Error Propagation Analysis of Multithreaded Programs Using Likely Invariants

Abraham Chan*, Stefan Winter†, Habib Saissi†, Karthik Pattabiraman*, Neeraj Suri†
*Department of Electrical and Computer Engineering, The University of British Columbia, Vancouver, BC, Canada
Email: {*abrahamc, karthikp*}*@ece.ubc.ca*
†Department of Computer Science (Informatik), Technische Universität Darmstadt, Darmstadt, Germany
Email: {*sw, saissi, suri*}*@cs.tu.darmstadt.de*

***Abstract*—Error Propagation Analysis (EPA) is a technique for understanding how errors affect a program's execution and result in program failures. For this purpose, EPA usually compares the traces of a fault-free (*golden*) run with those from a faulty run of the program. This makes existing EPA approaches brittle for multithreaded programs, which do not typically have a deterministic golden run. In this paper, we study the use of likely invariants generated by automated approaches as alternatives for golden run based EPA in multithreaded programs. We present an approach and a framework for automatically deriving invariants for multithreaded programs, and using the invariants for EPA. We evaluate the invariants in terms of their coverage for different fault types across six different programs through fault injection experiments. We find that stable invariants can be inferred in all six programs, although their coverage of faults is highly dependent on the application and the fault type.**

***Keywords*-Error Propagation Analysis; Fault Injection; Reliability; Concurrency; Multithreading;**

I. Introduction

Software fault injection (SFI) [6], [28], [1] is a method for testing the robustness of software components against faults in other components they interact with. For this purpose, SFI introduces faults into components (*targets*) that the software under test (SUT) interacts with, executes the SUT and observes its behavior. Similar to other types of robustness tests, such as fuzzing approaches, that directly expose software components to unexpected inputs, SFI relies on *negative oracles*. Rather than testing the outcome of the SUT's execution against an expected outcome, SFI tests pass if certain undesired behavior, e.g., a segmentation fault, does *not* occur.

Contrary to other robustness testing approaches, however, an observation of the immediate reaction of the SUT to the test input is insufficient for deciding the test results. For fuzzing and other types of tests that directly test via the SUT's interface it is reasonable to assume that the SUT reacts to any given input in a timely manner. SFI tests commonly entail longer latencies, because(1) the injected fault needs to get triggered in the execution of the SUT and its interaction with the target and (2) the resulting corruption of the target's state (that we call *error* in accordance with the Laprie terminology [2]) needs to *propagate* to the interface between the target and the SUT to affect the SUT's behavior.

Predicting the duration of this process is difficult, as it is highly dependent on the target component, the injected fault, and the workload of the SUT that triggers its interactions with the target. This makes the decision on the outcome difficult in cases where undesired behavior is not immediately observed, as it is unknown whether the SUT is robust against an injected fault or its effects have not yet manifested. For example, it would be problematic to assume that a given fault *cannot* result in a segmentation fault, only because it *does not* within the first few seconds of execution.

To cope with this problem, SFI oracles are usually augmented with execution traces that record all executed instructions and the data they operate on. If a test exceeds the timeout without any detection of undesired behavior, its execution trace is compared to an execution trace from a fault-free execution (a so-called *golden run* [4], [14], [21]). From such a comparison, the tester can see(a) to which degree the fault has affected the target's state, (b) if such affected state has led to changes in the interface interactions with the SUT and, if so, (c) to which degree the SUT's state has been corrupted. The identification of how state corruptions evolve from a fault activation in execution is referred to as *error propagation analysis* (EPA). Based on this information either the result can be reliably predicted or the test can be scheduled for re-execution with a longer timeout.

While the golden run comparison technique works well for deterministic programs and execution environments, it can lead to spurious outcomes in the presence of non-determinism, which can cause trace deviations that do *not* indicate error propagation. One of the primary sources of non-determinism is the use of multithreading in programs, which is becoming more prevalent as processors attain increasing core counts. In a multithreaded program, threads can execute in different orders (due to non-deterministic scheduling), and hence the traced values in the program may differ from run to run.

In this paper, we propose the use of dynamically generated "likely" invariants [7] to perform EPA in multithreaded programs. There are three reasons why likely invariants are a good fit for the EPA problem. First, likely invariants can be automatically generated by analyzing the traces from different executions of a program, without any programmer intervention. This is critical for the technique to scale to large, real-world applications. Second, likely invariants are often compact, and can be checked with low overhead at run-time, e.g., as predicates for executable assertions. This makes them easily applicable as oracles. Thirdly, and most importantly, likely

invariants can be refined such that they hold across the entire set of executions on which the program is trained, automatically abstracting out non-deterministic parts of the program. Despite these promising properties, likely invariants characterize correct executions with less precision than true invariants [7], which may reduce their efficacy for EPA.

Consequently, the question we ask in this paper is: "*How effective are the invariants*[1] *generated by automated techniques in tracking error propagation in multithreaded programs?"*. It is important to answer this question to determine if existing invariant generation techniques are sufficient for EPA, or if new techniques need to be developed. We experimentally measure the effectiveness of an invariant in terms of two attributes,(1) the *stability* of the generated invariant set across different (non-deterministic) executions of the program, and (2) the *fault coverage* of the generated invariants for different fault types, corresponding to common software faults.

We make the following contributions in this paper:

- We propose the use of invariants to perform EPA, and establish a suitable terminology for reasoning about the problem of EPA for non-deterministic multithreaded programs (Section III-A).
- We build a framework called IPA (Invariant-based Propagation Analysis) framework to derive dynamic invariants for multithreaded programs through an automated, end-to-end process (Section III-C).
- We assess the efficacy of the invariants derived using IPA for six multithreaded programs through fault-injection experiments. We find that the traditional form of EPA is unsuitable for multithreaded programs. We also find that the invariants are stable across multiple executions, and provide coverage ranging from 10% to 97% depending on the fault type and program. Finally, we find that the invariant type is highly correlated with coverage for a given program (Section IV).

II. Background and Related Work

In this section, we first describe the notions of fault injection and EPA. We then describe likely invariants and related work in the field on using likely invariants.

A. Fault Injection

Fault injection is the technique of modifying one or more components of a software system to emulate bugs. It has been widely deployed to advance test coverage and software robustness by exploring error handling paths of programs (e.g., [18], [8], [9], [10], [26]). There are two categories of software implemented fault injection (SWIFI): compile-time injection and run-time injection. Compile-time injections typically involve modifying source code (e.g., SAFE [28]) or binary code (e.g., G-SWFIT [6] or EDFI [12]), similar to mutation testing. In contrast, run-time injections act as software events that corrupt instructions and dynamic memory spaces. The vulnerability of programs to these bugs are difficult to find through traditional testing techniques [35]. In this paper, we focus on run-time injections, and refer to these as fault injections.

Traditionally, fault injection tools have targeted hardware faults, such as single event upsets caused by particle strikes on chips. However, an increasing number of fault injection systems now target software faults. Fault injection systems, such as FIAT [34], LLFI [24] or PDSFIS [16] explicitly support the emulation of a wide range of software faults at run-time. For instance, buffer overflow errors can be simulated by under-allocating malloc calls by some number of bytes. Other examples include simulating invalid pointer errors by randomly corrupting pointer addresses, and race conditions by acquiring non-existent or incorrect locks.

B. Error Propagation Analysis (EPA)

The effects of a software fault depend on both where it occurs and its type. Therefore, EPA attempts to answer the following question: "How does an injected fault propagate within a program?"

Existing EPA systems in tools such as PROPANE [14] or LLFI [24] make use of either instruction or variable trace comparisons between golden and faulty runs of programs. Deviations between traces can be interpreted as data violations or control flow violations. Data violations occur when identical instructions at the same program point are invoked with different values. Control flow violations occur when the instruction orders differ. Either violation is taken as an indication of a software fault. However, this approach assumes that traces from golden runs are identical, as long as the program is operating on the same inputs. Any non-determinism in the program can violate this assumption, such as that caused by multithreading, which is our focus.

Lemos et. al. [23] have addressed the non-determinism problem in EPA using approximate comparison techniques used in computational biology (e.g., DNA sequencing) to compare golden traces and faulty traces. This approach does not however compare the non-deterministic portions of the trace with the golden run, effectively limiting its coverage. Unfortunately, there is a significant amount of non-deterministic state in multithreaded programs, and hence this approach misses this state.

Leeke et al. [22] attempt to solve the non-determinism problem in EPA using a *reference model*, which is a statistical characterization of the system's outputs. At a high-level, reference models are similar to likely invariants. However, unlike likely invariants, which can be automatically derived, the reference model requires significant manual effort and domain knowledge. Further, for many systems, it may not be possible to derive a reference model if the outputs do not conform to well-known statistical distributions.

C. Likely Invariants

True invariants are predicates that are valid across the set of all executions of a program. Therefore, the violation of a true invariant necessarily indicates the presence of a fault, provided

[1]From this point on, when we say invariants, we mean likely invariants unless we specify otherwise.

the invariant was inferred from a correct program. Thus, true invariants are sound, but not necessarily complete indicators for error propagation. Unfortunately, the existence of such true invariants is undecidable in the general case [31], which makes their automated inference difficult.

Likely invariants in contrast, only hold for observed executions but not necessarily for all executions. Thus, they may contain spurious invariants in addition to true invariants. Further, likely invariants may not comprise all true invariants as some true invariants may not be exercised in the set of observed executions. Thus, likely invariants are both incomplete and unsound in the general case, and hence incur both false negatives *and* false positives.

Although likely invariants, unlike true invariants, bear a risk of false positives, we believe that this risk is substantially lower than for golden run comparisons in non-deterministic programs. This is because EPA is typically done over a set of known inputs, and we only require that the likely invariants are stable over this set. Further, likely invariants can be generated through automated techniques [7], [13], [5], which make them a viable option even for highly complex programs. This is why we use likely invariants in this work.

In this paper, we focus on the likely invariants generated by Daikon [7], which is the most widely used likely invariant inference engine at present. Daikon infers and reports likely invariants based on a set of execution traces. DySy [5] and DIDUCE [13] are other examples of dynamic invariant generation tools. DySy first applies symbolic execution and then observes dynamic execution traces to generate invariants. DIDUCE detects invariants and subsequently checks their violations to help programmers locate bugs. However, all three systems suffer from the effects of thread non-determinism themselves [19], rendering them unsuitable for multithreaded programs. In recent work, Kusano et al. [19] address this problem by developing an additional instrumentation front-end and a custom interleaving explorer for multithreaded programs known as Udon. However, Udon does not concern itself with EPA, which is our focus. Our framework builds on top of Udon for invariant inference.

Prior work has used likely invariants for mutation testing and error detection. For example, Schuler et al. [33] assess the viability of invariant checking in mutation testing. They find that an invariant approach yields a 97% detection rate in their mutation experiments. However, they evaluate the efficacy of invariants through the proportion of equivalent mutants detected (i.e., mutations that yield syntactically different but semantically identical results), which is different from our goal of using them for EPA. Sahoo et al. [32] use likely invariants to detect hardware faults through software-level symptoms. Their experiments show that their approach is able to identify over 95% of hardware faults. However, they focus only on range-based invariants (i.e., checking if values lie in a closed interval), significantly limiting the scope of the approach. Further, they focus on hardware faults (i.e., single bit flips). Lu et al. [25] develop a custom invariant extractor and utilize invariants to expose atomicity violations between thread interleavings. In contrast to these papers, our paper explores the use of a broad set of likely invariants to trace the propagation of software run-time faults in multithreaded programs.

III. Methodology

We first provide an overview of our proposed solution, followed by an example on a multithreaded program. Then, we present IPA, the tool that implements our solution, in Section III-C.

A. Solution Overview

In our approach, we start with a set of correct program executions and generate a set of likely invariants F from them, before we inject a fault and run the program again. The potentially faulty execution is then compared against the likely invariants. Suppose σ denotes an execution of a program and σ_f denotes a faulty execution of the same program. Let s belong to the set of all states reachable by a program execution. A likely invariant f is defined as a predicate over system states s such that $f(s)$ is true for all states reachable in σ. An execution σ_f is said to deviate from the correct runs iff there is an invariant $f \in F$ such that a state s is reachable by σ_f and $f(s)$ is false.[2]

In contrast to the use of golden runs, the invariant based approach for multithreaded programs is neither *sound* nor *complete* as an execution not satisfying a likely invariant does not necessarily indicate an error (soundness), and an error does not necessarily violate a likely invariant (completeness).

To be effective for EPA, the generated invariants must have the following properties, on which we base our empirical assessment in Section IV. The formulas for both metrics are given in Sections IV-D and IV-E.

1) **Stability**: The invariant must hold across multiple fault-free executions of the programs targeted for injection with different numbers of threads for a given set of inputs. This ensures a low false-positive rate.
2) **Coverage**: The invariants must provide high coverage for different types of faults, thereby ensuring a low false-negative rate. We define coverage of an invariant under a certain fault type as the probability that the invariant is violated given that a fault of this type occurs in the program and is activated during program execution.

B. Example

Figure 1 features a function within the Blackscholes application, a multithreaded benchmark program introduced in Section IV-B. A number of invariants can be visually identified by inspection. For the purpose of this example, we will focus on a single pair of likely invariants inferred at the entry and exit points of the function respectively: $\langle InputX > 0$, $InputX = orig(InputX)\rangle$.[3] The latter (exit) invariant only holds when the entry invariant $InputX > 0$ (inferred from a finite number of executions) holds. We pick this pair of likely invariants since they are not trivially related to the

[2]The appendix explains the formal system model in greater detail.

[3]This is similar to the invariants reported by automated invariant inference tools such as Daikon.

```
1  fptype CNDF ( fptype InputX ) {
2      int sign;
3
4      fptype OutputX;
5
6      // Check for negative value of InputX
7      if (InputX < 0.0) {
8          InputX = -InputX;
9          sign = 1;
10     } else
11         sign = 0;
12
13     OutputX = computeNPrimeX(InputX);
14
15     if (sign) {
16         OutputX = 1.0 - OutputX;
17     }
18
19     return OutputX;
20 }
```

Fig. 1: Example function in the Blackscholes application

function output. If the value of $InputX$ is modified after line 13, thus breaking the invariant, the return value $OutputX$ is left unaltered.

Suppose a patch of the program incorrectly alters the boolean expression in line 7 to $InputX > 0.0$ (a mistake even made by experienced programmers [37], [17]). Employing the selected pair of invariants for EPA, the faulty trace can be validated against the invariants to detect the bug. We performed a simple fault injection experiment to test the invariant, and found that that the exit invariant is violated in 100% of faulty runs involving the same inputs on varying numbers of threads. Simultaneously, the entry invariant was upheld across all of those runs.

Violated invariants not only reveal the presence of faults, but also localize the source of faults. Since the entry invariant of the function is retained, and the exit invariant is violated, the fault must have occurred between the entry point and the exit point. Note that these statements are not necessarily from the same function as other threads might have been interleaved with the function and might have modified the value of some of the variables.

This example shows that an invariant based EPA approach can offer both fault detection and fault localization while averting the pernicious effects of thread variance.

C. IPA: EPA Using Likely Invariants

We introduce IPA, a new EPA framework for multithreaded programs using dynamically inferred likely invariants. IPA consists of three main modules, (1) program profiling, (2) invariant inference, and (3) fault detection. Figure 2 shows an overview of the EPA process using IPA.

The *profiling module* (label ①) is invoked at program compilation time, and instruments tracing functions on the entry and exit points of every function in the program. Tracing program values at function entry and exit points allows us to capture preconditions and postconditions of procedures, which broadly encapsulate its functionality. A unique invocation nonce is also assigned to each pair of function entry and exit values, running on the same thread. The invocation nonce enables

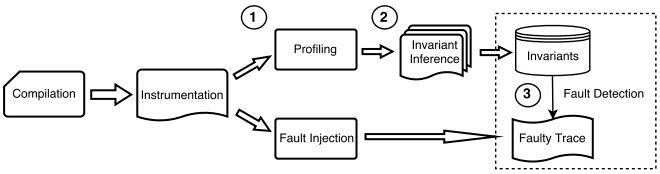

Fig. 2: IPA: Invariant-based EPA Model

inferred invariants to associate exit values with entry values. All of the traced values are accumulated in a trace file, which is passed to the invariant inference module.

The *invariant inference module* (label ②) examines the values in the trace file and generates likely invariants with a 100% confidence, meaning that the invariants will never be falsified within the single trace file. However, the 100% confidence does not guarantee that they are stable for *every* run involving the same inputs. As discussed in Section III-A, this stability across different runs is desired to keep the false-positive rate low. Therefore, programs must be checked to ensure that the set of likely invariants are stable for a given set of inputs. Typically, for terminating programs, this problem can be remedied by using multiple profiling runs to generate the trace file. Traces from multiple program runs can produce fewer invariants than single runs due to the heightened probability for falsification, but can also generate more invariants as larger traces offer higher statistical significance for previously neglected invariants. Once the invariant inference module produces a stable set of invariants, the invariants can be deployed for validation against faulty traces (i.e., traces generated from faulty program runs).

Finally, the *fault detection module* (label ③) parses and groups the invariants by their invoked functions. These invariant groupings are stored in a hash map structure. The faulty trace, which mirrors the format of the golden trace, is scanned line by line. The fault detection module retrieves the corresponding invariant(s) from the hash map and validates the invariant(s) based on the faulty trace values. The invariant violations are reported in a new file, which records the line number in the faulty trace, the function name, a flag indicating function entry or exit, and the violated invariant.

IV. Experimental Validation

The goal of our experimental validation is to evaluate the effectiveness of the likely invariants derived by IPA in performing EPA. As mentioned in Section III-A, to be effective, a likely invariant should have two properties: (1) stability, and (2) coverage. To evaluate the stability, we execute the program multiple times, and measure the number of executions after which the invariant set stabilizes (Section IV-D). We then measure the coverage provided by the invariants for different fault types by injecting faults into the program and checking whether any of the invariants are violated due to a fault (Section IV-E). We also group the invariants into different classes based on their structure, and measure the coverage provided by each class of invariants (Section IV-F).

A. Research Questions

We ask the following research questions in our experimental evaluation.

- **RQ0**: Does golden run EPA correctly detect faults in multithreaded programs?
- **RQ1**: Do the invariants stabilize across multiple executions of the program?
- **RQ2**: What is the coverage provided by the invariants as a whole, for different kinds of errors in the program?
- **RQ3**: What is the coverage provided by invariants of a specific type/class, for different kinds of errors in the program?

B. Experimental Setup

IPA consists of three modules as shown in Figure 2, namely the program profiling module, the invariant inference module, and the fault detection module. The program profiling module is implemented as a LLVM [20] compiler analysis pass, which is based on the instrumentation pass in the Udon tool [19]. The invariant inference module utilizes Daikon [7], since it is presently the most widely used tool for likely invariant generation. Therefore, the primary function of the program profiling module is to produce a trace file in a Daikon-compatible format. For simplicity of implementation, IPA only traces local values of primitive data types – this is similar to what Udon does. Lastly, the fault detection module consists of a single Python script and compares the values in the trace file with the derived invariants.

We evaluate the IPA framework using six multithreaded benchmarks that perform a wide variety of tasks: Quicksort, Blackscholes, Swaptions, Streamcluster, Nullhttpd, and Nbds. These benchmarks range from roughly 300 to 3000 lines of code. All benchmarks are implemented in C/C++, and use the POSIX threading library (i.e., *pthreads*). We run all benchmarks using default program inputs that come with the benchmark suites. Quicksort, as its name suggests, sorts a sequence of integers both sequentially and concurrently using the Quicksort algorithm. Blackscholes, Swaptions, Streamcluster are part of the PARSEC benchmark [3]. Blackscholes is an application that solves the Black-Scholes partial differential equation, which prices a portfolio of European-style stock options. Swaptions uses the Monte Carlo pricing algorithm to compute the prices of swaptions, a form of financial derivative. Streamcluster is a web server application performing the online clustering problem with streaming data. Nullhttpd is a small and efficient multithreaded web server for Linux and Windows [30]. Nbds [29] is an implementation of non-blocking data structures supporting concurrent key-value store transactions. We choose these benchmarks to represent a wide variety of domains.

We use LLFI [24], a LLVM based tool, to perform fault injections. Though LLFI was originally developed for hardware faults, it currently supports both software and hardware faults[4]. LLFI injects software faults into the program IR by modifying instruction or register values of the program at runtime. We assume that faults are uniformly distributed throughout the program code. Table I describes how LLFI injects each software fault. We consider only activated faults, or those in which the modified data is read by the program, when reporting coverage.

TABLE I. Description of faults injected using LLFI

Fault Type	**LLFI Implementation**
Data Corruption	Randomly flips a single bit in an arbitrary data value in the program
File I/O Buffer Overflow	Randomly increases the *size* in *fread* and *fwrite* operations
Buffer Overflow Malloc	Under allocates malloc and calloc to emulate overflowing the allocated buffers
Function Call Corruption	Randomly corrupts the source register (i.e., parameter) of a function call
Invalid Pointer	Randomly corrupts the returned pointer from malloc and calloc
Race Condition	Replaces a lock of a mutex in the program with a fake mutex

In this paper, we consider the following software faults: data corruptions, file I/O buffer overflows, buffer overflows (involving) malloc, function call corruptions, invalid pointers and race conditions. These software faults represent common bugs [36] that are difficult to capture through unit or regression tests, and have been used in prior work to emulate software faults [15], [11]. Data corruption is a generic fault type that can capture a wide variety of errors due to logical errors (e.g., the example in Section III), and implementation bugs (e.g., integer overflows, uninitialized variables). The buffer overflow fault categories can occur due to common bugs in C/C++ programs where array and pointer bounds are not checked. We distinguish between file I/O-related buffer overflows and other buffer overflows as the former can lead to security vulnerabilities. Function call corruptions can occur when one passes the wrong parameters to a function, and represents incorrect invocation of functions i.e., interface errors. Invalid pointers can arise due to errors in pointer arithmetic, or due to the use of pointers after freeing them, i.e., use-after-free bugs. Finally, race conditions occur due to locks not being acquired or acquired incorrectly, and at least one of the threads performing a write to shared data.

C. RQ0: Golden Run Variance

We conduct golden trace analysis (the traditional EPA model) over the benchmark applications (see Section IV-B), varying the number of threads for each program. To conduct EPA following the traditional EPA model shown in Figure 3, the application is compiled and instrumented to invoke a tracing function at every LLVM IR instruction. Hence, each line in a trace file represents an instruction identifier and its corresponding data value in the program. A golden trace of the original program instructions is generated in a process known as profiling. Then, a fault is injected into the program and a trace of the modified program instructions is produced. Finally, EPA is performed by comparing the golden and faulty traces line by line. Discrepancies between the two traces are expected to reveal how faults propagate through the program execution

[4]Available at: https://github.com/DependableSystemsLab/LLFI

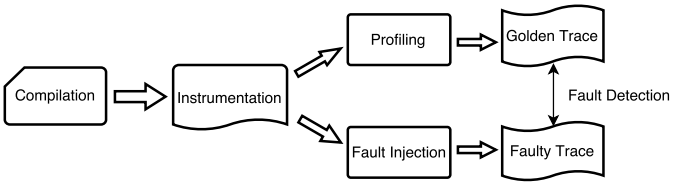

Fig. 3: Golden run based EPA

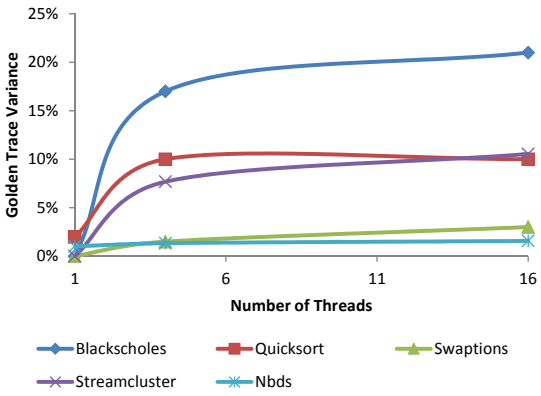

Fig. 4: Average variance between golden run traces

paths.

We collect golden runs over all benchmark programs except *Nullhttpd*, running them with a single thread, 4 threads and 16 threads respectively. We do not consider thread numbers over 16 as most current commodity processors do not have more than 16 cores. We find considerable variance between the golden traces upon running the applications with different numbers of threads using the same input, which obviously do not indicate error propagation. Variance is measured by taking the proportion of line conflicts between two trace files relative to the total number of lines in a single trace file. Figure 4 shows the average variances between 5 golden traces of each application at three distinct thread levels. As can be seen, variance between the golden runs is 10 % on average when a program is run in multithreaded mode.

This experiment was not conducted on Nullhttpd since the thread number was not externally configurable. However, we observed golden run variance when sending multiple server requests concurrently thus spawning multiple threads.

Note that it is possible to use traditional EPA for the deterministic portions of the program. However, it is non-trivial to identify the deterministic portions apriori, as these depend both on the number of threads and the inputs given to the program. Therefore, traditional methods for EPA cannot be used in a multithreaded context.

Observation 1 *If a multithreaded program is repeatedly executed with the same input, the golden runs extracted from these executions differ from each other.*

D. RQ1: Stability

To use invariants for EPA while minimizing false positives, the invariants must be reproducible among repeated program executions. In this experiment, we evaluate the stability of the set of dynamically generated invariants across execution reiterations. Let n denote the number of execution recurrences. Each application begins with $n = 1$ to produce a trace file, which is then delivered to the invariant inference module. The invariant inference module returns a single set of invariants. This process is repeated with $n = 2, 3, 4, 5, 10, 15$, resulting in a family of sets of invariants. The number of invariants obtained at each n value is reported in Figure 5. In all of our sample applications, we observe a convergence of likely invariants by $n = 10$. We have also verified manually that the invariant sets match when the invariants converge, i.e., the invariants derived are the same after 10 executions.

Table II shows the counts of inferred invariants in our sample applications. These are shown only for the stable invariants. We find that there is roughly one invariant for every 10–100 lines of source code. The invariant counts show that stable invariants can be inferred from multithreaded programs, when repeatedly executed with the same inputs.

Observation 2 *If a multithreaded program is repeatedly executed with the same input, the likely invariants generated from these executions stabilize within ten executions.*

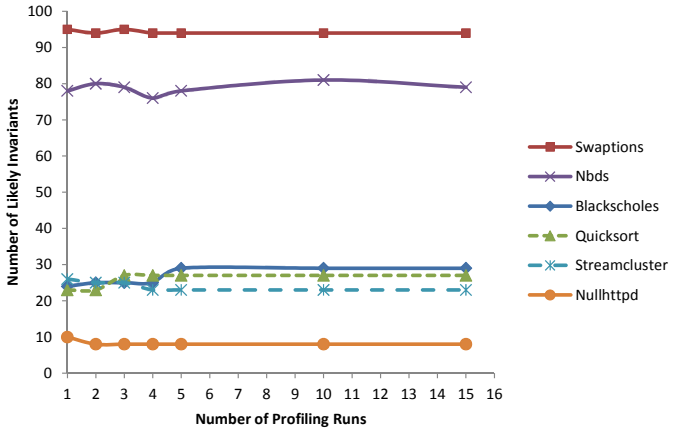

Fig. 5: Number of invariants generated from varying numbers of profiling runs for six benchmark applications

For our coverage assessment in the following section, we consider only the stable invariants, or those invariants that hold across all observed executions (in our experiments). This allows us to minimize the number of false-positives and obtain conservative lower bounds on the coverage.

E. RQ2: Coverage

As we showed in the previous sections, using invariants instead of golden run based comparisons, we were able to improve the soundness of EPA for multithreaded applications, i.e., minimize false positives. An important question is, whether we also miss true positives in the process of reducing false positives, i.e., if the likelihood of false negatives is increased for invariant based EPA. To answer this question, we perform 1000 fault injections of each fault type in Table I, one per run, on the benchmark applications. We choose a sample size of

TABLE II. Invariant counts and classification (refer to Table III) of IPA's generated invariants

Benchmark	LOC	Functions	Invariants	Invariant Classes								
				A	B	C	D	E	F	G	H	Other
Quicksort	330	9	27	3	-	-	-	1	1	16	6	-
Blackscholes	526	5	29	-	-	-	-	3	-	15	11	-
Streamcluster	1580	11	23	1	-	-	-	-	-	14	6	2
Swaptions	1635	14	94	7	4	3	1	4	4	59	11	1
Nullhttpd	2500	20	8	-	-	-	2	-	-	4	2	-
Nbds	3158	27	80	-	-	-	-	4	-	36	39	1

1000 fault injections to reduce the error margins of the fault coverage rates within a 95 % confidence interval. Subsequently, we compare the faulty program traces against the set of inferred invariants. If any of the likely invariants was violated due to the injected fault, we label the run as a successful detection. Suppose T is the set of all faulty program traces, and p is the number of violated invariants in a single trace. Let $T_{p\geq 1}$ be a subset of T, denoting the set of program traces that violate at least one invariant. Then,

$$\text{Fault Coverage} = \frac{|T_{p\geq 1}|}{|T|}$$

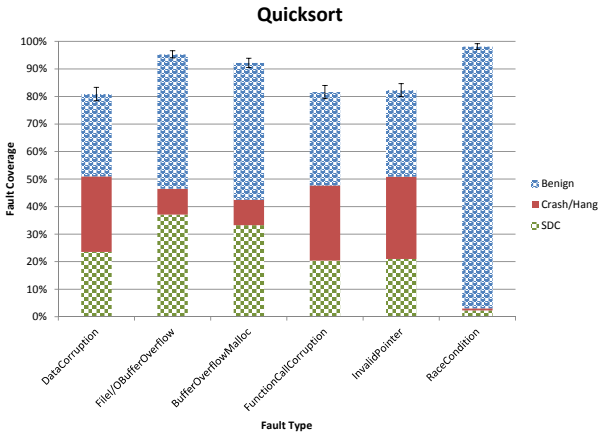

Fig. 6: Proportion of 1000 faulty Quicksort runs that violate at least one invariant

The fault coverages for each application are shown in Figures 6, 7, 8, 9, 11, 10. The error bounds denote the 95 % confidence intervals of the reported fault coverages. The figures show the fault coverage for different fault types divided into three failure modes: Benign, Crash/Hang and Silent Data Corruption (SDC). Benign indicates faulty program runs with no observable deviations in the final program output. Crash/Hang signifies faulty runs that either terminate with exceptions or time out. SDC specifies faulty runs that terminate normally but produce program outputs that deviate from the golden run (i.e., incorrect outputs).

We find that fault coverage provided by the invariants varies widely across applications, from 90 %–97 % for *Swaptions*, to 10 %–15 % for *Blackscholes*. This variation occurs due to variations in two factors: Invariant densities (i.e., number of invariants per lines of code) and invariant relevance (i.e., ability

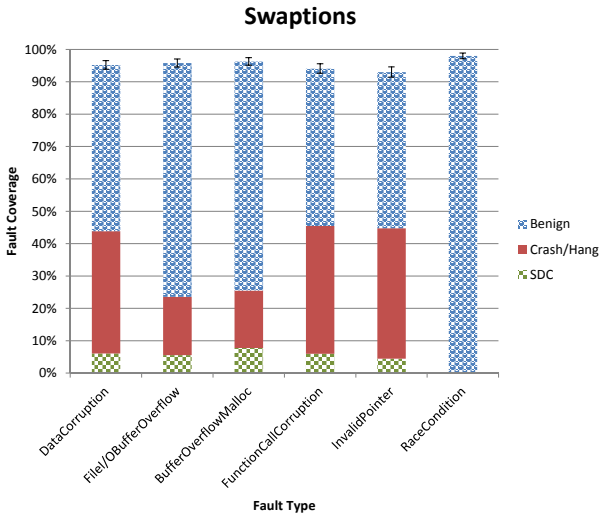

Fig. 7: Proportion of 1000 faulty Swaptions runs that violate at least one invariant

of the invariant to detect faults). Quicksort and Swaptions have higher invariant densities at 8.2 % and 5.7 % respectively. However, invariant density does not express the relevance of the invariants to fault detection. The sets of invariants for Quicksort and Swaptions both contain a number of invariants involving computation data, while Blackscholes is dominated by invariants on local environment variables. Computation data is more likely to be passed inter-procedurally, which increases the likelihood of fault detection. In contrast, local environment variables rarely carry beyond the scope of functions. Consider the case where a variable is corrupted at the function exit. If no invariants exist on that variable at the function exit, the fault would fail to be captured. However, the prospect of fault detection would increase if the value is passed to subsequent functions, which may have invariants checking the value.

Further, there is considerable variation across different fault types and their consequences on the benchmark applications. For example, in *Streamcluster*, the coverage for race conditions is only about 15 %, while it is 70 % for data corruption errors. In other benchmarks (e.g., Quicksort), the situation is reversed, with race conditions having the highest coverage (97 %), while data corruption errors have the lowest coverage (80 %). Data corruption errors directly affect the data as data operand bits are randomly flipped. On the contrary, the effects of race conditions can be difficult to predict as they are dependent on

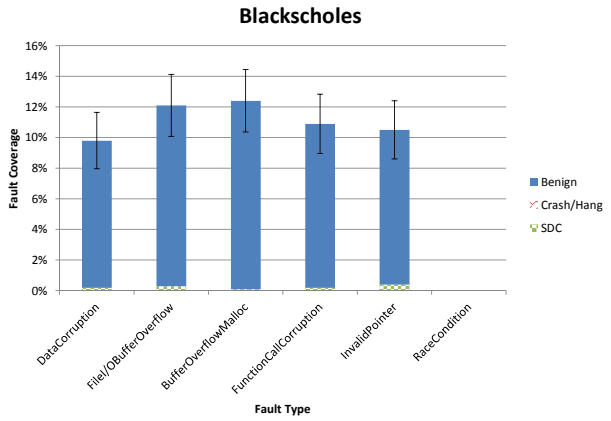

Fig. 8: Proportion of 1000 faulty Blackscholes runs that violate at least one invariant

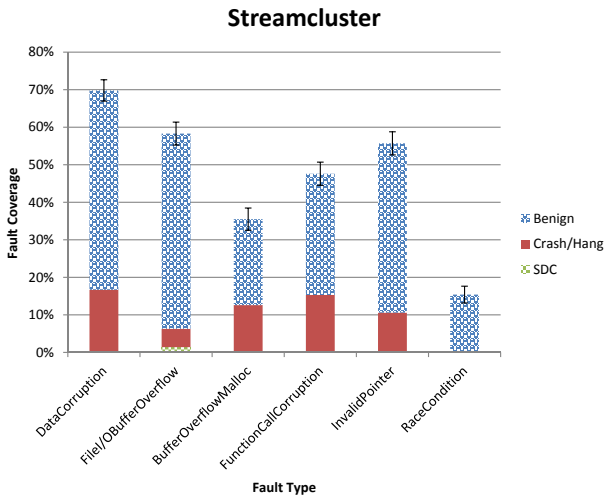

Fig. 9: Proportion of 1000 faulty Streamcluster runs that violate at least one invariant

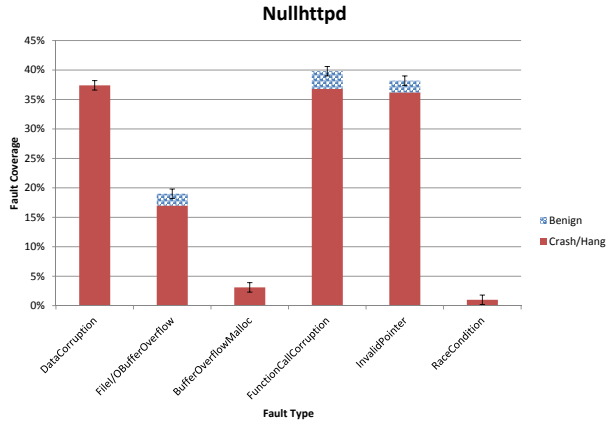

Fig. 10: Proportion of 1000 faulty Nullhttpd runs that violate at least one invariant

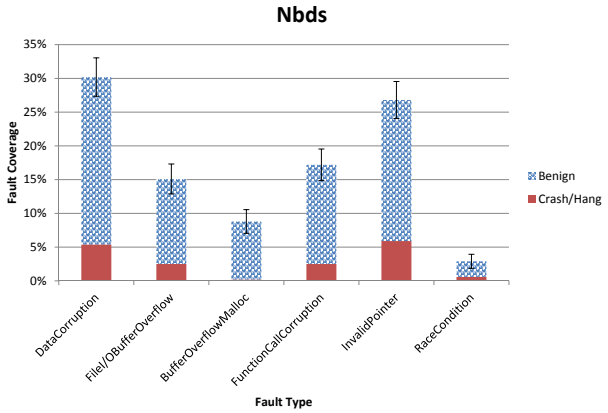

Fig. 11: Proportion of 1000 faulty Nbds runs that violate at least one invariant

the implementation of locking patterns in the threading library. In this case, race conditions cause Quicksort and Swaptions to violate (some) invariants, yet minimal effects are observed in other benchmarks.

Across all applications, benign errors constitute a majority of fault outcomes (73 % on average), followed by Crash/Hang (22 %) and SDCs (5 %). We do not measure SDCs in Nullhttpd and Nbds since the applications return either a successful response code or a failure message. We find that benign errors exhibit the highest fault coverage overall. Although benign errors are typically neglected in EPA, benign fault coverage shows that invariants can track benign faults before they are masked. This may be important to find latent bugs in the program. On the contrary, Crash/Hang are the most blatant failures. Nullhttpd has the highest rate of Crash/Hang fault coverage among the benchmarks. We find that a set of initialization invariants are violated whenever the web server fails to load. Finally, SDCs are typically the least commonly observed failure outcomes across applications, and consequently have the least coverage. Quicksort has the highest rates of SDC error detection among all the applications. This is because it contains many inequalities, and a single negated inequality can impact the final ordering of values. Correspondingly, many of the invariants in Quicksort consist of inequality conditions and ordering constraints that are sensitive to such value deviations, and hence yield high coverage.

Observation 3 *If faults are injected in a multithreaded application, their effects are indicated by violations of likely invariants generated from fault-free multithreaded executions of that application. However, the coverage provided depends both on the application and the type of faults injected.*

F. RQ3: Invariant Classification

During the automated inference of likely invariants, we observed that many of them have similar structure. For example, some invariants involve inequalities, while others involve set membership and ordering. This observation leads us to ask whether differences in structure of the invariants correlate with differences in the respective invariants' effectiveness for EPA.

TABLE III. Description of Invariant Classes

Invariant Class		Description
A	Array-equality	Equality condition on every element of an array
B	Elementwise-initialization	Initial values of array elements
C	Elementwise	Condition on the elements of an array
D	Initialization	Invariants that associate post-conditions to pre-conditions
E	Multi-value	Variable value must match exactly one element of a set
F	Order	Array is sorted in ascending or descending
G	Relational conditions	Invariants involving both equalities and inequalities
H	Return-value	Invariants involving the return value of a function

The results of this study can help us discover what constitutes a good invariant for EPA.

To study this effect, we first classify the invariants into eight different classes based on their structure and then consider the coverage of the invariant classes. The classes are: Array-equality, elementwise-initialization, elementwise, initialization, inequality conditions, multi-value, order, return-value invariants. Table III provides a brief description of each invariant class. The rightmost column of Table II shows the number of invariants per class in each benchmark. The invariants are classified exclusively, without overlap between classes. A small number of invariants did not fall into any of these eight classes – we ignore them for this study.

We calculate the coverage of an invariant class as the fraction of fault injection runs that violate at least *one of the invariants* in that class. For example, if an invariant class I has two invariants I_1 and I_2, and S_1 and S_2 are the sets of fault injection runs that result in violation of the invariants I_1 and I_2 respectively, then the coverage of the invariant class I is given by $(|S_1 \cup S_2|)/N$, where N is the total number of fault injection runs that had activated faults.

Table II shows the number of invariants that occur in different classes for the five applications. Due to space constraints, we only show the results for the Quicksort and Swaptions applications. However, similar results were observed for all benchmarks. Tables IV, V show the results of the fault injection experiment for these two programs, grouped by invariant classes. Note that the figures only show those invariant classes that had at least one invariant in that application as shown in Table II.

We observe that different invariant classes have different coverage depending on the application. For example, in Quicksort (Table IV), order invariants have the highest fault coverage, followed by return-value invariants. Order invariants check whether arrays are sorted in either ascending or descending order. Order invariants are violated if at least one element in an array is misplaced, which accounts for their high fault coverage in Quicksort. A sizeable proportion of faulty runs with violated order invariants result in SDC failures. In comparison,

TABLE IV. Classification of violated invariants from 1000 faulty Quicksort runs and their coverage

Fault Type	Failure	Invariant Classes (%)						
		A	**C**	**D**	**E**	**F**	**G**	**H**
DataCorruption	SDC	6	1	-	-	24	-	24
	Crash	-	1	-	-	27	-	-
	Benign	1	1	-	2	30	-	1
FileI/OBufferOverflow	SDC	8	2	-	2	37	-	37
	Crash	1	2	-	-	9	-	1
	Benign	2	3	-	2	49	-	1
BufferOverflowMalloc	SDC	9	3	1	2	33	-	33
	Crash	1	1	-	-	9	-	-
	Benign	3	3	2	2	50	3	3
FunctionCallCorruption	SDC	7	1	-	1	20	-	20
	Crash	1	2	-	-	27	-	-
	Benign	1	1	-	2	34	-	1
InvalidPointer	SDC	6	1	-	2	21	-	21
	Crash	1	1	-	-	30	-	-
	Benign	1	1	-	2	31	-	1
RaceCondition	SDC	-	-	-	-	2	-	2
	Crash	1	1	-	-	1	-	-
	Benign	1	1	-	-	97	-	-

TABLE V. Classification of violated invariants from 1000 faulty Swaptions runs and their coverage

Fault Type	Failure	Invariant Classes (%)				
		A	**B**	**C**	**D**	**H**
DataCorruption	SDC	3	3	6	1	1
	Crash	-	-	38	-	-
	Benign	26	26	51	-	-
FileI/OBufferOverflow	SDC	3	3	6	1	1
	Crash	1	1	18	-	-
	Benign	42	41	72	-	-
BufferOverflowMalloc	SDC	4	4	8	1	1
	Crash	-	-	18	-	-
	Benign	42	42	71	-	-
FunctionCallCorruption	SDC	2	2	6	1	1
	Crash	-	-	40	-	-
	Benign	26	26	49	-	-
InvalidPointer	SDC	2	2	5	-	-
	Crash	-	-	40	-	-
	Benign	28	28	48	-	-
RaceCondition	SDC	-	-	-	-	-
	Crash	-	-	-	-	-
	Benign	58	58	70	-	-

return-value invariants have a lower overall fault coverage than order invariants. However, we observe that the majority of return-value invariant violations result in SDCs, thus showing their importance.

In Swaptions (Table V), on the other hand, elementwise invariants have the highest fault coverage overall. Elementwise invariants correspond to predicates on individual elements of an array. Swaptions stores its dataset in an array, which is passed back and forth between its functions. As a result, a number of array element constraints arise. Elementwise invariants offer a marginally higher fault detection rate for SDCs compared to the other invariant classes. This contrasts with Quicksort where order and return-value invariants collectively yield high SDC fault detection.

However, there is much less variation in the coverage provided by different invariant classes for different types of faults. For example, in Quicksort, order invariants offer high coverage regardless of fault type, while multi-value invariants

offer uniformly low coverage for this application. Thus, the application and the invariant class have a greater impact on the fault coverage than the fault type, across applications.

Observation 4 *The coverage of invariants for an application differs across different classes of likely invariants generated from fault-free multithreaded executions of the application.*

V. Discussion

In this section, we first present the implications of our results, and then the threats to the validity of our study.

A. Implications

In this paper, we address the question whether likely invariants derived by automated techniques can be used for EPA in multithreaded programs. EPA requires stable invariants, which provide high coverage for different types of faults. We find that the invariants stabilize within a few executions of the program. However, their coverage is highly dependent on the application. For some applications, the coverage provided is high (80 % to 90 %), while for other applications, the coverage is quite low (10 % or less). This suggests that existing invariants derived by automated tools such as Daikon [7] may not be sufficient to ensure high fault coverage across applications. Further, the coverage provided by the invariants depends on the specific fault that is injected, e.g., race conditions. Finally, most of the invariants provide coverage for benign failures and crashes, both of which are much more numerous than SDCs. However, SDCs are an important concern in practice, as they can result in catastrophic failures, and likely invariants do not currently provide high coverage for SDCs. Improving the coverage of likely invariants for SDCs is a direction for future work.

We further study the effect of invariant structure on fault coverage by grouping the invariants into different categories. Here again, we find that there is a significant correlation between invariant structure and fault coverage, but this is more dependent on the application rather than the fault category. However, we find that there is no single class of invariants that provides high coverage across all applications. This implies that it may be better to derive invariants on an application-specific basis, say based on its algorithm, than to use generic approaches such as Daikon for deriving the invariants. This is also a direction for future work.

B. Threats to Validity

There are three threats to the validity of our results. First, IPA uses Daikon for generating likely invariants. Some results may not apply if an alternate approach to likely invariant generation is used, which is an external threat to validity. However, as Daikon is the de facto standard for likely invariant generation in use today, we consider our results valid for most realistic scenarios.

Second, since IPA is limited to tracing local values of primitive data types, the set of generated invariants excludes invariants involving objects and global values. As a result, the invariants deployed for validation are not necessarily the most relevant invariants for the program. This is an internal threat to validity. However, most benchmarks in this study use only primitive data types in their function parameters, and hence this was not an issue in our programs.

Finally, we have defined two metrics to assess the efficacy of IPA, namely stability and coverage. However, there may be other metrics that are more important in other scenarios, for example, the time taken to check an invariant. This was not a concern in our experiments as IPA was intended to be used in the testing and debugging phases when execution time is not as important. However, checking time may be an issue in production settings.

VI. Conclusion

With processors expanding core counts, multithreaded programs are rising in prevalence. Despite this trend, existing methods for EPA that make use of golden traces, are unequipped to handle multithreaded programs. To address this problem, we present an EPA framework using likely invariants in lieu of golden traces, and experimentally evaluate the effectiveness of invariants with respect to their stability, and fault coverage. Our results indicate that invariants can be dynamically derived in all of our benchmark applications, with reasonable stability. However, the fault coverage provided by the invariants is highly variable across applications. Therefore, likely invariants offer a viable replacement for golden-run based EPA only in some applications.

NOTE: We have made our experimental results available at the following URL: *http://goldeninvariants.tumblr.com/*. We will make IPA available under an open source license upon publication of this paper.

References

[1] M. R. Aliabadi, K. Pattabiraman, and N. Bidokhti. Soft-LLFI: A Comprehensive Framework for Software Fault Injection. In *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*, pages 1–5, Nov 2014.

[2] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan 2004.

[3] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[4] J. Christmansson, M. Hiller, and M. Rimen. An experimental comparison of fault and error injection. In *Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on*, pages 369–378, Nov 1998.

[5] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic Symbolic Execution for Invariant Inference. In *Proc. ICSE '08*, pages 281–290, 2008.

[6] J. Duraes and H. Madeira. Emulation of Software Faults: A Field Data Study and a Practical Approach. *IEEE Trans. Softw. Eng.*, 32(11):849–867, Nov 2006.

[7] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly Detecting Relevant Program Invariants. In *Proc. ICSE '00*, pages 449–458, 2000.

[8] C. Fetzer, P. Felber, and K. Hogstedt. Automatic detection and masking of nonatomic exception handling. *IEEE Trans. Softw. Eng.*, 30(8):547–560, Aug 2004.

[9] C. Fu, A. Milanova, B. Ryder, and D. Wonnacott. Robustness testing of Java server applications. *Software Engineering, IEEE Transactions on*, 31(4):292–311, April 2005.

[10] C. Fu and B. Ryder. Exception-Chain Analysis: Revealing Exception Handling Architecture in Java Server Applications. In *Proc. ICSE '07*, pages 230–239, 2007.
[11] A. Ghosh, T. O'Connor, and G. McGraw. An automated approach for identifying potential vulnerabilities in software. In *Proc. IEEE S & P*, pages 104–114, 1998.
[12] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. EDFI: A Dependable Fault Injection Tool for Dependability Benchmarking Experiments. In *Proc. PRDC '13*, pages 31–40, 2013.
[13] S. Hangal and M. S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *Proc. ICSE '02*, pages 291–301, 2002.
[14] M. Hiller, A. Jhumka, and N. Suri. PROPANE: An Environment for Examining the Propagation of Errors in Software. In *Proc. ISSTA '02*, pages 81–85, 2002.
[15] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, Apr. 1997.
[16] A. Jin. A PIN-Based Dynamic Software Fault Injection System. In *Proc. ICYCS '08*, pages 2160–2167. IEEE, 2008.
[17] A. J. Ko and B. A. Myers. Development and Evaluation of a Model of Programming Errors. In *Proc. HCC '03*, pages 7–14, 2003.
[18] P. Koopman and J. DeVale. The exception handling effectiveness of POSIX operating systems. *IEEE Trans. Softw. Eng.*, 26(9):837–848, Sep 2000.
[19] M. Kusano, A. Chattopadhyay, and C. Wang. Dynamic Generation of Likely Invariants for Multithreaded Programs. In *Proc. ICSE '15*, pages 835–846, 2015.
[20] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis transformation. In *Proc. CGO '04*, pages 75–86, 2004.
[21] M. Leeke and A. Jhumka. Evaluating the Use of Reference Run Models in Fault Injection Analysis. In *Dependable Computing, 2009. PRDC '09. 15th IEEE Pacific Rim International Symposium on*, pages 121–124, Nov 2009.
[22] M. Leeke and A. Jhumka. Evaluating the Use of Reference Run Models in Fault Injection Analysis. In *Proc. PRDC '09*, pages 121–124, 2009.
[23] G. Lemos and E. Martins. Specification-guided golden run for analysis of robustness testing results. In *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*, pages 157–166, June 2012.
[24] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman. LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults. In *Proc. QRS '15*, pages 11–16, 2015.
[25] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *Proc. ASPLOS XII*, pages 37–48, 2006.
[26] P. Marinescu and G. Candea. LFI: A practical and general library-level fault injector. In *Proc. DSN '09*, pages 379–388, 2009.
[27] A. Mazurkiewicz. Trace theory. In *Petri nets: applications and relationships to other models of concurrency*, pages 278–324. Springer, 1987.
[28] R. Natella, D. Cotroneo, J. Duraes, and H. Madeira. On Fault Representativeness of Software Fault Injection. *IEEE Trans. Softw. Eng.*, 39(1):80–96, Jan 2013.
[29] Non-blocking data structures. https://code.google.com/p/nbds/. Accessed: 2016-05-17.
[30] Null httpd. https://sourceforge.net/projects/nullhttpd/. Accessed: 2016-05-17.
[31] O. Padon, N. Immerman, S. Shoham, A. Karbyshev, and M. Sagiv. Decidability of Inferring Inductive Invariants. In *Proc. POPL '16*, pages 217–231, 2016.
[32] S. Sahoo, M.-L. Li, P. Ramachandran, S. Adve, V. Adve, and Y. Zhou. Using likely program invariants to detect hardware errors. In *Proc. DSN '08*, pages 70–79, 2008.
[33] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *Proc. ISSTA '09*, pages 69–80, 2009.
[34] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin. FIAT-fault injection based automated testing environment. In *Proc. FTCS-18*, pages 102–107, 1988.
[35] M. Sullivan and R. Chillarege. Software defects and their impact on system availability-a study of field failures in operating systems. In *Proc. FTCS-21*, pages 2–9, 1991.
[36] V. Vipindeep and P. Jalote. List of common bugs and programming practices to avoid them, 2005.
[37] E. A. Youngs. Human errors in programming. *International Journal of Man-Machine Studies*, 6(3):361 – 376, 1974.

APPENDIX

To systematically study the utility of likely invariants for EPA, we first introduce abstract models for sequential programs and parallel programs. Using these models, we demonstrate that the most widely used approach to EPA for sequential programs is not applicable for multithreaded programs. For brevity, we limit our models to terminating programs. We do not consider this a restriction to our argument's generality, as most EPA techniques (like other experimental software assessment) evaluate correctness properties on a finite execution sequence of program statements.

A. EPA in Sequential Programs

We define sequential programs by their control flow graphs (CFGs). A CFG of a program P is a directed graph (V, E). The set of vertices V represents the program statements and the set of directed edges $E \subseteq \{(v_i, v_j) \in V \times V\}$ is defined such that $(v, v') \in E$ iff v' is a *possible direct successor* statement to v. The relation E follows directly from the sequence of statements in the program text and the programming language's semantics. For every statement $v \in V$, we define a set of predecessors $Pred(v) = \{v' \in V : v' \rightarrow v\}$. A program has exactly one entry point e, and a single exit point x such that $Pred(e) = \emptyset$ and $Succ(x) = \emptyset$. A program with multiple exit points can easily be modified to produce an equivalent program with one exit point.

We model an execution of a sequential program as a path in the CFG of the program. Consider an execution $\sigma = e, v_1, ..., x$. We define a state map such that $s_\sigma(v_i)$ is a function assigning values to the program variables at statement v_i. The output of a program is solely determined by the provided input for a sequential program. The *functional specification* of a program relates all possible inputs to corresponding outputs. A program execution whose output satisfies the functional specification is said to be correct. Any deviation of an execution from a correct execution with the same input is called an error. In a program execution, we refer to the sequence between an error and the last output-defining statement as *error propagation*.

In EPA, faults are injected in the considered program to analyze their possible effects. A *fault injection* procedure consists of either adding or modifying a statement or its data, the *injection point*, in the CFG. Given a concrete input, the program is executed to obtain a correct execution, referred to as the *golden run*. Next, a fault is injected and the program is executed again with the same input. The obtained execution may deviate from the golden run as the code has been modified. If so, the fault is activated, resulting in an error, and one can analyze the faulty execution. Error propagation can be identified based on whether (1) the faulty execution σ_f follows a different path in the CFG compared to the golden run σ_g starting from the injection point v_r, OR (2) there exists a statement v in σ_f occurring after v_r, such that $s_{\sigma_f}(v) \neq s_{\sigma_g}(v)$.

B. EPA in Multithreaded Programs

Multithreaded programs consist of multiple threads executing concurrently. Each thread T_i is modeled as a separate CFG E_i. For a statement v in CFG E_i, we write $th(v)$ to refer to the thread T_i executing it. An execution of a concurrent system is a sequence of statements $\sigma = e_i, v_1, v_2, ..., x_j$ such that for every two successive statements v and v' with v occurring before v' and $th(v) = th(v')$, $vis \in Pred(v')$. In other words, an execution is a linearization of partial orders of statements induced by the respective CFGs.

Problem Statement: Due to the non-determinism of scheduling, different "equivalent" linearizations are possible. Given the same input, two executions σ and σ' are said to be equivalent iff they deliver the same output, that is, $s_\sigma(x_i) = s_{\sigma'}(x_j)$. Given a golden run $\sigma_r = ...v_i, v_j, ...$, one can obtain an equivalent linearization $\sigma'_r = ...v_j, v_i, ...$ by swapping adjacent statements v_i and v_j, as long as the CFG order and synchronization mechanisms allow it, if $s_{\sigma_r}(v_j) = s_{\sigma'_r}(v_i)$. Successive swapping of such statements generates more possible linearizations that can characterize a correct execution [27]. This is the reason why straightforward pairwise comparison of statements in the executions introduces unsoundness to golden run based EPA for multithreaded systems. Thus, golden run based EPA may erroneously flag a deviation of observed equivalent linearizations due to to scheduler non-determinism.

C. Likely Invariants

Likely invariants are predicates that are defined over states of the program. More formally, a likely invariant f of a set of executions Σ is defined such that $f(s)$ evaluates to true, for every state s reachable by an execution in Σ. Note that likely invariants are different from true invariants for which Σ contains the set of all possible executions of the program.