Resilience Evaluation via Symbolic Fault Injection on Intermediate Code

Hoang M. Le[1] Vladimir Herdt[1] Daniel Große[1,2] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[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 todays 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. accurary, 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

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project EffektiV under contract no. 01IS13022E and by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1 and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

```
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 }
```

Fig. 1. C source code of *npo2* example

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 literatur. 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 inject faults in the assumption part of a temporal property and applies property checking to obtain counter-examples that show the fault propagation towards property violation. Fey et al. [11] employ Bounded Model Checking techniques to estimate the lower and upper bound of robustness for a design.

On the software level, SWiFi approaches assume that (transient) hardware faults have already propagated as bit-flips into software-visible parts. To the best of our knowledge, the concept of symbolic SWiFi has been first mentioned in [6]. The authors described a case study where they successfully proved that a CRC implementation (in Java) can detect all possible bit-flips at source-code level. However, the proof required considerable human intervention and expertise. In [12], a framework extending Separation Logic to reason about bit-flips at source-code level has been proposed. There was no discussion of whether and how reasoning within this framework can be automated. SymPLFIED [7] is a more practical and automated approach that we consider to be most close to our work. It works on an internal representation termed as *generic assembly language* and considers transient errors in memory/registers, computation and control-logic, which manifest in the architectural state of the processor. SymPLFIED applies over-approximation in fault modeling: no distinction is made between single and multiple bit errors, or in case of single bit-flip, which bit has been flipped. Consequently, SymPLFIED may report *false negatives*, i.e. the symbolic error is propagated to the output but in real executions, any concrete fault is either masked or detected. The false negatives complicate the understanding and subsequent improvement of resilience mechanisms. Moreover, while SymPLFIED employs symbolic FI, it still requires concrete inputs for the program to perform the evaluation.

```
 1 define i32 @npo2(i32 %x) #0 {
 2 entry:
 3   %x.addr = alloca i32, align 4
 4   %result = alloca i32, align 4
 5   %i = alloca i32, align 4
 6   store i32 %x, i32* %x.addr, align 4
 7   %0 = load i32* %x.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 = ashr i32 %3, %4, !dbg !126
17   %or = or i32 %2, %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 %5, 2, !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 }
```

Fig. 2. Excerpt of the LLVM-IR of *npo2* showing entry code and loop body

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 *int*, 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 4 of Fig. 1.

B. SMT-based Symbolic Execution

Let $\mathcal{ET} = (\mathcal{IN}, \mathcal{INSTR})$ be an execution trace of the program under analysis $\mathcal{P}$. $\mathcal{IN} = \{in_1, \cdots, in_n\}$ are the inputs that lead $\mathcal{P}$ to the execution of the ordered instruction list $\mathcal{INSTR} = \{instr_1, \cdots instr_m\}$. 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.

Assuming that $\mathcal{P}$ does not have non-deterministic behavior (e.g. randomization), the ordered instruction list $\mathcal{INSTR}$ 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_ABV fragment of SMT is most often used. We denote $PC(\mathcal{INSTR})$ as the path condition of the instruction list $\mathcal{INSTR}$.

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 SWiFI 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 SWiFI 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 *npo2* in Fig. 2, register *%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 $instr$ in $\mathcal{INSTR}$, assuming that only one single fault can occur, we denote its fault list as $\mathcal{FL}(instr) = \{\cdots, (floc_i, fval_i), \cdots\}$ with $floc_i$ being a fault location within $instr$ and $fval_i$ 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 $\mathcal{INSTR}$. 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 $\mathcal{INSTR}$ is followed.

C. Coverage Strength of Symbolic SWiFI

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 SWiFI 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 SWiFI techniques are the lowest on the coverage strength scale. Symbolic SWiFI 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 SWiFI in the following in increasing order of strength:

1) Using the same set of $\mathcal{ET}$ as traditional SWiFI, the first variant enumerates the fault location *floc* on each concrete $\mathcal{ET}$ but allows the faulty value *fval* to be symbolic. SymPLFIED belongs to this class (although it is not discussed how the set of $\mathcal{ET}$ can be obtained). SymPLFIED also do not distinguish single and multiple bit-flips but over-approximate.
2) The next stronger variant is achieved by allowing also *floc*, and thus the entire state space $\mathcal{FL}$ of each concrete $\mathcal{ET}$, to be symbolic. We are not aware of any existing SWiFi approach with support for symbolic *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 *PathCovET-SymFL* and *SymET-SymFL*, respectively. Furthermore, we do not over-approximate but use bit-precise fault modeling by constraining the symbolic *fval* (more details in Section V-C). The coverage strength of *PathCovET-SymFL* and *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.

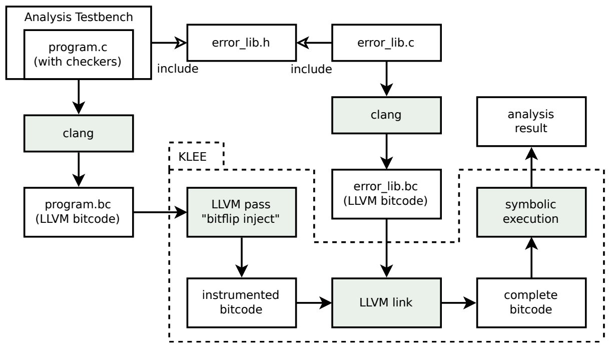

Fig. 3. Overview of our SWiFI approach

```
; original form
%mul = mul nsw i32 %5, 2
store i32 %mul, i32* %i, align 4
; instrumented form: introduce unique register
     (%mul_fi in this case) for each maybe_bitflip
     call and replace all subsequent uses of original
     register (%mul in this case) with it
%mul = mul nsw i32 %5, 2
%mul_fi = call i32 @maybe_bitflip(i32 %mul)
store i32 %mul_fi, i32* %i, align 4
```

Fig. 4. Bit-flip instrumentation example (debug information omitted)

V. Implementation

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

```
// initialize need to be called first once
void error_lib_initialize() {
  error_case = klee_int("error_case");
  klee_assume(error_case > 0);
}
// cut off paths not satisfying *cond*, disable and
     restore local_fi_active (impl. not shown) to
     avoid FI in *cond*
#define CHECK(cond)              \
  interrupt_local_fi_active(); \
  klee_assume(cond);           \
  restore_local_fi_active();
// returns a symbolic int where its binary
     representation differs from x by one bit
int bitflip(int x) {
  int res = klee_int("bitflip_result");
  int t = res ^ x;
  klee_assume(t != 0);
  klee_assume((t & (t - 1)) == 0);
  return res;
}
// non-deterministically inject a bit-flip
int maybe_bitflip(int x) {
  if (global_fi_active && local_fi_active) {
    ++error_counter;
    if (error_counter == error_case) {
      return bitflip(x);
    }
  }
  return x;
}
```

Fig. 5. Excerpt of symbolic SWiFI library

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 libary. 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 *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 non-deterministically 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.

```
int npo2(int x) {
  set_local_fi_active(true);
  CHECK(x > 0);
  CHECK(x < (1 << 30));
  int result = x;
  for (int i = 1; i < sizeof(int) * 8; i *= 2) {
    CHECK(i > 0);
    result = result | (result >> i);
  }
  result = result + 1;
  set_local_fi_active(false);
  CHECK(result <= (x << 1));
  CHECK((result & (result - 1)) == 0);
  return result;
}

int main() {
  error_lib_initialize();
  int num = klee_int("num"); // symbolic input
  set_global_fi_active(true);
  int ans1 = npo2(num); // result with SWiFI
  set_global_fi_active(false);
  int ans2 = npo2(num); // reference result
  assert(ans1 == ans2);
  return 0;
}
```

Fig. 6. Testbench and error detectors for *npo2*

TABLE I
CONSIDERED BENCHMARKS

Program	#Path	#FaultLocation		LOC		Time
		Min / Max	Total	C	LLVM	
npo2	1	50 / 50	50	35	220	0.01
bubblesort_3	6	102 / 129	693	61	444	0.11
bubblesort_4	24	183 / 237	5040	61	457	0.64
tcas	44	54 / 92	3704	244	914	0.21
pacemaker	16	111 / 127	1856	560	2683	0.15

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 minimim (*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 horizontaly 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

TABLE II

SYMBOLIC SWIFI RESULTS - RUNTIME IN SECONDS

Program	SymET-SymFL					PathCovET-SymFL				
	#Path	#SDCPath	#Instr	Time		#Path	#SDCPath	#Instr	Time	
				Total	FirstResult				Total	FirstResult
npo2_with_checks_1	102	22	210115	1.65	0.19	88	16	210101	0.80	0.14
npo2_with_checks_2	85	0	214533	1.76	1.76	85	0	199981	0.72	0.72
bubblesort_3_with_checks_1	1477	151	6955888	23.87	8.57	1471	101	7045887	20.63	8.32
bubblesort_3_with_checks_2	1475	0	9435510	30.06	30.06	1468	0	9112592	22.30	22.30
bubblesort_4_with_checks_1	11817	1832	94533322	581.64	79.62	11502	1071	95973025	252.64	46.16
bubblesort_4_with_checks_2	10869	0	111747753	3971.36	3971.36	11459	0	113120469	256.23	256.23
tcas_without_checks	8264	1472	23498341	64.64	1.35	6682	1098	22550631	57.34	1.07
tcas_with_checks	8264	1274	49282821	169.63	3.16	6818	886	45285172	84.48	1.70
pacemaker_without_checks	3323	894	9314774	15.96	0.51	3286	850	14860331	29.64	1.36
pacemaker_with_checks	3268	740	13206507	28.77	0.47	3110	657	18443082	32.58	1.55

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 resilence 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

[1] J.-H. Oetjens et al., "Safety evaluation of automotive electronics using virtual prototypes: State of the art and research challenges," in *DAC*, 2014, pp. 113:1–113:6.

[2] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "EDFI: A dependable fault injection tool for dependability benchmarking experiments," in *PRDC*, 2013, pp. 31–40.

[3] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *DSN*, 2014, pp. 375–382.

[4] M. Kooli, G. D. Natale, and A. Bosio, "Cache-aware reliability evaluation through LLVM-based analysis and fault injection," in *IOLTS*, 2016, pp. 19–22.

[5] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *DATE*, 2009, pp. 502–506.

[6] D. Larsson and R. Hähnle, "Symbolic fault injection," in *Int'l Verification Workshop in connection with CADE-21*, 2007.

[7] K. Pattabiraman, N. M. Nakka, Z. T. Kalbarczyk, and R. K. Iyer, "SymPLFIED: Symbolic program-level fault injection and error detection framework," *IEEE Transactions on Computers*, pp. 2292–2307, 2013.

[8] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008, pp. 209–224.

[9] U. Krautz et al., "Evaluating coverage of error detection logic for soft errors using formal methods," in *DATE*, 2006, pp. 176–181.

[10] S. A. Seshia, W. Li, and S. Mitra, "Verification-guided soft error resilience," in *DATE*, 2007, pp. 1–6.

[11] G. Fey, A. Suelflow, S. Frehse, and R. Drechsler, "Effective robustness analysis using bounded model checking techniques," *IEEE Transactions on CAD*, vol. 30, no. 8, pp. 1239–1252, 2011.

[12] M. L. Meola and D. Walker, "Faulty logic: Reasoning about fault tolerant programs," in *ESOP*, 2010, pp. 468–487.

[13] H. Cho et al., "Quantitative evaluation of soft error injection techniques for robust system design," in *DAC*, 2013, pp. 1–10.

[14] G. A. Reis et al., "SWIFT: software implemented fault tolerance," in *CGO*, 2005, pp. 243–254.

[15] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *DSN*, 2012, pp. 1–12.

[16] S. Mirkhani, H. Cho, S. Mitra, and J. A. Abraham, "Rethinking error injection for effective resilience," in *ASP-DAC*, 2014, pp. 390–393.

[17] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "On the application of formal fault localization to automated RTL-to-TLM fault correspondence analysis for fast and accurate VP-based error effect simulation - a case study," in *FDL*, 2016, pp. 1–8.

[18] B. Sangchoolie, K. Pattabiraman, and J. Karlsson, "One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors," in *DSN*, 2017, pp. 97–108.

[19] M. Hiller, "Executable assertions for detecting data errors in embedded control systems," in *DSN*, 2000, pp. 24–33.

[20] J. Vinter, J. Aidemark, P. Folkesson, and J. Karlsson, "Reducing critical failures for control algorithms using executable assertions and best effort recover," in *DSN*, 2001, pp. 347–356.

[21] "Pacemaker system specification," Boston Scientific, Tech. Rep., 2007.

[22] A. Sharma, "End to end verification and validation with SPIN," *CoRR*, 2013.

[23] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzé, "Using symbolic execution for verifying safety-critical systems," in *ESEC/FSE*, 2001, pp. 142–151.