TraceSanitizer – Eliminating the Effects of Non-determinism on Error Propagation Analysis

Habib Saissi, Stefan Winter, Oliver Schwahn
Technische Universität Darmstadt
Darmstadt, Germany
{saissi, sw, os}@cs.tu-darmstadt.de

Karthik Pattabiraman
The University of British Columbia
Vancouver, Canada
karthikp@ece.ubc.ca

Neeraj Suri
Lancaster University
Lancaster, UK
neeraj.suri@lancaster.ac.uk

***Abstract*—Modern computing systems typically relax execution determinism, for instance by allowing the CPU scheduler to interleave the execution of several threads. While beneficial for performance, execution non-determinism affects programs' execution traces and hampers the comparability of repeated executions. We present *TraceSanitizer*, a novel approach for execution trace comparison in Error Propagation Analyses (EPA) of multi-threaded programs. TraceSanitizer can identify and compensate for non-determinisms caused either by dynamic memory allocation or by non-deterministic scheduling. We formulate a condition under which TraceSanitizer is guaranteed to achieve a 0% false positive rate and automate its verification using Satisfiability Modulo Theory (SMT) solving techniques. TraceSanitizer is comprehensively evaluated using execution traces from the PARSEC and Phoenix benchmarks. In contrast with other approaches, TraceSanitizer eliminates false positives without increasing the false negative rate (for a specific class of programs), with reasonable performance overheads.**

I. Introduction

To maximize resource utilization and system throughput, computing systems often relax the determinism of program executions provided it does not affect the program's functionality. A prominent example are preemptive CPU schedulers that dynamically assign CPUs to executable programs and revoke such assignments at any point of the programs' executions. Similarly, dynamic memory allocators, which assign memory to a program upon request, have the freedom to decide at which memory address the requested memory region is located.

EPA analyzes how software faults affect program control and data flow at run time. It has many uses such as error detector placement [1], [2] and robustness testing [3]. EPA is typically performed by injecting faults into the program, and comparing the fault-affected (faulty run) against the fault-free (golden run) *execution traces*, i.e., records of which program instructions have been executed in which order.

Unfortunately, EPA is adversely affected by relaxing execution determinism [4], [5] as instructions from different threads can appear in different orders, and referenced memory addresses may differ. Such deviations between fault-free and fault-containing traces, which are caused by relaxed execution determinism, constitute false positives in EPA since they do not indicate the effects of actual faults, but only occur to benign execution non-determinism.

Deterministic replay techniques [6] can eliminate the deviations due to non-determinism across program executions. However, they potentially render EPA results invalid, as the comparison of executions in EPA is not across identical copies of a program, but across an original and a mutated version. For example, if CPU schedules are affected by time-intensive operations introduced by the mutation, enforcing the original schedule can lead to false conclusions in EPA.

In this paper, we propose an automated technique to perform execution trace comparisons for EPA in the presence of execution non-determinism, without resorting to deterministic replay-like techniques. We address both memory and scheduling non-determinism and find the latter to be significantly more difficult to handle because of possible inter-thread data dependencies, i.e., concurrent accesses to a shared memory object with at least one write. Furthermore, non-deterministic scheduling decisions directly impact the data values and the instructions seen by each thread, in the presence of data-dependent instructions between threads. Therefore, the order in which these instructions are observed allows deviations of the values and instructions in the trace, and the number of correct golden runs for a program grows exponentially with the number of threads executing data-dependent instructions.

The main insight in our work is that EPA in the presence of non-determinism, while being very challenging in the general case, becomes solvable for a specific but important class of programs that satisfy two *conditions*: (1) exhibit identical externally observable behavior across repeated executions, and (2) are not affected by non-deterministic external functions.

The first condition excludes both programs with data races and intentionally non-deterministic programs. For the latter, reference executions generally cannot serve as an oracle, and hence no differential testing technique (including EPA) is suited for these types of programs. Races are more problematic, as they can result in non-deterministic behavior of intentionally deterministic programs, i.e., it is difficult to know apriori if a program meets our postulated condition. To determine whether a program meets this condition, we introduce an automated test for data races and order violations based on maximal causality models [7] derived from a reference execution (Section IV-C3).

The second condition excludes programs that deliberately make use of non-deterministic external libraries (e.g., random number generators) because these pose a similar problem as intentionally non-deterministic programs. Hence, this condition only imposes an additional constraint on externally deterministic programs, i.e., the second condition excludes programs that process non-deterministic data, but for which this non-deterministic data has no effect on the program's externally observable behavior. A corresponding check could be implemented via a black-listing mechanism for such external calls.

We term programs that satisfy the above conditions to be pseudo-deterministic, e.g., programs following the MapReduce paradigm, where a master thread distributes partitioned data chunks to worker threads that process these chunks identically and report the results back to the master thread. The worker threads' operations are independent since they operate on disjoint chunks of data. While the master thread does interact with the worker threads, this interaction always follows the same pattern leading to the same behavior of each thread. Consequently, programs that follow this pattern, which is often referred to as data parallelism (as opposed to task parallelism) or SPMD (single program, multiple data), are pseudo-deterministic. SPMD is considered the dominant style of parallel programming in NIST's Dictionary of Algorithms and Data Structures [8], and has been identified as the most common pattern in the usage of parallel libraries [9]. Therefore, we expect many parallel programs to satisfy the pseudo-deterministic condition.

Prior work on performing EPA for non-deterministic traces either skips the non-deterministic parts of the trace [10], or uses statistical properties and likely invariants to capture the non-determinism [11], [12]. The former techniques may miss error propagation in important parts of the execution. The latter techniques are unsound, as they may classify deviations from the invariants or statistical measures as errors, although they are legitimate behaviors. *To the best of our knowledge, ours is the first technique to perform EPA for (a class of) non-deterministic programs that is (1) is sound, (2) covers error propagation in non-deterministic parts of the execution, and (3) requires neither programmer support nor annotations.*

Contributions. We make the following contributions:

- Develop a novel reversibility check based on SMT solving techniques to reliably identify pseudo-deterministic programs for which EPA is sound despite relaxed execution determinism.
- Introduce a trace sanitizing approach for *pseudo-deterministic* multi-threaded programs that abstracts away the non-determinism due to both dynamic memory allocation and non-deterministic scheduling.
- Implement our trace sanitizing algorithm in TraceSanitizer, a trace comparison tool for EPA of multi-threaded programs, and evaluate its effectiveness on a set of five widely used benchmarks. We show that TraceSanitizer reduces the rate of false positives to 0 % and achieves a high fault coverage, at a reasonable performance overhead.

II. Related Work

Deterministic Execution. The effects of non-determinism due to multi-threading can be mitigated through the use of deterministic execution. Examples of this approach are Dthreads [13] and Deterministic Parallel Java (DPJ) [14]. These approaches work by constraining either the set of possible interleavings that the OS scheduler interposes on the program or the set of possible programming language constructs that the developer is allowed to use. The former imposes performance overheads as the scheduler has less flexibility in ordering the program's threads to optimize for performance, while the latter imposes a burden on the programmer as they need to ensure their program is free of the "problematic" constructs.

Error Propagation Analysis (EPA). EPA has traditionally been performed by comparing the faulty execution trace to a golden run (i.e., fault-free execution) of the program [15], [1]. Most papers in this area assume that a fault-free golden run trace is deterministic and hence perform a simple line-by-line comparison of a fault-injected run with the golden run [11]. Unfortunately, this is not the case for multi-threaded programs.

To our knowledge, there have been only three approaches that have attempted to address the issue of non-deterministic golden traces for EPA. First, DeLemos et al. [10] used biological sequence alignment algorithms to compare non-deterministic golden traces with faulty executions, effectively skipping the non-deterministic sections of the trace. The underlying implicit assumption is that most parts of the trace are deterministic, and hence skipping the non-deterministic portions is acceptable. Unfortunately, this need not be the case for multi-threaded programs as the OS scheduler has considerable freedom to vary the thread interleaving and memory ordering from one execution to another. Second, Leeke et al. [11] attempted to characterize a golden run using statistical techniques such as clustering, and perform a coarse-grained comparison of the faulty run with reference to these statistical characteristics. Only if there is a significant deviation in the characteristics do they consider it as an erroneous execution. However, their approach requires significant manual intervention to annotate the clusters, and also requires that the system's outputs conform to well-known statistical distributions. Further, they may not detect subtle errors that violate the event orderings of the program unless the errors result in significant deviations from the characteristics. Finally, Chan et al. [12] used dynamic invariants to characterize a non-deterministic golden run, and consider any execution that violates the invariants as an erroneous execution. This approach is, however, unsound as the invariants are only *likely* invariants extracted using Daikon [16].

Unlike such state of the art approaches, our goal is to develop a sound EPA approach in the presence of non-determinism arising from multi-threading in programs. We do not attempt to constrain the set of execution orderings imposed by the OS scheduler nor do we constrain the language features used by the programmer. Furthermore, our approach incurs low performance overheads and requires no programmer effort.

III. Motivating Example

While the performance-driven relaxation of execution determinism does not affect the correctness of a program execution, it may affect the execution trace recorded from that execution. In that case, a direct comparison of such non-deterministic traces for EPA leads to false positives. To illustrate this problem

```
1  #include <stdio.h>
2  #include <pthread.h>
3  int arr[2];
4  void *inc(void* arg) {
5    arr[0]++;
6    pthread_exit(NULL);
7  }
8  void *dec(void* arg) {
9    arr[1]--;
10   pthread_exit(NULL);
11 }
12 void main(int argc, char **argv) {
13   pthread_t id1, id2;
14   arr[0] = 3;
15   arr[1] = 6;
16 
17   pthread_create(&id1, NULL, inc, NULL);
18   pthread_create(&id2, NULL, dec, NULL);
19   pthread_join(id1, 0);
20   pthread_join(id2, 0);
21   printf("Result: %d\n", arr[0]+arr[1]);
22 }
```

Fig. 1. Example multi-threaded program.

```
0  0 call-main 0 1 7ffcfe3287e8
...
1  0 alloca 7ffcfe3282e8 8
2  0 alloca 7ffcfe3282e0 8
...
3  0 store 3 603d74
4  0 store 6 603d78
5  0 call-pthread_create 0
       ↪ 7ffcfe3282e8 0 400ae0 0
6  0 call-pthread_create 0
       ↪ 7ffcfe3282e0 0 4012c0 0
7  1 call-inc 0
8  1 alloca 7f0ccbc55d58 8
9  0 load 7f0ccbc56700 7ffcfe3282e8
10 1 alloca 7f0ccbc55d50 8
11 1 store 0 7f0ccbc55d50
12 1 load 3 603d74
13 2 call-dec 0
14 2 alloca 7f0ccb454d58 8
15 1 store 4 603d74
16 2 alloca 7f0ccb454d50 8
17 2 store 0 7f0ccb454d50
18 2 load 6 603d78
19 0 call-pthread_join 0 7f0ccbc56700 0
20 0 load 7f0ccb455700 7ffcfe3282e0
21 2 store 5 603d78
22 0 call-pthread_join 0 7f0ccb455700 0
...
```

```
0  0 call-main 0 1 7ffda8e0e598
...
1  0 alloca 7ffda8e0e098 8
2  0 alloca 7ffda8e0e090 8
...
3  0 store 3 603d74
4  0 store 6 603d78
5  0 call-pthread_create 0
       ↪ 7ffda8e0e098 0 400ae0 0
6  0 call-pthread_create 0
       ↪ 7ffda8e0e090 0 4012c0 0
7  0 load 7fd5571d9700 7ffda8e0e098
8  1 call-inc 0
9  1 alloca 7fd5571d8d58 8
10 1 alloca 7fd5571d8d50 8
11 1 store 0 0 7fd5571d8d50
12 1 load 3 603d74
13 1 store 4 603d74
14 2 call-dec 0
15 2 alloca 7fd5569d7d58 8
16 2 alloca 7fd5569d7d50 8
17 2 store 0 7fd5569d7d50
18 2 load 6 603d78
19 2 store 5 603d78
20 0 call-pthread_join 0 7fd5571d9700 0
21 0 load 7fd5569d8700 7ffda8e0e090
22 0 call-pthread_join 0 7fd5569d8700 0
...
```

Fig. 2. Execution traces from two executions of the program in Figure 1.

in multi-threaded programs, Figures 1 and 2 show the effects of memory allocation and thread scheduling non-determinism.

Figure 1 is a typical MapReduce-like program, an important class of programs that fulfills our first definition criterion of pseudo-determinism. It defines a global array `arr` (line 3) to store the data to be processed, initializes its contents (lines 14 and 15) and then spawns two threads (lines 17 and 18) that independently operate on different partitions of the data. The initial thread waits for the two worker threads to return (lines 19 and 20) before it aggregates the results from their operations by printing the sum of the array elements (line 21).

Figure 2 depicts two shortened execution traces recorded from repeated executions of that program. The traces were recorded using the EPA framework LLFI [17], [18] and contain one line for each executed instruction of the program's LLVM intermediate representation (IR). The line starts with the index of the instruction in the trace. The second number is a (simplified) ID of the executing thread, followed by the instruction's name, and its return and operand values.

Despite being functionally identical, an EPA on the two traces would identify them as deviating because of differing memory addresses, i.e., any number with more than one digit in Figure 2. Moreover, the different interleaving of instructions from different threads causes EPA to falsely identify deviations between the two execution traces. For instance, while the instructions from thread 1 and 2 are interleaved in the first trace, they are executed in groups in the second trace. Note that while the threads share global memory locations (a source of dependency), the accesses are not concurrent in any interleaving. For instance in Figure 1, although the main thread and the first thread access the first slot in `arr` at lines 5 and 21, there can be no other execution of the program where line 21 is executed before line 5 due to the explicit synchronization call `pthread_join` (line 19).

The goal of TraceSanitizer is to transform these traces in a way that preserves *functionally relevant* deviations, e.g., deviating variable values, and eliminates *functionally irrelevant* deviations, e.g., deviating addresses of memory objects representing these variables. TraceSanitizer leverages the explicit synchronization in multi-threaded programs to simplify the comparison in EPA.

In summary, TraceSanitizer addresses two sources of execution non-determinism that cause spurious trace deviations.

1. Non-deterministic memory allocations: For portability reasons, programs should not make assumptions about memory layout, and leave memory management entirely to the OS. Consequently, the addresses of memory objects that programs operate on should be irrelevant to the program's functionality, and should not distort execution trace comparisons for EPA, nor any other analysis reasoning about the program's functionality.

2. Non-deterministic thread scheduling: To maximize CPU utilization and thereby improve throughput, the CPU scheduler may suspend threads that execute blocking instructions, e.g., when waiting for I/O or lock access, and schedule another thread. The decision of which thread is executed after some other thread has been suspended is dynamically made by the CPU scheduler at run time, and may differ across repeated program executions depending on system load and other factors. As a result, the sequence of instructions in the execution trace can deviate across repeated executions. For *thread safe* programs, these deviations do not affect their functionality and should not affect trace comparisons. A deviation in the order of instructions in an execution trace does not necessarily result in non-deterministic values read or written by the program (as shown in Figure 2). In this case, re-executing the program might result in a different interleaving of instruction, but still lead to the same effects on the program's data, which holds especially for programs that implement the MapReduce paradigm.

The problem of whether a program is thread safe is outside the scope of this work, and is covered elsewhere [19], [7].

IV. Sanitizing Algorithms

We present a novel approach to address non-deterministic memory allocation and thread scheduling in EPA. The core idea behind our approach is to leverage the structure of a pseudo-deterministic programs to apply two trace sanitizing algorithms,

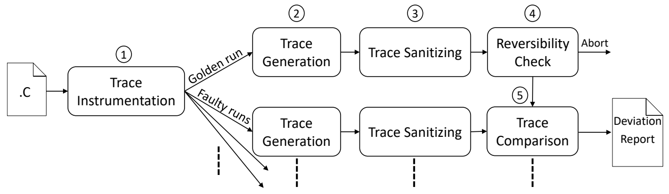

Fig. 3. Overview of TraceSanitizer.

each dealing with a specific source of non-determinism. We first introduce the notion of pseudo-deterministic traces and describe a corresponding automated reversibility check. We then describe TraceSanitizer, our prototype implementation, and show how it soundly compares traces.

A. Overview

Figure 3 overviews TraceSanitizer's workflow. To obtain a fault-free execution trace (golden run), we first instrument the program to log the executed instructions (step ①) and generate a trace by running the instrumented program ②. These first two steps are fundamental building blocks of EPA and we can reuse the existing implementation of LLFI EPA tool [17], which we only slightly modify to deal with multi-threaded programs and include more data in the traces. Next, we run our trace sanitizing algorithms on the generated trace in step ③. We then run the reversibility check to verify whether the generated trace satisfies the pseudo-deterministic condition. If the condition holds, the comparison of its traces is guaranteed to be free from false positives induced by scheduling non-determinism. In case it does not satisfy it, we abort the process. It is important to note that the reversibility check is run only once on the fault-free trace and its results are valid for comparison with any faulty trace given the same inputs. To perform EPA, we inject multiple faults into the instrumented program, re-run steps ② - ③ to generate faulty execution traces (faulty runs), and compare that trace against the fault-free trace (step ⑤) to identify how the program execution has been affected by each injected fault.

We introduce our notation and the pseudo-deterministic condition in Section IV-B, followed by the sanitizing algorithms and the reversibility check in Section IV-C.

B. System Model

We adopt a general and simple model to describe execution traces of a multi-threaded program. An execution trace is a sequence of events $\sigma = e_1, e_2, ..., e_n$. An execution trace is said to be feasible for a program if its sequence of events follows an order that is allowed by the program's semantics. Every event can be directly mapped to an executed instruction such as spawning a new thread or synchronizing with other threads. For simplicity, we focus on read and write events. We write $e \in \sigma$ for any event that has been executed by σ. To refer to the total order of events incurred by a trace σ, we write $e_i \prec_\sigma e_j$ if $i < j$ and $e_i, e_j \in \sigma$. We refer to the thread that executed an event e as $Tid(e) \in T$ where T is the set of threads that are spawned during execution.

We define a binary dependency relation D between events based on the memory objects they access. Two events e and e' are said to be dependent if they both access the same object o and at least one of them is a write event. In that case, we write $(e, e') \in D$. We write D_{tr} to refer to the transitive closure of D. That is, if $(e_i, e_j) \in D_{tr}$ and $(e_j, e_k) \in D_{tr}$ then $(e_i, e_k) \in D_{tr}$, and if $(e_i, e_k) \in D$ then $(e_i, e_k) \in D_{tr}$.

For our sanitizing algorithms to be sound and to result in a false-positive free EPA, the considered execution traces need to satisfy the pseudo-deterministic condition.

***Definition** 1 (pseudo-deterministic traces):* A trace $\sigma = e_1, e_2, ..., e_n$ is said to satisfy the pseudo-deterministic condition if and only if:

1) for every event $e \in \sigma$, the next event executed by $Tid(e)$ and the value it reads or writes are solely determined by the events $e' \prec_\sigma e$ such that $(e', e) \in D_{tr}$ or $Tid(e) = Tid(e')$ (local determinism), and
2) for every two dependent events $(e_i, e_j) \in D$ such that $e_i \prec_\sigma e_j$, there is no other feasible interleaving of the trace σ' where $e_j \prec_{\sigma'} e_i$ (reversibility).

The local determinism condition excludes programs with inherent non-deterministic behavior. The nature of an event (control flow) and the value it reads/writes (data flow) is solely determined by the events it depends on or events executed by the same thread. Threads in a multi-threaded program act according to the data they read. Intuitively, the next instruction to be executed by each thread and how it modifies the program's data depends the values it has read and its program counter position. That is, given two interleavings of a program, for a certain thread t at a certain program counter position, if all the values read by t so far are the same, the next instruction to be executed by that same thread and the value it reads or writes are guaranteed to be the same in both interleavings. For instance, the value generated by invoking a random number generator is neither determined by the events it depends on nor the events executed by the same thread. We refer to the subsequence of events that fully specify an event as its determining events. For a trace $\sigma = e_1, e_2, \ldots, e_n$ and event $e_i \in \sigma$, its determining events subsequence σ_{e_i} contains only events $e_j \prec_\sigma e_i$ such that $(e_j, e_i) \in D_{tr}$ or $Tid(e_j) = Tid(e_i)$. Thus, given a feasible interleaving σ' of σ such that $\sigma_e = \sigma'_e$ for a common event e, the value written/read by e is guaranteed to be the same.

A trace satisfies the reversibility condition if there can be no interleaving σ' where two dependent events $(e_i, e_j) \in D$ occur in a reversed order. This implies that $\sigma'_{e_j} \neq \sigma_{e_j}$ since $e_i \notin \sigma'_{e_j}$ and $e_i \in \sigma_{e_j}$. In this case, it is possible for event e_j to read/write a different value (data deviation). Thus, different data values may be observed over repeated executions.

C. Algorithms

If a trace satisfies the pseudo-deterministic condition, memory addresses are allocated in the same order by each thread. The threads are spawned in the same order by the same parent threads for any feasible interleaving of the trace. In this section, we present our two sanitizing algorithms: 1) A memory

Algorithm 1: Memory abstraction algorithm.

```
input  : Execution trace $\sigma$
output : Set $O$ of symbolic memory objects, such that each
         concrete memory object referenced in $\sigma$ is mapped
         to exactly one symbolic memory object
1  $O \leftarrow \emptyset$;
2  foreach $e \in \sigma$ do
3      $g\_idx \leftarrow e$.getGlobalIndex();
4      $t \leftarrow e$.getThread();
5      if $e$.isAllocation() then
6          $size \leftarrow e$.getSize();
7          $bAddr \leftarrow e$.getBaseAddr();
8          $val \leftarrow [g\_idx, \_]$;
9          $l\_idx \leftarrow e$.getLocalIndex();
10         $s \leftarrow e$.isStackAllocation();
11         $o \leftarrow (bAddr, t, l\_idx, size, val, s)$;
12         append $o$ to $O$;
13     if $e$.isDeAlloaction() then // for heap objects
14         $o \leftarrow$ getObject($e$.getBaseAddr());
15         $o$.updateValidity($g\_idx$);
16     if $e$.isNewScope() then
17         $sp[t] \leftarrow g\_idx$;
18     if $e$.isExitScope() then // for stack objects
19         foreach $o \in O$ s.t. $o$.getThread()
             $= t \wedge o.s \wedge o$.getValidityStart() $> sp[t]$ do
20             $o$.updateEndValidity($g\_idx$);
21         $sp[t] \leftarrow$ restoreStackPointer();
```

Algorithm 2: Thread abstraction algorithm.

```
input  : Execution trace $\sigma$
output : A map $M$ of concrete thread IDs in $\sigma$ to canonical
         IDs that are deterministic across traces from
         repeated executions, as long as the
         pseudo-deterministic condition holds
1  $M \leftarrow \langle\rangle$;
2  $Q \leftarrow \emptyset$;
3  $T \leftarrow \sigma$.getAllThreads();
4  $G \leftarrow (T, \emptyset)$;
5  foreach $t \in T$ do
6      foreach $t' \in t$.getSpawnedThreads() do
7          append $(t, t')$ to $G$;
8  $t_c \leftarrow G$.getRootNode();
9  $M[t_c] \leftarrow$ "T_0";
10 push $t_c$ to $Q$;
11 while $Q \neq \emptyset$ do // breadth-first search
12     $t_c \leftarrow Q$.pop();
13     $i \leftarrow 0$;
       // Get the spawned threads in their order of creation.
14     foreach $t \in t_c$.getOrderedChildren() do
15         $M[t] \leftarrow M[t_c]$"_"$i$;
16         push $t$ to $Q$;
17         $i \leftarrow i + 1$;
```

object abstraction algorithm that deals with memory allocation non-determinism by tracking the order in which memory addresses are allocated to achieve a canonical naming where every object is uniquely identified by its position in the sequence of allocated objects (Section IV-C1), and 2) a thread identity abstraction algorithm that handles non-deterministic scheduling by tracking the order in which threads are spawned relatively to their spawning thread and naming them accordingly to achieve consistent IDs across multiple executions (Section IV-C2).

1) Memory Object Abstraction: Algorithm 1 outlines the pseudo-code for the memory object abstraction algorithm. Given an execution trace σ, the program outputs a set of symbolic memory objects O that can be used to replace the concrete addresses in the original execution trace.

For every event $e \in \sigma$, the algorithm first stores a global index (its position in the trace sequence) as well as the executing thread t (lines 3-4). If e is a memory allocation event, a memory object $o = (bAddr, t, l\_idx, size, val, s)$ is created where $bAddr$ is its concrete base address, l_idx is a thread local index that is incremented with every new instruction executed by t, $size$ is the size of the object, val is its initial validity range starting from g_idx, and s is a Boolean value indicating whether the allocation is a stack allocation (as opposed to heap) (lines 6-11). The object o is then added to O (line 12). The initial validity range of every added object has to be updated according to the scope where it was defined. If e is a memory de-allocation event, e.g., a call to the `free` function, the algorithm updates the validity range of the object $o \in O$ with the same base address (lines 14-15). Note that in this case o must be a heap object, and if it is never de-allocated the default range is still valid and also covers, for instance, global variables. In this case, the object is accessible from its creation until the end of the trace.

Memory objects on the stack are handled separately. If a new scope event for thread t is encountered, e.g., entering a function, the current global index is stored in $sp[t]$ for later use (line 17). In case e is an exit scope event, the algorithm updates the validity of all objects that were added after $sp[t]$ and restores the previous stack pointer (lines 19-21). Objects that were added after $sp[t]$ represent the objects that have to be de-allocated because the program is leaving the scope where they were defined. The restored stack pointer is assigned the global index of the last new scope event by thread t so that once that scope is exited, the validity range of the objects defined within it can be accordingly updated.

Once the set of objects O is generated, TraceSanitizer replaces each reference to a concrete memory address by a corresponding object eliminating trace deviations due to memory locations. An address is replaced by an object if it lies within its allocated space specified by its base address and size. If a memory address matches more than one memory object, we use the validity range to identify the correct object.

2) Thread Identity Abstraction: Algorithm 2 provides the pseudo-code for the thread identity abstraction that enables the matching of threads in different execution traces. The algorithm's goal is to achieve canonical thread IDs such that for every two executions of the same programs with the same input it is guaranteed that the same threads will receive the same id. Given a sequence of events σ, the algorithm builds a mapping function M that maps each thread ID to a canonical ID. The algorithm works by building a thread tree G where the nodes represent the spawned threads and the edges a spawning relation. If a thread $t_1 \in T$ spawns a thread t_2, we add a directed edge (t_1, t_2) between these two nodes (lines 5-7). The next step consists of breadth-first traversing the tree G, starting from the root node such that for every node the children are visited in their order of creation (lines 11-17). The canonical thread IDs are then recursively generated as follows:

- The root node is the main thread and is assigned the ID "T_0" (lines 9-10).
- Every node is assigned an ID that consists of its parent node's ID as a prefix and its position in the list of children (lines 15-17).

After running the thread identity abstraction algorithm, we use the mapping function M to rename the threads by their canonical names, and replace every reference to a thread ID in the execution trace. If a program satisfies the pseudo-deterministic condition, it is guaranteed that the canonical thread IDs match exactly, enabling the matching of all spawned threads across multiple traces in the trace comparison phase.

3) Reversibility Check: We developed an automated reversibility check to test whether an execution trace σ satisfies the pseudo-deterministic condition. The automated check focuses on the reversibility condition from Definition 1. While we manually checked the local determinism condition, the process can easily be automated using a black-listing approach to such external libraries or functions.

The reversibility check is based on the maximal causality technique which has been used for race detection in prior work [7]. A maximal causality formula encodes the maximal number of interleavings of a given trace that are guaranteed to be feasible, i.e., that are valid executions of the same program with the same input. Our reversibility check uses a modified version of the maximal causality formula that omits the constraints that ensure that only valid executions are encoded.

We utilize a reversibility formula Φ_σ whose satisfying solutions encode executions that are not necessarily valid, while preserving the soundness and completeness of the check. The formula defines integer variables x_i for every event $e_i \in \sigma$. The variables are then constrained in their order such that only the set of interleavings that are guaranteed to be feasible are allowed. For instance, if e_i is an event spawning a new thread, the formula adds a constraint $x_i < x_j$ for the first event e_j executed by the spawned thread. To guarantee the sequentiality of every thread t, a condition $x_i < x_j$ is added for every successive event by t. To prevent an overlap of two critical sections in the trace that are guarded by the same mutex, the formula adds a constraint $x_j < x'_i \vee x'_j < x_i$ where e_i and e'_i are two mutex acquiring events and e_j and e'_j are the two corresponding mutex release events.

Finally, we add additional constraints to encode our pseudo-deterministic condition:

$$R := \bigvee_{(e_i,e_j)\in D \wedge e_i \prec_\sigma e_j} x_j \leq x_i$$

Intuitively, the constraint encodes the fact that *any* two dependent events in the trace occur in a reversed order.

In the last step, we check the satisfiability of formula $\Phi_\sigma \wedge R$ using an SMT solver. If the solver does not return a solution, we have a proof that there cannot be any two dependent events that can occur in a reversed order and therefore the trace satisfies the pseudo-deterministic condition. If, however, the formula has been proven satisfiable, the solver returns a solution that encodes an execution trace where at least two dependent events are reversed. In this case, the trace does not satisfy the pseudo-deterministic condition.

Correctness. In our check, we omit the constraints that reduce the set of allowed interleavings to only those that are guaranteed to be feasible (i.e., the read conditions in [7]). Furthermore, the maximal causality model, upon which the reversibility formula is based, does not cover all feasible interleavings since it does not include executions that take new control flow paths [20]. These limitations, however, affect neither the soundness nor the completeness of the reversibility check as outlined next[1].

The soundness of the check is based on the fact that our reversibility formula can only contain infeasible interleavings if the execution is reversible. Let us assume that the check is unsound, i.e., for a trace σ that does not satisfy the reversibility condition, the reversibility formula is wrongly satisfiable. This means that the event order encoded by the reversibility formula describes an infeasible execution σ' due to the missing read conditions from [7]. Let e' be the first event in σ' that is not feasible and e the last event in the feasible prefix of σ' such that $Tid(e) = Tid(e')$. Because of local determinism we have $\sigma_e \neq \sigma'_e$ since otherwise e' (the next event by the same thread) would be executable in σ'. This would mean that σ'_e, and therefore also the feasible prefix of σ', contains at least a set of reversed dependent events. But this contradicts our initial assumption that σ does not satisfy the reversibility condition since the prefix of σ up to event e' is feasible.

Similarly, the completeness of the check follows from the fact that the set of interleavings covered by the formula is not complete only if the considered trace is reversible. Let us assume the check is incomplete, i.e., for a trace σ that satisfies the reversibility condition, the reversibility formula is wrongly unsatisfiable. This means that there is a feasible interleaving σ' of execution σ where two dependent events occur in reversed order and that is not covered by the reversibility formula. These two events cannot both be included in σ because otherwise the reversibility formula would be satisfiable. If at least one of the events, e, is not included in σ, its determining events σ'_e must include two events that occur in reversed order and are in σ, assuming that e is the first such an event in σ'. This means, however, that the reversibility formula will be satisfiable, contradicting our initial assumption.

4) An Example Trace Comparison: We use the example from Figure 1 to illustrate the working of TraceSanitizer. Given the execution trace from Figure 4 (upper right), the sanitizing algorithms produce a sanitized trace (upper left) and a memory object set and thread identity mapping (bottom left).

5) Memory Object Abstraction: Initially, the set of identified memory objects is empty. The algorithm starts by iterating over all events in the execution trace σ. After reaching an allocation instruction (line 1), a new object o4 is created and

[1] We refer the reader to [21] for a fuller discourse

```
0  T_0   call-main 0 1 o0
...
1  T_0   alloca o4
2  T_0   alloca o5
...
3  T_0   store 3 g0
4  T_0   store 6 g0+4
5  T_0   call-pthread_create-u 0 o4
       ↪ 0 400ae0 0
6  T_0   call-pthread_create-u 0 o5
       ↪ 0 4012c0 0
7  T_0_0 call-inc 0
8  T_0_0 alloca o6 1 8
9  T_0   load T_0_0 o4
10 T_0_0 alloca o7 1 8
11 T_0_0 store 0 o7
12 T_0_0 load 3 g0
13 T_0_1 call-dec 0
14 T_0_1 alloca o8 1 8
15 T_0_0 store 4 g0
16 T_0_1 alloca o9 1 8
17 T_0_1 store 0 o9
18 T_0_1 load 6 g0+4
19 T_0   call-pthread_join 0 T_0_0 0
20 T_0   load T_0_1 o5
21 T_0_1 store 5 g0+4
22 T_0   call-pthread_join 0 T_0_1 0
...
```

```
0  0 call-main 0 1 7ffcfe3287e8
...
1  0 alloca 7ffcfe3282e8 8
2  0 alloca 7ffcfe3282e0 8
...
3  0 store 3 603d74
4  0 store 6 603d78
5  0 call-pthread_create 0 7ffcfe3282e8
       ↪ 0 400ae0 0
6  0 call-pthread_create 0 7ffcfe3282e0
       ↪ 0 4012c0 0
7  1 call-inc 0
8  1 alloca 7f0ccbc55d58 8
9  0 load 7f0ccbc56700 7ffcfe3282e8
10 1 alloca 7f0ccbc55d50 8
11 1 store 0 7f0ccbc55d50
12 1 load 3 603d74
13 2 call-dec 0
14 2 alloca 7f0ccb454d58 8
15 1 store 4 603d74
16 2 alloca 7f0ccb454d50 8
17 2 store 0 7f0ccb454d50
18 2 load 6 603d78
19 0 call-pthread_join 0 7f0ccbc56700 0
20 0 load 7f0ccb455700 7ffcfe3282e0
21 2 store 5 603d78
22 0 call-pthread_join 0 7f0ccb455700 0
...
```

```
g0 := {ba=603d74, t=_, l_idx=0, s=8, v=[0,33]}
...
o0 := {ba=7ffcfe3287e8, t=T_0,   l_idx=0, s=8, v=[0,33]}
o4 := {ba=7ffcfe3282e8, t=T_0,   l_idx=1, s=8, v=[1,33]}
o5 := {ba=7ffcfe3282e0, t=T_0,   l_idx=2, s=8, v=[2,33]}
o6 := {ba=7f0ccbc55d58, t=T_0_0, l_idx=0, s=8, v=[8,15]}
o7 := {ba=7f0ccbc55d50, t=T_0_0, l_idx=1, s=8, v=[10,15]}
o8 := {ba=7f0ccb454d58, t=T_0_1, l_idx=0, s=8, v=[14,21]}
o9 := {ba=7f0ccb454d50, t=T_0_1, l_idx=1, s=8, v=[16,21]}

M(0) := T_0
M(1) := T_0_0
M(2) := T_0_1
```

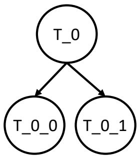

Fig. 4. Trace sanitizing example. Execution trace from Figure 2 after sanitizing (upper left). We show the first trace on the upper right side along with the generated memory object set and thread identity mapping on the bottom left. At the bottom right we present the thread tree generated by our algorithm.

added to the set of memory objects O. The object structure contains information about the base address returned by the LLVM-IR alloca instruction (7ffcfe3287e8), the size of the allocated object (8), the thread executing the instruction (0), the local index reflecting the position of the instruction in the sequence executed by the thread (1) and the initial validity range of the object ([1,33]) where 33 is the total number of instructions in the trace.

At the invocation of the inc function (line 7), a new scope is created and the stack pointer for thread 1 is updated to the index of the event where the scope was entered (7). This value will be used later to update the validity range of every new object created within the new scope. Similarly, at the call to dec in line 13, the algorithm updates the stack pointer for thread 2 to 13. At the end of each of the functions, the scope for both inc and dec ends and the validity range for the objects created within that scope has to be updated. Every object that has been added to O after the value in the stack pointer has to be updated. For instance, the validity range of object o6, added at line 8 within inc's scope, is updated be to [8,15] where 8 is the index of its allocation instruction in the trace and 15 the end of the scope. The stack pointer also has to be updated to the start of the previous scope. However, thit is unnecessary since only one scope has been created by this thread.

Finally, TraceSanitizer replaces the concrete addresses with the generated objects in every instruction in the trace. In this example there are also global variables that are accessed by the program. While Algorithm 1 does not include the handling of global variables, our implementation, however, handles these variables separately. Before iterating over the instructions, we add a memory object with maximal validity range for each global variable. The object g0 in Figure 4 (bottom left) represents the global array arr defined in the example. Note that accesses to g0 are not always at the base address. For instance, the access at line 18, occurs on the second element in the array, hence the reference g0+4 with an int type of byte length 4. Encountering an address that has not been explicitly allocated leads to the creation of a new object with default size, e.g., the argument of the main function results in the creation of object o0 (bottom left).

6) Thread Identity Abstraction: The algorithm first fetches the set of threads in the trace 0, 1, and 2 and adds nodes for these threads to G. We show the resulting spawning tree at the bottom right of Figure 4 (The nodes are renamed by their canonical IDs). Since thread 0 spawns threads 1 and 2, edges $(0,1)$ and $(0,2)$ are added to G. Next, the algorithm traverses the generated tree G to map concrete thread IDs to deterministically calculated thread IDs. The initial thread 0 is mapped to ID T_0. Every time a new node is reached in the tree, its child nodes are traversed in the order of the corresponding threads' creation. For instance, since thread 1 was created before 2, it will be traversed first. Following the renaming pattern presented in Algorithm 2, thread 1 is mapped to an ID consisting of its parent node ID (T_0) concatenated with an index indicating its creation order 0: T_0_0. Likewise, thread 2 is mapped to ID T_0_1 as shown in Figure 4. Finally, TraceSanitizer replaces every reference to a thread's ID in the program using the generated map M.

7) Reversibility Check: To build the reversibility check formula, a unique variable is assigned to every instruction in the trace. First, we encode the allowed interleavings of the trace by imposing constraints on the order of the formula variables. In addition to the constraints encoding the sequentiality of every thread, we add constraints for inter-thread synchronization. The call pthread_join at line 19 in Figure 4, for instance, forces the invoking thread 0 to wait for the termination of thread 2 before executing the next instruction. In this case, we add a constraint $x' < x$ for the call e to pthread_join and e' being the last instruction executed by thread 2.

Next, we identify all the dependencies between load and store instructions that read from or write to the same memory location. Consider the instructions e and e' at lines 4 and 18, respectively. Instruction e' reads from the same memory object g0, with the same offset, that e writes to. Therefore, both instructions are dependent on each other. Since e occurs before e' in the trace, we add the constraint $x' \leq x$ to the reversibility formula to check whether the two dependent instructions can occur in the reverse order. However, the formula does not allow any interleaving of the trace where instruction e occurs after e' as that would mean that the thread executing e' would start executing before it has been spawned. Since no such pair of instruction can be found for the trace, the generated reversibility formula cannot be satisfied and the trace will be declared to satisfy the reversibility condition. Note that our check ignores dependencies between instruction from the same thread as these can obviously not be reversed. Moreover, the reversibility formula only considers instructions that have "global" effects in the trace such as a synchronizing events and writes/reads on memory objects that are accessible by more than one thread.

8) Trace Comparison: After passing the reversibility check, we can safely compare the sanitized fault-free trace against sanitized traces from fault injection experiments with the same program processing the same input. The second trace, referred to as the faulty trace, is obtained by performing fault injection on the original program and re-executing it with the same input parameters. Since the threads are renamed systematically, the set of created thread ids in both traces are guaranteed to be the same. We start by dividing the sanitized traces into subsequences where each sequence contains only instructions belonging to a single thread. Next, we match the subsequences belonging to the same thread and compare every instruction. Since the threads are renamed identically in traces from different executions with the same input, the set of created IDs in both traces should match in a comparison of fault-free executions. Furthermore, if the first trace had passed the reversibility check and the second trace is a re-execution of the same program with the same input, then every pair of two such subsequences should be identical unless one of the traces is affected by an injected fault. This is guaranteed by the local determinism property of every thread induced by the pseudo-deterministic condition of the first trace. If dependent instructions cannot occur in a reversed order, every thread will be created in the same order by the same parent thread, and memory objects will also be allocated in the same order and by the same thread. Therefore, a deviation between both traces implies that the injected fault has been activated in the experiment and its effects on the execution show in the comparison. Concretely, the comparison algorithm checks whether instructions in both subsequences occur in the same order and whether they access the same objects with the same offsets. Applying the sanitizing algorithms on both traces from fig. 2 results in pairs of identical subsequences for every thread in the trace (`T_0`, `T_0_0` and `T_0_1`) since the first trace satisfies the pseudo-deterministic condition and the second trace is not faulty.

V. TraceSanitizer Implementation

The publicly available TraceSanitizer implementation[2] consists of two modules: (1) an instrumentation and fault injection module that is implemented as an extension of the LLFI EPA tool [17], and (2) a sanitization and trace comparison module. The first module adds more logging information in the trace generation process and thread safety to the LLFI tool. The instrumentation and fault injection parts are performed at the level of the intermediate level representation of the LLVM compiler infrastructure. We implemented the second module in the Rust programming language and used the Z3 SMT solver [22], [23] for the reversibility check. Since the reversibility formula we generate uses the standard SMT-LIB format [24], TraceSanitizer can use most existing solvers.

TABLE I
Overview of the Benchmark Programs. SLOC reports the source lines of code, #Th the sum of spawned threads and #Inst the number of executed instructions in one run. Mem-Sound/Sched-Sound whether false positives occurred due to memory/CPU non-determinism (✗) or not (✓).

Program	SLOC	#Th	#Inst	Mem-Sound		Sched-Sound	
				Naïve	TSAN	Naïve	TSAN
`quicksort`	198	72	45 k	✗	✓	✗	✓
`pca`	301	17	89 k	✗	✓	✗	✓
`kmeans`	425	65	44 k	✗	✓	✗	✓
`blackscholes`	393	3	91 k	✗	✓	✗	✓
`swaptions`	1118	4	1.1 M	✗	✓	✗	✓

VI. Evaluation

We first enumerate the research questions we aim to answer, followed by the experimental setup and obtained results. We conducted all of the experiments on machines with an Intel Core i7-4790 CPU, 16 GiB of RAM, and a 500 GB SSD running Debian 8.11 with a Linux 3.16 kernel.

A. Research Questions

The goal of our evaluation is to show that the TraceSanitizer approach eliminates *all* false positives in EPA due to execution non-determinism, and to measure the performance overhead of TraceSanitizer. Four research questions (RQs) are pertinent.

RQ 1 What are the false positive rates resulting from non-determinism in dynamic memory allocations with and without TraceSanitizer?

RQ 2 What false positive rates result from CPU scheduling non-determinism with and without TraceSanitizer?

RQ 3 What is the rate of false negatives with TraceSanitizer?

RQ 4 What is the performance overhead of TraceSanitizer?

B. Target Programs and Execution Environment

Our evaluation targets five C/C++ programs listed in Table I, four of which are taken from the PARSEC [25] and Phoenix benchmarks [26], that satisfy the pseudo-deterministic condition. These programs were also used in prior work on using likely invariants for EPA to counter non-determinism [12].

Table I reports the number of SLOCS of each program along with the total number of instructions and spawned threads. `quicksort` is a parallel implementation of the well-known sorting algorithm. `pca` and `kmeans` are two machine-learning algorithms taken from the Phoenix benchmark suite [26]. `kmeans` is an implementation of the Kmeans clustering algorithm and `pca` implements the Principal Component Analysis statistical procedure. Additionally, we used the `blackscholes` and `swaptions` programs from the PARSEC benchmark suite [25]. The `blackscholes` programs solves the `blackscholes` partial differential equation used in pricing a portfolio of European-style stock options. `swaptions` uses Monte-Carlo simulations to compute

[2] https://github.com/DEEDS-TUD/TraceSanitizer

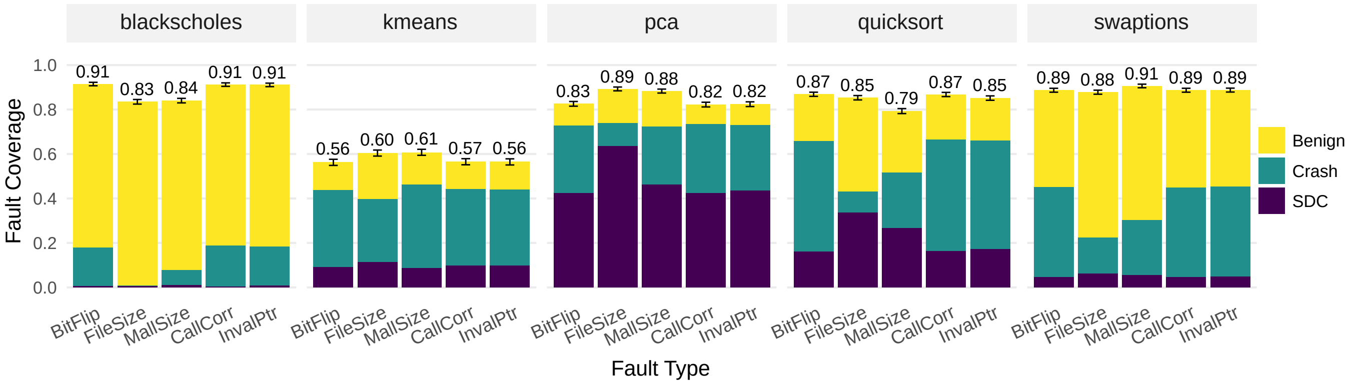

Fig. 5. Results from 5000 runs of each combination of target and fault type. The error bars indicate the 95 % confidence interval.

`swaptions`, a form of financial derivatives. We verified that all five programs satisfy the pseudo-deterministic condition.

C. RQ 1: False Positives from Memory Addresses

Our work is motivated by the observation that non-determinism can lead to false positives if a naive execution trace comparison is applied for EPA. As such benign non-determinism can affect execution traces differently depending on the origin, TraceSanitizer employs different analyses, which we evaluate separately. We separately evaluate the effectiveness of TraceSanitizer in eliminating false positives in EPA that are due to dynamic memory allocation and non-deterministic scheduling. We begin with an evaluation of the impact that dynamic memory allocation non-determinism has on execution traces, and how well TraceSanitizer can deal with those cases. We then evaluate false positives resulting from CPU scheduling non-determinism in Section VI-D.

To evaluate the impact of dynamic memory allocation non-determinism independently from CPU scheduling effects, we conduct a number of trace comparisons on single-threaded executions of our target programs without any fault injections. For this purpose, we use single-threaded versions of the five programs from Table I. As we do not inject any faults, any observed deviation across repeated executions of the same program must be a false positive. Moreover, since CPU scheduling cannot cause deviations in single-threaded programs, any observed false positive is likely due to memory non-determinism.

To simulate an "unlucky" injection campaign with numerous unactivated faults, we generate 10 000 fault-free execution traces for each of the single-threaded programs by performing steps ① and ② in Figure 3, and perform a line-by-line comparison, just as conventional EPA approaches would compare golden run and fault injection traces. We then run TraceSanitizer's memory abstraction algorithm on the same traces – the results are in the Mem-Sound column of Table I.

The results show that no execution trace is identical to any other from the 10 000 repetitions, and that TraceSanitizer is able to eliminate all of these spurious deviations for all of the benchmarks. Besides demonstrating the effectiveness of our memory abstraction, this result confirms that the second criterion from our definition of the pseudo-deterministic condition in Section I is satisfied for the chosen program input. If non-deterministic external functions affect the execution, this would lead to trace deviations across the repeated executions.

D. RQ 2: False Positives from CPU Scheduling

To assess the effectiveness of TraceSanitizer to compensate for the effects of non-deterministic CPU scheduling, we recompiled the target programs to use multiple threads, and generate 10 000 fault-free execution traces for each program. We sanitized the effects of memory allocation non-determinism in the execution traces and compared the resulting traces of the repeated executions. As no faults are injected in these executions, any deviations between traces constitute false positives. Moreover, as the effects of memory allocation non-determinism are sanitized, all deviations must result from CPU scheduling.

We again compare the obtained traces in a line-by-line fashion without TraceSanitizer, and observe deviations in each comparison. We then run TraceSanitizer's thread abstraction algorithm on the traces and perform the comparison again. The results shown in the Sched-Sound column of Table I demonstrate that TraceSanitizer is able to fully eliminate false positives resulting from non-deterministic CPU scheduling for pseudo-deterministic programs.

E. RQ 3: False Negatives Introduced by TraceSanitizer

Accurately measuring false negative rates in EPA experiments is challenging, because there is no oracle to distinguish between true and false negatives. If a fault is injected and no effect is observed, it is unclear whether no effect has occurred (true negative), or an effect has occurred and it was missed by the detection mechanism (false negative). Moreover, differential testing using different detection mechanisms is difficult to apply in the case of EPA for multi-threaded programs, because other approaches are not sound, and their false positives would distort the results. Therefore, we base our evaluation of false negatives on a conservative estimate.

Assuming that each injected fault leads to error propagation (this may not always hold [27]), each succeeding trace comparison between an injection and a fault-free run constitutes a false negative. We term the fraction of these succeeding comparisons from all comparisons the *maximal possible false negative rate (MPFNR)*.

To ensure that TraceSanitizer does not achieve soundness at the cost of an increased false negative rate, we executed a number of fault injection experiments following the steps outlined in Figure 3 and discussed in Section IV-A and assessed the MPFNR. We used the multi-threaded versions of the benchmark programs from Section VI-D, and first generated and sanitized execution traces from one fault-free execution of each program. We then ran our memory and thread abstraction algorithms (cf. Section IV-C) on the trace and performed the reversibility check to ensure that comparisons against this trace yield sound results if the same program inputs are used.

To obtain traces from faulty executions, we repeated the execution of the program with the same inputs and injected one fault per execution using the LLFI fault injection framework. The injection points for the faults are decided dynamically by the LLFI framework. In total we executed $25\,000$ such experiments, consisting of 5000 fault instances for each of the fault types listed in Table II (these were also used in prior work [12]). We then sanitized each of the resulting execution traces using TraceSanitizer.

Figure 5 shows the *fault coverage* of EPA using TraceSanitizer, i.e., the fraction of experiments for which the sanitized fault-injection traces differed from the fault-free trace and, thus, indicate error propagation. The MPFNR is the difference between 1 and the reported fault coverage and ranges between $44\,\%$ and $9\,\%$ depending on the program and fault type. While we cannot tell whether any of the succeeded comparisons was due to the lack of error propagation or due to a false negative of our approach, we can tell if TraceSanitizer has any obvious blind spots by investigating the false negative rates for experiments that led to program failures (i.e., externally observable deviation from correct behavior). If the program behavior deviates from correct behavior as observed in the fault-free execution, an error *must* have propagated, and missing such propagation in the traces would be a false negative.

To assess the error propagations that we missed, we have calculated the MPFNR for different classes of experiment outcomes that are indicated by different colors in Figure 5. A *crash* denotes cases where the program terminated abnormally after a fault was injected, whereas *SDC (silent data corruption)* indicates cases where the program terminated without error indication, but its results differed from the fault-free case. For both crashes and SDCs, we found the MPFNR to be $0\,\%$. This demonstrates that there were no obvious cases of error propagation that were missed by TraceSanitizer.

To the best of our knowledge, TraceSanitizer is the first to achieve a $0\,\%$ *false positive rate for EPA on multi-threaded programs without increasing the false negative rate for known cases of error propagation (observed crashes and SDCs).*

F. RQ 4: TraceSanitizer Overhead

Achieving a high fault coverage and fully eliminating false positives comes at the cost of (a) running the reversibility check on the golden run to ensure the soundness of the approach, and (b) running the sanitization algorithms on the traces.

TABLE II
Overview of Injected Fault Types

Fault Type	Short Description
BitFlip	Flips single bits in arbitrary data values.
FileSize	Increases the `size` parameter in `fread` and `fwrite` function calls for file I/O.
MallSize	Decreases the `size` parameter in `malloc` and `calloc` function calls for memory allocation.
CallCorr	Corrupts the first parameter of function calls.
InvalPtr	Corrupts the returned pointers from `malloc` and `calloc` function calls.

TABLE III
Performance Results for TSAN. Reversibility Check times are reported in minutes and EPA times in seconds. EPA times are median values over 5000 runs.

Program	Rev. Check				EPA	
	#Obj.	#Dep.	Solver [m]	Total [m]	San. [s]	Cmp. [s]
`quicksort`	38	24 650	30.36	30.38	1.57	0.3
`pca`	64	12 126	150.41	150.43	1.29	0.17
`kmeans`	31	13 460	81.93	81.94	0.79	0.13
`blackscholes`	13	2810	0.87	0.99	1.58	0.2
`swaptions`	16	22 630	116.66	144.61	8.57	2.86

It is important to note that the time overhead incurred by the reversibility check is a one-time cost as the check needs to be run only once. Further, the fault injection experiments can be run in parallel with the checks. On the other hand, running the sanitization algorithms needs to be done for each fault that is injected (typically thousands of times for obtaining statistically significant estimates). We measure the time it takes to run each of these two steps – the results are shown in Table III.

1) Reversibility Check: To assess the run time overhead of the reversibility check we performed it on a golden run of each of the benchmarks. We report the total run time along with time taken by the SMT solver, the number of memory objects accessed in the trace, and the inter-thread dependencies on these objects in Table III. For all programs, the overall time for the reversibility check ranges from approximately $1\,\mathrm{min}$ for `blackscholes` to $150\,\mathrm{min}$ for `pca`, and is strongly dominated by the SMT solver's execution time. For `swaptions`, which is the only program showing a notable difference between these times, building the formula takes considerably longer due to the higher number of instructions in the trace.

In addition to the solving time, the total overhead consists of the time it takes TraceSanitizer to build the formula, including the identification of data-dependencies in the trace. While the number of dependencies and objects, along with the total number of instructions hint at the size and complexity of the formulas generated, they do not directly correspond to the measured execution times. For instance, `quicksort` has a higher complexity than `kmeans` in terms of memory objects and dependencies in the traces with a comparable trace size, but takes significantly less time for the check.

To understand how our technique performs for target programs of different complexity, we conducted a scalability study for one of the benchmarks (`blackscholes`). For this pur-

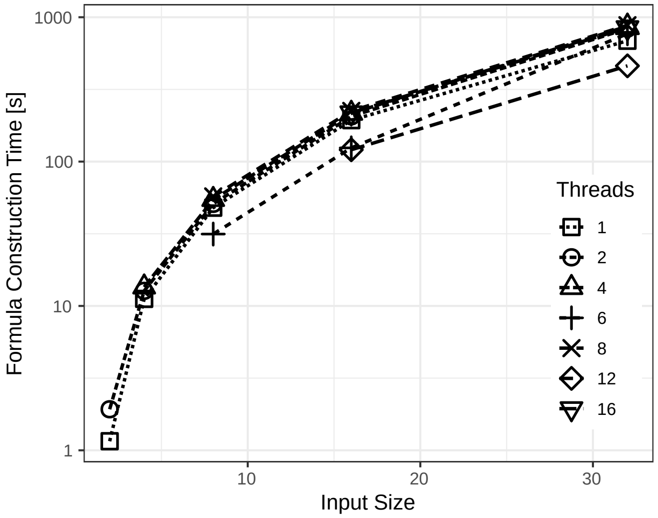

Fig. 6. Reversibility formula build time in seconds (log scale) for `blackscholes` traces with different input sizes and numbers of threads

pose, we generated execution traces of the benchmark with inputs of varying size (2,4,8,16,32 inputs) and varying numbers of threads (1,2,4,6,8,12,16) handling these inputs. We have repeated the reversibility check four times for each input/thread count combination to account for execution time variations.

We divide the execution time into building and solving the reversibility formula. Figure 6 shows the average time of TraceSanitizer for constructing the reversibility formula in relation to input size and thread count for the `blackscholes` benchmark. As `blackscholes` intrinsically limits the number of worker threads to the number of inputs, the plot only shows data points where the number of threads is higher than or equal to the number of inputs. As can be seen, the formula build time increases with the input size, but remains below $15\,\mathrm{min}$ in all cases. The number of threads only has a relatively small influence; for example, for input sizes 16 and 32, the time taken for 12 threads is lower than the time for fewer threads.

For brevity, we do not report on solver times in detail, but summarize our findings.

- The solver time significantly exceeds the formula building time (by an average factor of 191), with a maximum average solver time of almost $103\,\mathrm{h}$ (32 inputs, 16 threads).
- Although we observed the highest solver time for the most complex configuration, we find that solver time does not strictly increase with thread count or input size.
- We find solver time to vary strongly across repetitions with a coefficient of variation of up to $32.4\,\%$.

From our scalability analysis we conclude that (1) building reversibility formulas for TraceSanitizer is not a performance bottleneck, (2) solving reversibility formulas dominates the overall time for the reversibility check and may become a bottleneck, but is a one time cost for TraceSanitizer and will improve as SMT solvers evolve.and (3) solver time can vary strongly in unforeseen ways for different execution configurations.

2) Trace Sanitizing: Once the golden run has passed the reversibility check, TraceSanitizer proceeds with the sanitization and comparison of faulty runs. Next, we measured the additional overhead incurred by running the sanitizing algorithms and the actual comparison on each faulty run.

Table III shows a break-down of the median time across 5000 experiments that TraceSanitizer requires to perform these sanitization (column 6) and comparison (column 7) steps. The median time for trace sanitization ranges between $0.79\,\mathrm{s}$ and $8.75\,\mathrm{s}$ with a median absolute deviation (MAD) of $1.9\,\mathrm{s}$ for `swaptions` and less than $0.4\,\mathrm{s}$ for the other benchmarks. The trace comparison of a sanitized golden run and a faulty run takes between $0.17\,\mathrm{s}$ and $2.86\,\mathrm{s}$ with a MAD of $0.2\,\mathrm{s}$ for `swaptions` and under $0.02\,\mathrm{s}$ for the other benchmarks.

While we cannot directly compare these results to existing approaches due to the strong impact of machine configurations on performance measurements, we can provide an indirect comparison. As TraceSanitizer is the only sound tool for EPA trace comparisons, it does not require any manual inspection of the obtained comparison results to check for false positives, which are required by unsound tools. To beat TraceSanitizer's performance for 5000 injections in the slowest case of `swaptions`, 4400 trace diffs ($5000 \cdot 0.88$, the smallest coverage in Figure 5) would need to be inspected (manually) in less time than $\frac{5000 \cdot 8.57\,\mathrm{s} + 144.61 \cdot 60\,\mathrm{s}}{4400}$, which is less than 12 seconds for a diff across traces with more than a million lines (Table I). An analogous calculation yields less than 4 seconds for manual inspection of any other benchmark. Such small times are almost impossible to achieve for any realistic program trace, including those in our evaluation. Moreover, the time taken by TraceSanitizer will become smaller as computing becomes faster, which is not the case for manual inspection.

VII. Conclusion

In this paper, we introduced a class of multi-threaded programs that we termed pseudo-deterministic, and for which EPA can be sound in the presence of non-deterministic memory allocations and CPU scheduling. We have developed an automated technique to determine whether a program belongs to this class as well as a novel trace sanitizing approach that soundly handles non-determinism. We implemented the technique in an automated tool called TraceSanitizer using the LLVM compiler, and Satisfiability Modulo (SMT) Solvers.

We empirically evaluated our TraceSanitizer prototype on five benchmark programs and demonstrated that it is able to fully eliminate false positives. Further, it achieves a high fault coverage in an EPA study, across five different fault types. Finally, TraceSanitizer provides reasonable performance, and compares very favourably with unsound EPA tools that require manual inspection of false-positives.

Acknowledgments

This work was supported in part by DAAD (project 57389931), EC H2020 CONCORDIA GA# 830927, the Lancaster Security Institute, the NSERC, and the Killam Research Fellowship from UBC. The scalability analysis results were obtained using the Chameleon testbed supported by the NSF.

References

[1] M. Hiller, A. Jhumka, and N. Suri, "On the placement of software mechanisms for detection of data errors," in *Proceedings International Conference on Dependable Systems and Networks*, June 2002, pp. 135–144.

[2] N. Coppik, O. Schwahn, S. Winter, and N. Suri, "TrEKer: Tracing Error Propagation in Operating System Kernels," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 377–387. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155612

[3] R. Natella, S. Winter, D. Cotroneo, and N. Suri, "Analyzing the Effects of Bugs on Software Interfaces," *IEEE Transactions on Software Engineering (to appear)*, pp. 1–1, 2018.

[4] J. Voas, "Error propagation analysis for COTS systems," *Computing Control Engineering Journal*, vol. 8, no. 6, pp. 269–272, Dec 1997.

[5] M. Hiller, A. Jhumka, and N. Suri, "An approach for analysing the propagation of data errors in software," in *2001 International Conference on Dependable Systems and Networks*, July 2001, pp. 161–170.

[6] Y. Chen, S. Zhang, Q. Guo, L. Li, R. Wu, and T. Chen, "Deterministic Replay: A Survey," *ACM Comput. Surv.*, vol. 48, no. 2, pp. 17:1–17:47, Sep. 2015. [Online]. Available: http://doi.acm.org/10.1145/2790077

[7] J. Huang, P. O. Meredith, and G. Rosu, "Maximal sound predictive race detection with control flow abstraction," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 337–348, 2014.

[8] P. E. Black, "Algorithms and Theory of Computation Handbook, CRC Press LLC, 1999, "single program multiple data"," https://www.nist.gov/dads/HTML/singleprogrm.html, 2004, accessed 2019-05-14.

[9] S. Okur and D. Dig, "How Do Developers Use Parallel Libraries?" in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 54:1–54:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393660

[10] G. Lemos and E. Martins, "Specification-guided golden run for analysis of robustness testing results," in *Proc. SERE '12*, 2012, pp. 157–166.

[11] M. Leeke and A. Jhumka, "Evaluating the Use of Reference Run Models in Fault Injection Analysis," in *Proc. PRDC '09*, 2009, pp. 121–124.

[12] A. Chan, S. Winter, H. Saissi, K. Pattabiraman, and N. Suri, "Ipa: Error propagation analysis of multi-threaded programs using likely invariants," in *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*. IEEE, 2017, pp. 184–195.

[13] T. Liu, C. Curtsinger, and E. D. Berger, "Dthreads: efficient deterministic multithreading," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 327–336.

[14] R. L. Bocchino Jr, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian, "A type and effect system for deterministic parallel java," in *ACM Sigplan Notices*, vol. 44, no. 10. ACM, 2009, pp. 97–116.

[15] J. Christmansson, M. Hiller, and M. Rimen, "An experimental comparison of fault and error injection," in *Proc. ISSRE '98*, 1998, pp. 369–378.

[16] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1, pp. 35 – 45, 2007, special issue on Experimental Software and Toolkits. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016764230700161X

[17] A. Thomas and K. Pattabiraman, "LLFI: An intermediate code level fault injector for soft computing applications," in *Workshop on Silicon Errors in Logic System Effects (SELSE)*, 2013.

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

[19] S. Hong and M. Kim, "A survey of race bug detection techniques for multithreaded programmes," *Software Testing, Verification and Reliability*, vol. 25, no. 3, pp. 191–217, 2015. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1564

[20] T. F. Şerbănuţă, F. Chen, and G. Roşu, "Maximal causal models for sequentially consistent systems," in *International Conference on Runtime Verification*. Springer, 2012, pp. 136–150.

[21] H. Saissi, "On the Application of Formal Techniques for Dependable Concurrent Systems," Ph.D. dissertation, Technische Universität, Darmstadt, 2019. [Online]. Available: http://tuprints.ulb.tu-darmstadt.de/8600/

[22] https://github.com/Z3Prover/z3, 2019.

[23] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[24] C. Barrett, A. Stump, C. Tinelli *et al.*, "The smt-lib standard: Version 2.0," in *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, vol. 13, 2010, p. 14.

[25] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 72–81.

[26] https://github.com/kozyraki/phoenix, 2016.

[27] W. Masri and R. A. Assi, "Prevalence of Coincidental Correctness and Mitigation of Its Impact on Fault Localization," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 1, pp. 8:1–8:28, Feb. 2014. [Online]. Available: http://doi.acm.org/10.1145/2559932