 define int @npo2(i32 %x) #0 (  
2   entry:  
3     %w.addr = alloca i32, align 4  
4     %result = alloca i32, align 4  
5     %i = alloca i32, align 4  
6     store i32 %x, i32* %w.addr, align 4  
7     %0 = load i32* %w.addr, align 4, !dbg !123  
8     store i32 %0, i32* %result, align 4, !dbg !123  
9     store i32 1, i32* %i, align 4, !dbg !124  
10    br label %for.cond, !dbg !124  
11  for.cond:  
12    ...; omitted for compact presentation  
13    for.body:  
14      %3 = load i32* %result, align 4, !dbg !126  
15      %4 = load i32* %i, align 4, !dbg !126  
16      %shr = shr i32 %3, %4, !dbg !126  
17      %or = or i32 %3, %shr, !dbg !126  
18     store i32 %or, i32* %result, align 4, !dbg !126  
19     %5 = load i32* %i, align 4, !dbg !124  
20     %mul = mul nsw i32 %4, %5, !dbg !124  
21     store i32 %mul, i32* %i, align 4, !dbg !124  
22     br label %for.cond, !dbg !124  
23  for.end:  
24    ...; omitted to for compact presentation  
25 }
```

---
Assuming that $\mathcal{P}$ does not have non-deterministic behavior (e.g., randomization), the ordered instruction list $\mathcal{IN}_{STR}$ depends entirely on the evaluation of $\mathcal{IN}$ at every branching instruction. SMT-based symbolic execution computes these branch decisions as path conditions based on symbolic inputs and fork the execution into two new execution states when both directions of a branch are feasible. The feasibility is decided using SMT solvers. For bit-precise reasoning, $QF_{\text{ABV}}$ fragment of SMT is most often used. We denote $PC(\mathcal{IN}_{STR})$ as the path condition of the instruction list $\mathcal{IN}_{STR}$.

State-of-the-art implementations provide many optimizations that go beyond basic symbolic execution (e.g., constraint simplification, path query caching, etc.). Interested readers are referred to [8] for more detail.

IV. Symbolic Software-Implemented Fault Injection – An Overview

In this section we describe the ingredients of our symbolic SWIFl approach and provide a classification of its different variants with varying strength. This high-level description serves as foundation for the concrete implementation outlined in Section V. We reuse the notation introduced Section III.

A. Considered Faults

Our approach assumes that hardware faults have already propagated as bit-flips in a single software-visible location, e.g., instruction operand or register. This bit-flip model, while not necessarily in direct correlations with real faults [13], is a consensus among existing SWIFl approaches targeting hardware faults, see e.g. [3], [4], [7], [14], [15]. Better fault models are under active research [16]–[18]. In particular, we consider bit-flips in LLVM instructions. Recent findings [3] showed that LLVM-based FI produces comparable results to assembly-level FI w.r.t. silent data corruption (SDC), which is arguably the most important class of failures at software level. This motivates us also to focus on SDC in this work and restrict FI to similar classes of LLVM instructions as used in [3]. More precisely, we inject bit-flips into the target virtual register of LLVM computation and load instructions (see Section III-A). For example, in the LLVM-IR of $\text{npo2}$ in Fig. 2, register $\%\text{shr}$ (Line 16, result of an arithmetic right shift) or register $\%5$ (Line 19, result of a load operation) are candidates for injecting bit-flips.

B. Fault List Formalization

For an instruction $\text{instr}$ in $\mathcal{IN}_{STR}$, assuming that only one single fault can occur, we denote its fault list as $\mathcal{FL}(\text{instr}) = \\{ \ldots , (\text{floc}_i, \text{fval}_i), \ldots \}$ with $\text{floc}_i$ being a fault location within $\text{instr}$ and $\text{fval}_i$ a faulty value that can be injected at this location. A faulty value can differs from the correct value by one single or multiple bits, depending on the considered fault model. An injected fault can cause the execution of $\mathcal{P}$ using inputs from $\mathcal{IN}$ to deviate from $\mathcal{IN}_{STR}$. This deviation can be harmless or induce a crash or more dangerously, produce silently a wrong result, i.e. SDC. SDC is also possible even in case the same $\mathcal{IN}_{STR}$ is followed.

C. Coverage Strength of Symbolic SWIFl

Obviously, to completely assess the behavior of $\mathcal{P}$ under fault, every possible execution trace $\mathcal{ET}$ must be considered together with its associated fault list $\mathcal{FL}$. Traditional SWIFl techniques relies on statistical sampling of this combined state space of $\mathcal{ET}$ and $\mathcal{FL}$ denoted as $\mathcal{ET} \times \mathcal{FL}$. A set of $\mathcal{ET}$ from functional verification test cases is often reused and for each $\mathcal{ET}$, the fault lists are randomly sampled according to a distribution specified by the user. Sometimes the fault lists are small enough, such that they can be exhaustively enumerated. Explicit enumeration of all possible $\mathcal{ET}$ or $\mathcal{IN}$, on the other hand, is often not feasible, even for very small programs. Traditional SWIFl techniques are the lowest on the coverage strength scale. Symbolic SWIFl improves coverage by allowing $\mathcal{FL}$ and $\mathcal{ET}$ to be symbolic values. This enables an appropriate symbolic reasoning back-end to explore the state space implicitly, which is often more efficient than explicit enumeration. We discuss some variants of symbolic SWIFl in the following in increasing order of strength:

1) Using the same set of $\mathcal{ET}$ as traditional SWIFl, the first variant enumerates the fault location $\text{floc}$ on each concrete $\mathcal{ET}$ but allows the faulty value $\text{fval}$ to be symbolic. SymPLIFIED belongs to this class (although it is not discussed how the set of $\mathcal{ET}$ can be obtained). SymPLIFIED also do not distinguish single and multiple bit-flips but over-approximate.

2) The next stronger variant is achieved by allowing also $\text{floc}$, and thus the entire state space $\mathcal{FL}$ of each concrete $\mathcal{ET}$, to be symbolic. We are not aware of any existing SWIFl approach with support for symbolic $\text{floc}$.

3) Coverage can be improved further on the $\mathcal{ET}$ side by using a complete set of $\mathcal{ET}$. While several coverage criteria can be employed to create this set, we use path coverage, which is the strongest code coverage criterion.

4) Finally, highest on the scale is a full symbolic exploration of $\mathcal{ET} \times \mathcal{FL}$.

We consider the last two variants, denoted as $\text{PathCovET-SymFL}$ and $\text{SymET-SymFL}$, respectively. Furthermore, we do not over-approximate but use bit-precise fault modeling by constraining the symbolic $\text{fval}$ (more details in Section V-C). The coverage strength of $\text{PathCovET-SymFL}$ and $\text{SymET-SymFL}$ comes at the expense of scalability, since the combined space $\mathcal{ET} \times \mathcal{FL}$ is often very large. Both variants are meant to be applied at the unit level for small but critical code.

D. Error Detection Mechanisms

The main application of our framework is to evaluate the resilience of a program with error detectors under hardware faults. The primarily supported error detection mechanism is based on the concept of executable assertions. This promising program-level low-cost detection mechanism [15], [19], [20], checks the validity of a set of invariants on program variables at different points at runtime. Errors that violate these invariants can thus be detected and prevented from reaching program output. We employ a special CHECK annotation to integrate these detectors into $\mathcal{P}$. It is assumed that the detectors are not affected by faults.
In this section we describe the implementation of the symbolic SWiFI techniques outlined in the last section. As mentioned, they are implemented on top of KLEE. While it is possible to implement everything within the KLEE core, we take the alternative approach of creating an add-on C library for symbolic SWiFI to minimize dependencies. Due to space limitation, we only describe the implementation of SymET-SymFL (i.e. full symbolic exploration of $\mathcal{ET} \times \mathcal{FL}$). Most is shared with PathCovET-SymFL with the exception of a non-trivial extension to the seed functionality of KLEE. Given a concrete input sequence $\mathcal{IN}$, called seed, KLEE will follow the same execution trace $\mathcal{ET}$ taken by $\mathcal{IN}$ to build the initial search tree. After that, normal symbolic execution is resumed. Thus, the original seed functionality of KLEE just provides a mean to influence the order of exploration. We modify this functionality to strictly follow $\mathcal{IN}$ and allow forking only at branches involving symbolic variables related to the fault list $\mathcal{FL}$. The details can be worked out by inspecting the source code available at https://github.com/agra-uni-bremen/symbolic-swifi.

A. Overall Workflow

The overall workflow is shown in Fig. 3. In the first step an analysis testbench is created which executes the program with and without FI and compares the results to detect SDC. The same (symbolic) inputs are used for both runs. The testbench is compiled together with the program under analysis and into LLVM bitcode (i.e. physical representation of LLVM-IR). Then, this bitcode goes through an LLVM transformation pass implement in KLEE to insert calls to symbolic SWiFI functions from the compiled library (error_lib.cc in the workflow). Finally, the bitcode is analyzed using KLEE’s symbolic execution will report all elements from the combined state space $\mathcal{ET} \times \mathcal{FL}$ that cause the fault-injected program to produce SDC. Such trace means that the error detectors were unable to prevent the injected fault. In this case the user can try to improve the detectors and re-run the analysis.

In the following we describe the LLVM pass as well as the symbolic SWiFI library in more details and give an illustration.

B. LLVM-based Instrumentation

We employ an LLVM function pass to implement the instrumentation. The function pass runs for every function in the bitcode. For every function we iterate over all basic blocks and their instructions. This allows to reach and instrument every instruction I in the LLVM-IR.

For each instruction that is an injection candidate (see Section IV-A), the target register is passed through a maybe_bitflip function provided by the symbolic SWiFI library. This function non-deterministically injects a single bitflip error and can be controlled at runtime (more details follow in Section V-C). Subsequently, the result of the maybe_bitflip function is used in place of the original instruction result. This instrumentation is illustrated in Fig. 4. It shows an LLVM-IR excerpt (Line 20-21 from Fig. 2) for the C statement $i *= 2$ in original and instrumented form.

C. Symbolic SWiFI Library

Fig. 5 shows the implementation of the library. The variables $global_fi_active$ and $local_fi_active$ are used to control the injection. The first is used within the testbench, while the second inside the program for a more fine grained control of injection, in particular to avoid injection in initialization logic or directly into an output variable. The CHECK macro implements the error detectors. Finally, $error_case$ and $error_counter$ are used to model symbolic injection location $\text{floc}$. The variable $error_case$ is initialized with a symbolic value to represent a non-deterministic location.
The `maybe_bitflip` function increments the counter for the number of injection locations `error_counter` and nondeterministically selects a location by comparing `error_case == error_counter`, that can evaluate to both true (i.e. inject) and false (i.e. no injection). When KLEE encounters the corresponding `br` instruction, the execution will be forked into two new execution paths. Please note that, on the path where `error_case == error_counter` evaluates to true, `error_case` will no longer be symbolic. Consequently, `maybe_bitflip` injects only at single location.

The injected fault value `fval` is determined in the `bitflip` function. It returns a new symbolic `int` which is constrained (Line 15-16) to differ to the function argument `x` in exactly one bit. Essentially, this symbolic `int` represents all 32 possible bit-flips applied to `x`. The implementation of symbolic SWiFI as a library offers flexibility in modeling fault semantics. For example, it is possible to inject double bit-flips (at the same location) by changing the constraints in the function `bitflip`.

D. Illustration

Fig. 6 shows the `npo2` program now with error detectors (the `CHECK` annotation) and the testbench provided in the `main` function. As can be seen, the testbench initializes the symbolic SWiFI library, prepares symbolic inputs, execute `npo2` with and without FI and finally compare the results. The first two `CHECKs` ensure that the input is in the supported range. The `CHECK` in the middle of the loop catches faulty cases where `i` becomes negative, which would cause a very long loop. Two `CHECKs` at the end verify that `result` is functionally correct, i.e. it is a power of 2 and smaller than `(2 * x)`. Additionally, an internal check for oversized shift, which happens when a bit-flip is injected into `i` in the evaluation of `(result >> i)` (i.e. register `%4`, Line 15 in Fig. 2), is employed. At the first glance these `CHECKs` seem to be able to catch all errors. However, `SymET-SymFL` would reveal very fast a few SDC cases. For example, a bit-flip during the assignment `result = x` (i.e. register `%0`, Line 7 in Fig. 2) would make the initial value of `result` before the loop too small, such that at the end it still satisfies the two functional `CHECKs`, but it is smaller than `x`. After adding `CHECK(result > x)` before the return statement, `SymET-SymFL` would report that no SDC can be detected now. Detailed results are shown in the next section.

VI. EXPERIMENTS

The implementation is carried out on top of KLEE v1.3.0 with LLVM 3.4.2 and STP 2.1.2 as the SMT solver. We apply the implemented techniques to a set of programs with varying complexity. All experiments were run on a Linux machine with 8 Intel cores at 3.5GHz and 32 GB memory. In the following we introduce the benchmarks and present the obtained results. For reproducibility purposes, the implementation and the benchmarks are freely available at https://github.com/agra-uni-bremen/symbolic-swifi.

A. Benchmarks

We evaluate our technique on the `npo2` program introduced earlier, a bubblesort implementation, a well-known aircraft traffic collision avoidance system (TCAS) from the Siemens suite and a pacemaker system [21], [22]. Table I provides information about the benchmarks obtained by applying KLEE without symbolic SWiFI to generate a test suite with full path coverage. It shows the total number of paths (`#Path`), the minimum (`Min`) and maximum (`Max`) number of FI locations per path as well as the total number of FI locations over all paths (`Total`), the lines of code (`LOC`) in C and LLVM-IR for the program (with testbench) and the time in seconds for KLEE to enumerate all paths. Two variants of bubblesort are used which operate on 3 and 4 symbolic array elements, respectively. In the following, we compare the results of `SymET-SymFL` and `PathCovET-SymFL` on these benchmarks. `PathCovET-SymFL` reuses the KLEE-generated test suite.

B. Symbolic SWIFI Results

Table II shows the results of our symbolic SWIFI approaches. We use the previously introduced benchmarks with varying strength of error detectors. For the TCAS program we reuse the functional properties from [23] as error detectors, and similarly for the pacemaker benchmark we reuse specified properties from [22]. The table is horizontally separated in two halves. The left half shows results for the full symbolic `SymET-SymFL`, while the right half shows results for the weaker `PathCovET-SymFL`. For both variants we report the number of explored paths (`#Path`), the number of paths with SDC (`#SDCPath`), the number of instructions executed by KLEE during the analysis (`#Instr`), and the analysis time in seconds.
for detecting all SDC paths (Total) and for reporting the first result (FirstResult). For some benchmarks where no SDC can be detected (i.e. SDCPath = 0), Total and FirstResult are obviously the same.

It can be observed that with symbolic SWiFI, KLEE has to explore significantly more paths. This is, however, to be expected, since in traditional SWiFI, also for each concrete test case a significant number of FI runs must be conducted. The trade-off between PathCovET-SymFL and SymET-SymFL can also be clearly observed. While PathCovET-SymFL cannot find more SDCs than SymET-SymFL, it is considerably faster on the majority of cases. This suggests to apply PathCovET-SymFL first to find SDCs and improve detectors until no SDCs can be found. Then, SymET-SymFL is employed to find more elusive SDC cases or to prove resilience (which PathCovET-SymFL cannot due to its incompleteness). SymET-SymFL was able to prove in reasonable time that the error detectors in npo2 and bubblesort can catch all injected faults. Furthermore, if the user is only interested in the first found SDC, both variants can deliver quickly.

The results also show that more error detectors will generally lead to less SDC. While the error detectors for TCAS and pacemaker are still weak and need further improvements, they are not in the focus of the experiments.

VII. CONCLUSION AND FUTURE WORK

In this paper, we present the first symbolic SWiFI-based approach for resilience evaluation, which performs a full symbolic bit-precise exploration of the combined state space of program behavior and possible faults at the level of compiler’s intermediate code to find silent data corruption (SDC). A more efficient but incomplete variant is also proposed, which can be employed in complementary manner. For a set of benchmarks, both finding FI instances that show “holes” in the detection as well as proving resilience have been shown to be feasible. In the next steps, we plan to apply our techniques to larger programs as well as extend to support more detection (and recovery) mechanisms. To handle large programs, further optimizations might be necessary. In our experiments, we have already observed many SDC paths, that show similar symptoms despite of different fault locations. This suggests it is possible to group/merge these in an intelligent way to speed-up the evaluation.

REFERENCES