SIFT: A Tool for Property Directed Symbolic Execution of Multithreaded Software

Tuba Yavuz
ECE Department
University of Florida
Gainesville, FL, USA
tuba@ece.ufl.edu

***Abstract*—**
Analyzing multithreaded programs is notoriously hard due to the exponential number of thread interleavings. Although race detectors can help developers find and fix such bugs before the code is deployed, multithreaded code may still be buggy due to memory errors and assertion violations that are not due to race conditions. This paper presents a property directed symbolic execution of multithreaded code. Our approach, named SIFT, differs from previous work on detecting errors in multithreaded code by being property directed and by handling both memory safety and assertion checking that can be further customized by the user. SIFT can detect bugs that may or may not be due to data races, and works in an iterative way. In each step, it explores the state space using selective scheduling based on a set of interleaving points that have been inferred in the previous step. We have developed three partitioning strategies for improved effectiveness and performance. We have implemented SIFT on top of the KLEE symbolic execution engine and applied it to various real-world and academic benchmarks. SIFT could detect more vulnerabilities than a state-of-the-art memory vulnerability detector.

***Index Terms*—concurrency, symbolic execution, bug finding**

I. Introduction

Analyzing multithreaded programs is notoriously difficult due to the exponential number of thread interleavings. Although race detectors [14], [6], [13] can help developers find and fix such bugs before the code is deployed, multithreaded code may still be buggy due to memory errors and assertion violations that are not due to race conditions. Also, race condition warnings may require additional analysis in case the programmers may not be convinced of the potential risks.

There have been some recent efforts [9], [12], [17], [4], [11], [7], [13] to fill this gap in the reliability and security of multithreaded software. These approaches share in common the idea of analyzing the dependencies in the state space of a multithreaded code and inferring bug revealing schedules.

Some of these works depend on an offline static analysis for pointer analysis [9], [13] while others analyze dynamic execution traces [12], [17], [4], [11] to identify the schedule relevant program actions. Thread scheduling related search space reduction techniques include assertion guided prediction of error relevant states [9], constraint solving [12], [17], [11], distributed trace partitioning [11], leveraging stack traces from crashing runs [4], and distance-based selection of event reordering [7]. The approach in [9] targets assertion failures. Cortex [17] and the approach in [12] do not target specific types of bugs. ConCrash [4] is not restricted in terms of the types of bugs it can detect as long as the crash report contains sufficient information about the crash. UFO [11] detects Use-After-Free (UAF) vulnerabilities. ConVul [7] detects, in addition to UAF, the Null pointer dereferencing and Double-Free (DF) vulnerabilities. RAZZER [13] guides its fuzzing engine to detect race conditions.

We think that there is a need for a generic *property directed* approach for the analysis of multithreaded programs. The properties of interest include memory safety and other user defined safety properties. The analysis should be able to identify property relevant data-flow dependencies as precisely as possible to effectively search the huge state space of scheduling scenarios.

This paper presents a property directed symbolic execution of multithreaded code. We choose symbolic execution as the underlying program analysis technique due to its precise memory model and its intrinsic capability to detect memory vulnerabilities. Our approach, named SIFT, differs from previous work on detecting errors in multithreaded code by being property directed and by handling both memory safety and assertion checking that can be further customized by the user. SIFT can detect bugs that may or may not be due to data races, and works in an iterative way. In each step, it explores the state space using selective scheduling based on a set of thread interleaving points that have been inferred in the previous step. We have developed three partitioning strategies for improved effectiveness and performance. We have implemented SIFT on top of the KLEE symbolic execution engine [5] and applied it to various real-world and academic benchmarks. SIFT could detect more vulnerabilities than a state-of-the-art memory vulnerability detector.

This paper makes the following contributions:

- We present a property guided selective symbolic execution technique that performs on-the-fly data-flow analysis without relying on any offline analysis. The state space search can be optimized using three different partitioning strategies with different strengths.
- We have implemented the presented approach in a tool

called SIFT[1] on top of the KLEE symbolic execution engine.
- We have applied SIFT to a variety of real-world and academic benchmarks. Experimental results show that SIFT is fast in detecting bugs and detects more vulnerabilities than what could be detected by ConVul [7].

This paper is organized as follows. In Section II, we provide motivating examples to demonstrate the underlying insights of our approach. In Section III, we present the technical details of our approach. In Section IV, we present an evaluation of our approach on various benchmarks. In Section V, we discuss our work within the context of related work. In Section VI, we conclude with directions for future work.

II. A Motivating Example

In this section, we present a motivating real-world example and demonstrate the salient features of our approach. Figure 2 shows a simplified code snippet from the security subsystem of the Linux kernel. When two threads execute the `lookup_user_key` and the `install_user_keyrings` concurrently, a NULL pointer dereference may be encountered as shown in Figure 1. This is because Thread 0, which performs the key lookup, first finds out the session keyring has not been created yet. So, it decides to install it by calling `install_user_keyrings`, but before it actually attempts to install the keyring, a context switch happens. When Thread 1 executes, it also realizes that the keyring is not created yet and, so, it creates the keyring and starts installing it. However, Thread 1 can only partially install the keyring as it can only perform the update for the `uid_keyring` field at line 130 and a context switch happens before it can execute the line at 131 that sets the `session_keyring`. When Thread 0 continues executing, it checks only the `uid_keyring` field, and assumes that the keyring must have been properly installed. Accessing the not yet initialized `session_keyring` at line 174, leads to a kernel crash. An error revealing scheduling scenario is depicted in Figure 1.

The manifestation of the vulnerability depends on whether the context switches between the two threads happen at some specific program locations. We call such locations the *interleaving points*. For this example, the first interleaving point is when the program counter of Thread 0 refers to the read operation at line 114 and the second interleaving point is when the program counter of Thread 1 refers to the write operation at line 131. The context switch happens before the instructions referred by the program counter gets executed.

Although this bug is due to a data race (see Section III for a vulnerability not due to a data race), not all thread schedules of this code would lead to the NULL pointer dereferencing problem. So, our approach leverages the inherent capability of a symbolic execution engine in detecting memory errors, and performs on-the-fly data-flow analysis on a minimal number of paths to infer the interleaving points. For instance, SIFT can leverage even a seemingly useless thread scheduling scenario such as Thread 0 performing the keyring installation as part of its lookup, followed by Thread 1 checking if the keyring needs to be installed and finding that the keyring has actually been installed, to glean information to identify interleaving points. This is because in addition to using the data-flow facts available on an execution trace, it also performs light-weight on-the-fly static analysis to identify branches that may be error relevant. Then, in an iterative way, SIFT generates additional scheduling scenarios and performs data-flow analysis on the new traces to enrich the set of interleaving points until it detects the error.

[1]Available at https://github.com/sysrel/SIFT.

III. Approach

In this section, we present the technical details of our property directed symbolic execution approach for multithreaded code using the running example provided in Figure 3, in which each of the functions is executed by a separate thread. This example contains a use-after-free at line 27 that is not due to a data race and two memory overflows at lines 29 and 30 that involve data races.

Section III-A presents the terminology related to multithreaded execution. Section III-B presents the inference rules for identifying program locations that will be used as interleaving points. Section III-C presents how dynamic symbolic execution is extended to leverage the inferred interleaving points in effective exploration of a multithreaded code with the goal of detecting errors.

A. Preliminaries

We assume that the multithreaded code runs under a sequentially consistent memory model. In this work, we consider memory safety and safety properties that can be checked using assert statements or customized error checking functions. Our approach can handle memory vulnerabilities such as memory out of bounds, NULL pointer dereferencing, Use-After-Free, and Double-Free. We abstract the semantics of an execution path P in terms of the property relevant memory objects accessed by the end of the path, O, and the property relevant instructions that get executed on P, PR, and represent this fact with $P \vdash O, PR$.

Given an execution path P and an expression exp that got evaluated on P, $Mem(exp)$ denotes the set of memory objects that are accessed while evaluating exp on P and $Def(exp)$ denotes all the instructions that define exp on P. For a given address expression $aexp$, $PointsTo(aexp)$ denotes the memory object that $aexp$ refers to. Note that $aexp$ may refer to multiple objects due to being a symbolic expression rather than a concrete value. However, we assume that it gets resolved into a single object by the time the instruction completes its execution as in symbolic execution a separate path would be generated for each candidate object that the pointer expression refers to. Our approach assumes that each array and each `struct` is modeled as a single memory object, and, therefore, our points-to analysis maps address expressions that refer to different elements of the same array to the memory object that represents the array. Similarly, it maps address expressions

```
THREAD 0 (executing lookup_user_key)                      THREAD 1 (executing install_user_keyrings)
============================================              ============================================
if (!cred->user->session_keyring) { // Line 166
 // interleaving point at 114 in install_user_keyrings
                                                          ...
                                                          if (!user->uid_keyring) { // Line 114
                                                          user->uid_keyring = uid_keyring; // Line 130
                                                          // interleaving point at 131
   { ... if (user->uid_keyring) return 0; } // Line 114
key = cred->user->session_keyring;
atomic_inc(&key->usage);[OOPS] // Line 174
                                                          user->session_keyring = session_keyring; //
                                                          ↪  Line 131
```

Fig. 1: The erroneous thread interleaving scheduled produced by SIFT to detect the NULL pointer dereference at line 92 (accessed from line 174) within the code shown in Figure 2.

```
void atomic_inc(atomic_t * v) { (v->counter)++; }
 ↪  // Line 92
int install_user_keyrings(int thread_num) {
   user = cred->user;
   if (user->uid_keyring) return 0; // Line 114
   mutex_lock(&key_user_keyring_mutex);
   if (!user->uid_keyring) { // Line 122
      uid_keyring = (key *)malloc(sizeof (struct
      ↪  key));
      session_keyring = (key *)malloc(sizeof
      ↪  (struct key));
      user->uid_keyring = uid_keyring; // Line
      ↪  130
      user->session_keyring = session_keyring; //
      ↪  Line 131
   }
   mutex_unlock(&key_user_keyring_mutex);
   return 0;
 }

 key_ref_t lookup_user_key(...) {
     if (!cred->user->session_keyring) { // Line
     ↪  166
        printf("thread 1: session_keyring not
        ↪  exist\n");
        ret = install_user_keyrings(1);
        if (ret < 0) goto error;
     }
     key = cred->user->session_keyring;
     atomic_inc(&key->usage); // Line 174
        ...
}
```

Fig. 2: A simplified multithreaded code segment related to CVE-2013-1792 [1] that involves a NULL pointer dereference.

```
pthread_mutex_t  mutex;
int data = 0;
char *name = NULL;
char *address = "1000NW10thSt";
char letter;
char *zipcode = "66666";
int ind=4;

void *thread1(void *arg) {
  pthread_mutex_lock(&mutex);
  if (data > 0)
     free(name);
  pthread_mutex_unlock(&mutex);
  return 0;
}

void *thread2(void *arg) {
  pthread_mutex_lock(&mutex);
  data++;
  pthread_mutex_unlock(&mutex);
  ind++;
  return 0;
}

void *thread3(void *arg) {
  pthread_mutex_lock(&mutex);
  letter = name[10];
  pthread_mutex_unlock(&mutex);
  letter = address[12+data];
  zipcode[ind] = '1';
  return 0;
}

int main()
{
  name = malloc(20);
  pthread_mutex_init(&mutex, 0);
  pthread_t t1, t2, t3;
  pthread_create(&t1, 0, thread1, 0);
  pthread_create(&t2, 0, thread2, 0);
  pthread_create(&t3, 0, thread3, 0);
  pthread_join(t1, 0);
  pthread_join(t2, 0);
  pthread_join(t3, 0);
  printf("Letter %d\n", letter);
  return 0;
}
```

Fig. 3: An example multithreaded application with three threads.

that refer to different fields of the same `struct` object to the memory object that represents the `struct` object.

We abstract the load instructions by ignoring the offset value and represent them with read operations, $read\ x$, where x refers to a memory object. In a similar way, we abstract the store instructions by ignoring the written value and by representing them with write operations as $write\ x$.

We distinguish memory objects with a global scope using the predicate $isGlobal$. We consider a local variable as escaping if the address of the local variable is stored into a global memory object or if it is directly or indirectly passed as an argument to the thread creation API function. We distinguish local variables that escape the local scope from those that do not with the predicate $localEscapes(x)$, where x is the locally defined memory object. We consider a memory object as globally visible if it is a global variable or it is an escaping local variable and denote this fact with the $globallyVisible$ predicate, i.e., $globallyVisible(x)\ \equiv$

$isGlobal(x) \vee localEscapes(x)$.

The instructions in a multithreaded program are related to each other according to the *happens-before* relation [15]. There is a happens-before relationship between two instructions i_1 and i_2 that get executed on the same path P, denoted by $i_1 \xrightarrow[P]{\text{hb}} i_2$ if one of the following holds:

- i_1 and i_2 are executed by the same thread and i_1 gets executed before i_2 according to the program order.
- i_1 gets executed by a thread t_k before t_k creates thread t_j that executes i_2.
- i_1 gets executed by a thread t_k after t_k joins thread t_j that executes i_2.
- i_1 is a release operation on a lock object o and i_2 is an acquire operation on o and i_1 and i_2 gets executed by different threads.
- There exists i_3 such that $i_1 \xrightarrow[P]{\text{hb}} i_3$ and $i_3 \xrightarrow[P]{\text{hb}} i_2$.

B. Inferring Interleaving Points

Our goal is to analyze an execution trace of a multithreaded program and to identify the program locations at which a thread interleaving, i.e., context switch, may lead to a failure of a correctness property such as memory safety or an invariant that must hold at a specific program location. We call such program locations *interleaving points*. In our approach, the process of inferring interleaving points works in three steps for each execution path: 1) Collecting data-flow and control-flow facts about the instructions executed on that path, 2) Creating thread access information for each property relevant object discovered in Step 1), and 3) Filtering out accesses that must not happen in parallel. Below, we present the details about each step.

1) Step 1: Collecting data-flow and control-flow facts: Figure 4 shows the inference rules that are applied to the instructions that get executed on an execution path P. All the rules are applied on the instructions executed on a path P until a fixpoint is reached for the property relevant memory objects O and the property relevant instructions PR.

Rule ARRAY ACCESS analyzes expressions that are used in array element accesses. We analyze the trace to find instructions that get involved in defining the expression as well as the objects that are accessed in the construction of the expression. Such objects and instructions are included in the error relevant object and instruction sets, respectively. For example, the array index expression at line 27 in Figure 3 is a constant, and, hence, it would not yield any memory objects or instructions whereas the array access at line 29 would mark the global variable $data$ and the write operation at line 19 as property relevant depending on the order of scheduling, i.e., if line 19 gets executed before line 29 on the analyzed path.

Rule DEALLOC analyzes the pointer expressions that are passed to deallocation functions, which we denote with $free(x)$. As long as the pointer expression x refers to a globally visible object, the object and the deallocation instructions are included in the error relevant object and instruction sets, respectively. An important detail to consider is that, we record the deallocation instruction as a write operation as it deallocates the object and interferes with any subsequent read operation. For example, both the deallocation operation on line 12 as well as the read operation at line 27 that depends on line 12 are marked as property relevant.

Rule ALLOC analyzes the addresses that are returned by allocation functions such as `malloc`, which we denote with $x \leftarrow malloc(size)$. As long as the pointer expression x refers to a globally visible object, the object and the allocation instruction are included in the error relevant object and instruction sets, respectively. An important detail to consider is that, we record the allocation instruction as a write operation as it creates the object and interferes with any subsequent read and write operations. For example, because of the deallocation operation at line 12, the dynamic memory operation and the memory object created at line 36 are both marked as property relevant.

Rule READ and WRITE analyze the load instructions and the store instructions, respectively. As long as the accessed object is globally visible, the object and the access instruction are included in the error relevant object and instruction sets, respectively. For example, the read operations at lines 11, 12, 19, 21, 27, 29, and 45 and the write operations at lines 19, 21, 30, and 36 would be marked as property relevant. Similarly, the variables $mutex$, $data$, $name$, $address$, $letter$, ind, and $zipcode$ would be marked as property relevant.

Rule TARGET incorporates functions that are deemed important by the user. $Target$ denotes the set of function names provided by the user as property relevant, and we will refer to it as the target list. At callsites of these functions, the expressions that correspond to the arguments are analyzed and the instructions that define these expressions and memory objects accessed during the computations of these expressions are included in the error relevant instruction and object sets, respectively. In our approach, by default we consider the `assert` and `abort` functions as part of the target list. However, the user can specify additional function names to be used as targets.

Rule CNTFLOW1 analyzes the branch instructions with multiple targets. If the segment of the execution trace that gets executed after the branch instruction have error relevant instructions recorded, we also include the instructions that define the branch condition and the memory objects that are accessed during the computation of the expressions in the error relevant instruction and object sets, respectively. For example, consider the `if` statement at line 11. Assume that the condition evaluates to false and the instructions that get executed under the false branch condition includes a property relevant instruction, e.g., the read operation at line 29. This makes any objects accessed for evaluating the loop condition, i.e., $data$, also property relevant.

Rule CNTFLOW2 also analyzes the branch instructions with multiple targets. However, the goal of this rule is to statically analyze the instructions that are reachable by the branch target, t_2, that was not executed on the current path. Since we do not have runtime information about such in-

$$[\text{ARRAY ACCESS}]\ \frac{P \equiv P'; A[exp] \quad P' \vdash O', PR'}{P \vdash O' \cup Mem(exp), PR' \cup Def(exp)} \qquad [\text{DEALLOC}]\ \frac{P \equiv P'; free(x) \quad globallyVisible(PointsTo(x)) \quad P' \vdash O', PR'}{P \vdash O' \cup \{PointsTo(x)\}, PR' \cup \{write\ x\}}$$

$$[\text{ALLOC}]\ \frac{P \equiv P'; x \leftarrow malloc(size) \quad globallyVisible(PointsTo(x)) \quad P' \vdash O', PR'}{P \vdash O' \cup \{PointsTo(x)\}, PR' \cup \{write\ x\}} \qquad [\text{READ}]\ \frac{P \equiv P'; read\ x \quad globallyVisible(x) \quad P' \vdash O', PR'}{P \vdash O' \cup \{x\}, PR' \cup \{read\ x\}}$$

$$[\text{WRITE}]\ \frac{P \equiv P'; write\ x \quad globallyVisible(x) \quad P' \vdash O', PR'}{P \vdash O' \cup \{x\}, PR' \cup \{write\ x\}} \qquad [\text{TARGET}]\ \frac{P \equiv P'; f(args) \quad f \in Target \quad P' \vdash O', PR'}{P \vdash O' \cup \bigcup_{a \in args} Mem(a), PR' \cup Def(a)}$$

$$[\text{CNTFLOW1}]\ \frac{P \equiv P_1; branch\ exp, t_1, t_2; P_2 \quad P_1 \vdash O_1, PR_1 \quad P_2 \vdash O_2, PR_2 \quad PR_2 \neq \emptyset}{P \vdash O_1 \cup O_2 \cup Mem(exp), PR_1 \cup PR_2 \cup Def(exp)}$$

$$[\text{CNTFLOW2}]\ \frac{P \equiv P_1; branch\ exp, t_1, t_2; P_2 \quad t_1\ executed\ in\ P_2 \quad isTargetRelevant(t_2) \quad P_1 \vdash O_1, PR_1 \quad P_2 \vdash O_2, PR_2}{P \vdash O_1 \cup O_2 \cup Mem(exp), PR_1 \cup PR_2 \cup Def(exp)}$$

Fig. 4: The rules used by **ComputeDFCFFacts** for the collection of data-flow and control-flow facts for error relevant instructions.

$$\frac{P \vdash \{x\} \cup O, \{write_i\ x\} \cup PR \quad tid(write_i\ x) = t_k}{\{(t_k, write_i\ x)\} \subseteq WM[x]}$$

$$\frac{P \vdash \{x\} \cup O, \{read_i\ x\} \cup PR \quad tid(read_i\ x) = t_k}{\{(t_k, read_i\ x)\} \subseteq RM[x]}$$

Fig. 5: The rules used by **ComputeWriteReadMaps** for constructing the write (WM) and read (RM) maps for each error relevant memory object.

structions, we determine the relevance based on whether there are any reachable memory allocation/deallocation instructions and any callsites that involve functions from the target list. We use the predicate $isTargetRelevant(t)$ to refer to the result of this static analysis stage, which is performed intra-procedurally and can be configured with respect to the depth of the search that is expressed in terms of the number of basic blocks analyzed. We use a default value of 1, which means the immediate basic block that was not executed, t, is analyzed only. If so, we include the instructions that define the branch condition expression and the memory objects that are accessed during the computation of the expression in the error relevant instruction and object sets, respectively. For example, if the if statement at line 9 evaluates to false, with the CNTFLOW2 rule, the global variable $data$ would be considered as property relevant as there is a callsite of the `free` function in the branch that was not executed. However, we would not be able to reason about the objects that get accessed at that callsite.

We record each instruction along with the context information. This is because an instruction of a function may be property relevant in one calling context but it may not be property relevant in others. So, we capture the stack trace to represent the calling context of each instruction. However, since the property relevant instructions will be eventually used as interleaving points in other paths, we scrub path specific information such as the actual arguments of the functions from the stack traces that we use as context information.

2) Step 2: Creating thread access maps on property relevant objects: Recall that in Step 1, we record each property relevant instruction as either a write operation or a read operation. In Step 2, for a given execution path P we create thread access maps, WM and RM, for the write accesses and the read accesses, respectively, based on the property relevant objects that have been collected in Step 1. Figure 5 shows the rules for creating these maps. Once each instruction gets executed on P, we record the thread that executes it. We use the notation t_i to refer to a thread instance with the unique identification number i and the function $tid(inst)$ to refer to the thread instance that executes the instruction $inst$ on P.

3) Step 3: Filtering out accesses that must not happen in parallel: In this step, we use the read and write access maps that were created in Step 2 to identify conflicting accesses, i.e., the write-write and write-read accesses that are performed on common memory objects. Our goal is to identify those pairs of accesses that may be subject to a different order of execution on some alternative schedule. In this step, we generate a set of shared objects, $Shared$, that may be accessed by different threads in a different order in some alternative schedule and a map from these shared memory objects to the set of instructions that may be reordered, IPM. So, $IPM[x]$ denotes the set of conflicting instructions that access shared object x and may be used as interleaving points. If we identify that the two conflicting accesses do have only one possible ordering, which was realized on the current execution path P, we ignore these access pairs as they would not happen in parallel in any execution path that satisfies the same data constraints that hold on P. So, a memory access a is only filtered out if there are no other conflicting memory accesses on P that may happen in parallel with a.

We define two rules: INSYNC and NOTINSYNC. In both rules, for conflicting operations, $a_j\ x$ and $a_i\ x$, we check if

$$[\text{INSYNC}]\ \frac{\begin{array}{r} P \vdash O, PR \quad x \in O \quad (t_k, a_j\ x) \in WM[x] \quad (t_m, a_i\ x) \in WM[x] \cup RM[x]\ t_k \neq t_m \quad inCommonSync(a_i\ x, a_j\ x) \\ a_i\ x \not\xrightarrow[P]{hb} create\ t_k \quad term\ t_m \not\xrightarrow[P]{hb} a_j\ x \quad a_j\ x \not\xrightarrow[P]{hb} create\ t_m \quad term\ t_k \not\xrightarrow[P]{hb} a_i\ x \end{array}}{\{a_j\ x, a_i\ x\} \cup Sync(a_j\ x) \cup Sync(a_i\ x) \subseteq IPM[x], \quad x \in Shared}$$

$$[\text{NOTINSYNC}]\ \frac{\begin{array}{r} P \vdash O, PR \quad x \in O \quad (t_k, a_j\ x) \in WM[x] \quad (t_m, a_i\ x) \in WM[x] \cup RM[x] t_k \neq t_m \quad \neg inCommonSync(a_i\ x, a_j\ x) \\ a_i\ x \not\xrightarrow[P]{hb} create\ t_k \quad term\ t_m \not\xrightarrow[P]{hb} a_j\ x \quad a_j\ x \not\xrightarrow[P]{hb} create\ t_m \quad term\ t_k \not\xrightarrow[P]{hb} a_i\ x \end{array}}{\{a_j\ x, a_i\ x\} \subseteq IPM[x],, \quad x \in Shared}$$

Fig. 6: The rules used by **UpdateInterleavingPoints** for inferring interleaving points that may create error revealing schedules.

there are any happens-before ordering between them due to thread creation or thread join operations. If so, we continue checking for other pairs of accesses. Otherwise, we consider the pair of accesses as interleaving relevant. However, we need to do an additional check on whether these two accesses are embedded within synchronization blocks that access the same lock object. This is because switching from one thread to the other when the former is holding a lock that the latter may try to acquire may create unnecessary switching overhead. As once the latter one gets blocked another thread will need to be scheduled. So, if the two conflicting instructions are within related synchronization blocks, in rule INSYNC, we find all the acquire instructions that precede the memory accesses without a matching release operation that also comes before the memory access. Similarly, we find the matching release operations for those acquire instructions. Let $Sync(i)$ denote the acquire and release pairs that enclose instruction i. We include $Sync(a_j\ x)$ and $Sync(a_j i\ x)$ in $IPM[x]$ along with the conflicting instructions. Otherwise, in rule NOTINSYNC, only the conflicting instructions, $a_j\ x$ and $a_i\ x$, are included in $IPM[x]$. In both cases, we include x in $Shared$.

For example, lines 10, 13, 18, and 20 would be marked as property relevant due to the conflicting accesses on lines 11 and 19. On the other hand, since the conflicting instructions on lines 21 and 30 are not enclosed by the acquire and release operations of a common lock, lines 21 and 30 would be marked as property relevant.

C. *Selective Symbolic Execution*

In this section, we present the technical details of performing selective symbolic execution for multithreaded code. Our goal is to direct symbolic execution to select thread schedules that are likely to reach some error over those that may not. Below, first we provide some background on symbolic execution, and then present how we adopt baseline symbolic execution to achieve our goal of detecting errors in multithreaded code faster.

1) Background on Dynamic Symbolic Execution: Dynamic symbolic execution is a static program analysis technique that can reason about symbolic inputs. The word "dynamic" refers to the fact that concrete and symbolic values can be mixed on an execution path. Dynamic symbolic execution has two major flavors: concolic and execution-tree generation based. A symbolic execution engine typically interprets the instructions of an intermediate language, such as the LLVM IR [16], so that expressions that involve symbolic values are manipulated according to the semantics of the instruction. In this paper, we focus on the execution-tree generation based approach. When interpreting conditional branch instructions with symbolic branch conditions, a symbolic execution engine checks the satisfiability of the branch condition for each target using an SMT solver and to simulate each feasible target it generates a separate path. On each path it conjoins the symbolic branch conditions to generate the *path constraint*. So, the symbolic execution engine generates a tree of symbolic execution paths or states, where the internal nodes with multiple children denote branching points and each leaf node denotes a completed execution corresponding to an equivalence class of the input space. A challenge in symbolic execution is the well-known path explosion problem as the tree of executions may grow exponentially with the increasing number of branching instructions. So, symbolic execution is typically configured to run up to some timeout value.

Algorithm 1 The main algorithm as implemented by the SIFT tool.

```
1: SIFT($Prog$: Multi-threaded Program, $pmode$: SINGLETONS, COMMON, ONE, $timeout$: Real, $N$: Natural)
2: $(IP, Shared) \leftarrow (\emptyset, \emptyset)$
3: $partitions \leftarrow \{\emptyset\}$
4: for $i$: 1 to $N$ do
5:     for each $p \in partitions$ do
6:         $(TIP, TShared) \leftarrow ExploreInferSelective(Prog, p, timeout)$
7:         $IP \leftarrow IP \cup TIP$
8:         $Shared \leftarrow Shared \cup TShared$
9:     end for
10:    if $pmode$ is COMMON then
11:        $partitions \leftarrow \{IPM[x] \mid x \in Shared\}$   ▷ Initialize partitions
12:        while exists $s_1, s_2 \in partitions$ s.t. $s_1 \cap s_2 \neq \emptyset$ do   ▷ Common int. points
13:            $partitions \leftarrow partitions \cup \{s_1 \cup s_2\} \setminus \{s_1, s_2\}$   ▷ Merge
14:        end while
15:        $partitions \leftarrow sortNonDecreasing(partitions)$
16:    else
17:        if $i = N - 1$ and $pmode$ is SINGLETONS then   ▷ Apply each int. point separately
18:            $partitions \leftarrow \bigcup_{x \in Shared} \{\{ip\} \mid ip \in IPM[x]\}$
19:        else   ▷ $pmode$ is ONE or earlier steps for SINGLETONS
20:            $partitions \leftarrow \{\bigcup_{x \in Shared} IPM[x]\}$
21:        end if
22:    end if
23: end for
```

2) The main algorithm: SIFT: Algorithm 1 shows our approach at a high-level. It takes as input a multithreaded program, $Prog$, the mode of partitioning the interleaving points, $pmode$, a timeout value, $timeout$, and the number of steps, N. In each step, **SIFT** performs symbolic execution on the given multithreaded program by selectively generating alternative scheduling of the threads based on the current set of interleaving points and generates new interleaving points to be considered for the next step.

SIFT can be configured to use the interleaving points according to one of the three modes of partitioning, which can be ONE, $SINGLETONS$, or $COMMON$. In the ONE mode, all interleaving points are combined in a single partition. Although this mode provides the most exhaustive exploration of the thread schedulings with respect to the interleaving points, it is the most costly one due to the number combinations. In the $SINGLETONS$ mode, SIFT generates as many partitions as the number of interleaving points so that each of them can be considered as the only interleaving point during symbolic execution. This mode can be more efficient than the ONE mode. However, it can only be effective for generating error paths that require only a single voluntary context switch, i.e., one that is not due to a blocking operation, at a specific program location. In the $COMMON$ mode, a separate partition is created for each shared object. However, if two partitions somehow have common interleaving points then they get merged into one.

For example, for our running example in Figure 3, the set of shared objects that are used to identify the partitions consists of $data$, $name$, and ind. The read operation at line 21 and the write operation at line 30 are included in one partition due to the shared object ind and the remaining property relevant instructions, lines 10, 13, 18, 20,, 26, 28, 29, 36, end up being placed in another partition due to the shared objects $data$ and $name$. The reason instructions that access $data$ and those that access $name$ get merged in one partition is due to being enclosed by common acquire and release instructions.

The $COMMON$ mode can have better coverage than the $SINGLETONS$ mode and if multiple partitions can be generated, it can achieve better performance than the ONE mode. In this mode, we also sort the partitions in nondecreasing order of their sizes.

So, at every step **SIFT** goes through each partition and uses the interleaving points in that partition to reach the error state and at the same time it generates the interleaving points for the next state. The $SINGLETONS$ mode is different from the ONE and $COMMON$ modes as it only gets applied in the last mode. This means that when configured in the $SINGLETONS$ mode, **SIFT** works in the ONE mode in all steps except the last one.

3) Selective Scheduling Exploration: It is intractable to generate all possible thread interleavings during the analysis of a multithreaded program. Algorithm 2, instead, uses the given set of interleaving points, IP, to explore the thread interleaving space of the given program $Prog$. Similar to baseline symbolic execution, it first creates an initial symbolic execution state, in which the global variables with initializers have been initialized, the stack frame has the stack frame for the `main` function, the program counter, *pc*, is initialized to the

Algorithm 2 An algorithm for inferring thread interleaving points from the symbolic execution of a program until a bound is reached.

```
1: ExploreInferSelective($Prog$: Multi-threaded Program, $IP$: Set of Interleaving Points, $timeout$: Real): (Set of Interleaving Points, Set of Shared Objects)
2: $state \leftarrow init(Prog)$
3: $States \leftarrow \{state\}$
4: Let $IP \leftarrow \emptyset$
5: while $States$ not empty and $timeout$ not reached do
6:     $cur \leftarrow ChooseNext(States)$
7:     if $s.thread$ is blocked then
8:         $succs \leftarrow ExecuteNextInstConservative(s, s.thread)$
9:     else
10:        $succs \leftarrow ExecuteNextInstInterleave(s, s.thread, IP)$
11:    end if
12:    for each $suc \in succs$ do
13:        if bug manifested in $suc$ then
14:            Report bug
15:        else
16:            if $suc$ terminated then
17:                $Term \leftarrow Term \cup \{suc\}$
18:            else
19:                $States \leftarrow States \setminus \{cur\} \cup \{suc\}$
20:            end if
21:        end if
22:    end for
23: end while
24: $(IP', Shared') \leftarrow (\lambda x.\ \emptyset, \emptyset)$
25: for each $state \in States \cup Term$ do
26:     $(O, PR) \leftarrow$ ComputeDFCFFacts($state.Path$)
27:     $(WM, RM) \leftarrow$ ComputeWriteReadMaps($state.Path, O, PR$)
28:     $(TIP, TShared) \leftarrow$ UpdateInterleavingPoints($Prog$, $state.Path$,$WM$,$RM$)
29:     $(IP', Shared') \leftarrow (\lambda x.IP'[x \leftarrow IP'[x] \cup TIP[x]], Shared' \cup TShared)$
30: end for
31: return $(IP', Shared')$
```

Algorithm 3 The algorithm that symbolically executes an instruction for a multi-threaded program and keeps executing the same thread until it gets blocked.

```
1: ExecuteNextInstConservative($s$: Execution State, $t$: Thread): set of Execution States
2: if $isEnabled(s, t)$ is false then
3:     $i \leftarrow 0$
4:     while $i < s.queue.size()$ do
5:         $t' \leftarrow s.queue.remove()$
6:         if $isEnabled(s, t')$ is true then
7:             break
8:         else
9:             $s.queue.add(t')$
10:            $t' \leftarrow t$
11:        end if
12:        $i \leftarrow i + 1$
13:    end while
14:    if $t' = t$ then
15:        Report deadlock and terminate path
16:    else
17:        $s.queue.add(t)$
18:        $s.thread \leftarrow t'$
19:    end if
20: end if
21: return ExecuteNextInst($s$, $s.thread.stack$, $s.thread.pc$)
```

first instruction of the `main` function, and the path constraint , PC, is initialized to true. We have extended a symbolic execution state to accommodate multiple threads. Each thread, including the main thread, has its own stack and the program counter and is identified by a unique integer. The state keeps track of the id of the current thread that is in execution. It keeps other threads in a queue and records whether a thread is enabled or not. It keeps track of the symbolic execution states in $States$ and in each iteration of the main loop, it chooses one state according to some scheduling policy, executes the next instruction for the current thread.

Algorithm 4 The algorithm that symbolically executes an instruction for a multi-threaded program and creates alternative thread schedules at the interleaving points.

```
1: ExecuteNextInstInterleave($s$: Execution State, $t$: Thread, $IP$:Interleaving Points): set of Execution States
2: if $s.thread.pc \in IP$ is true then    ▷ An interleaving point
3:     $succ \leftarrow \emptyset$
4:     for each thread $t' \neq s.thread$ do    ▷ Schedule every other enabled thread
5:         if $isEnabled(s, t')$ then
6:             $s' \leftarrow s.copy()$
7:             $s'.queue.remove(t')$
8:             $s'.queue.add(s'.thread)$
9:             $s'.thread \leftarrow t'$
10:            $succ \leftarrow succ \cup \{s'\}$
11:        end if
12:    end for
13:    return $succ \cup$ ExecuteNextInst($s$, $s.thread.stack$, $s.thread.pc$)
14: else ▷ Do not schedule any other thread and keep executing the current one
15:    return ExecuteNextInst($s$, $s.thread.stack$, $s.thread.pc$)
16: end if
```

The execution of an instruction involves thread scheduling, which depends on two factors: 1) whether the current thread is enabled or not, and 2) whether the instruction to be executed is an interleaving point. If the current thread is not enabled, Algorithm 2 calls Algorithm 3 to choose the next enabled thread from the queue and executes the instruction mostly like baseline symbolic execution except the fact that any updates to the local variables are performed on the relevant thread's stack. If the thread is enabled then Algorithm 2 calls Algorithm 4, which generates an alternative schedule in a copy of the current execution state if the executed instruction is an interleaving point. An alternative schedule based on an interleaving point is generated before the interleaving point gets executed as shown in Algorithm 4. However, there is an exception to this rule: the release operations. This is because scheduling another thread while the lock is held by the current thread would just incur extra overhead in terms of extra context switching if the scheduled thread would attempt to acquire the same lock. We abstract away this implementation detail in Algorithm 4 to keep it concise. If the instruction to be executed is not an interleaving point then the instruction gets executed similar to the baseline symbolic execution while updating thread related data structures properly.

After generating the symbolic execution states, Algorithm 2 goes over each state and analyzes the execution path to perform data-flow analysis as explained in Section III-B. Finally, it returns the interleaving point map and the set of shared objects to be used in the next step in Algorithm 1.

TABLE I: Comparing partitioning modes of SIFT on SV-COMP and CVE benchmarks in terms of the number of times yielding the minimum detection time within a timeout of 500 secs.

N	S	PI	SINGLETONS	COMMON	ONE
2	RC	10	13	4	1
2	RC	100	13	2	5
2	D	10	5	5	11
2	D	100	4	9	6
3	RC	10	3	11	6
3	RC	100	6	10	4
3	D	10	6	11	11
3	D	100	10	10	3
		Total	60	62	47

IV. Evaluation

We have implemented our approach on top of the KLEE symbolic execution engine [5]. SIFT currently works on the LLVM 3.8 bitcode. We have run our experiments on an Intel Xeon CPU 2.30GHz with 256 GB memory. We have applied SIFT to the Linux device driver benchmarks from SV-COMP [3] and to the CVE benchmarks that were used to evaluate ConVul [7], [2]. For each mode of partitioning the interleaving points, SINGLETONS, COMMON, and ONE, we have explored various configurations in terms of the number of steps (see the input to Algorithm 1), N, the path scheduling algorithms, S, as provided by the symbolic execution engine (`D` for Depth-First Search and `RC` fr Random and coverage based), and the maximum number of paths analyzed to infer the interleaving points, PI. Motivated by the empirical evidence [20], [19], [18] that only few thread interleavings manifest concurrency bugs, we bounded the maximum number of context switches to 3 on each symbolic execution path. For each configuration and benchmark combination, we ran SIFT three times and computed the average time to detect the error.

While Table I shows a summary of overall results, Tables II and III show the lowest average error detection times in seconds for each benchmark, for $N = 2$ and $N = 3$, respectively, along with the configuration that yielded those times. Among the SV-COMP benchmarks, `race-1_1-join` and `race-4_1-thr` are known not to have any bugs. Therefore, SIFT does not report any errors for these benchmarks as denoted by "-". In the following subsections, we evaluate our approach with regards to various research questions.

A. In which partitioning mode SIFT detects the errors the fastest?

Table I shows the number of times each partitioning strategy yields the fastest detection for each configuration. as well as the total numbers across all configurations. Comparing the total number of cases, the COMMON and SINGLETONS modes perform better than the ONE mode. Also, as shown in Table III.a, for the SV-COMP benchmark `race-4_2-thr`, SIFT could detect the error within 500 secs only in the COMMON mode. Although, the SINGLETONS mode, perform close to

TABLE II: Comparing various configurations that yield the lowest average error detection times for the three modes of **SIFT**. Timeout=500 secs. **N=2**. ET, S, and PI denote the error detection time in seconds, the path scheduling algorithm, and the number of paths analyzed to infer interleaving points, respectively. RC denotes random + coverage based scheduling and D denotes depth-first search. - means the error could not be detected.

a) Error detection for SV-COMP benchmarks.

Benchmark	**SINGLETONS**			**COMMON**			**ONE**		
	ET (s)	**S**	**PI**	**ET (s)**	**S**	**PI**	**ET (s)**	**S**	**PI**
race-1_1-join	-	-	-	-	-	-	-	-	-
race-1_2-join	0.19	D	100	0.07	D	100	**0.06**	D	10
race-1_3-join	0.19	D	100	**0.06**	D	100	**0.06**	D	10
race-2_1-con.	0.87	D	10	**0.22**	D	100	**0.22**	D	10
race-2_2-con.	**0.18**	D	100	0.19	D	100	**0.18**	D	10
race-2_3-con.	0.21	D	100	**0.18**	D	100	**0.18**	D	10
race-2_4-con.	0.21	D	100	**0.19**	D	100	**0.19**	D	10
race-2_5-con.	**0.21**	D	100	0.22	D	100	0.22	D	10
race-3_1-con.	0.90	D	100	0.63	D	100	**0.21**	D	10
race-3_2-con.	0.35	D	100	0.58	D	100	**0.20**	D	10
race-4_1-thr.	-	-	-	-	-	-	-	-	-
race-4_2-thr.	-	-	-	-	-	-	-	-	-

b) Error detection for real world CVE benchmarks

Benchmark	**SINGLETONS**			**COMMON**			**ONE**		
	ET (s)	**S**	**PI**	**ET (s)**	**S**	**PI**	**ET (s)**	**S**	**PI**
CVE_2009_3547	0.20	D	100	**0.14**	D	100	**0.14**	D	100
CVE_2011_2183	8.32	D	10	**6.20**	D	100	16.14	D	100
CVE_2013_1792	-	-	-	-	-	-	-	-	-
CVE_2015_7550	0.72	RC	100	**0.42**	D	10	0.44	D	100
CVE_2016_1972	-	-	-	-	-	-	-	-	-
CVE_2016_1973	19.96	D	10	**4.31**	RC	100	4.99	D	10
CVE_2016_7911	0.38	D	10	0.28	D	100	**0.16**	D	100
CVE_2016-9806	0.56	D	100	2.07	D	10	**0.55**	D	10
CVE_2017_15265	**2.98**	D	10	5.29	D	10	-	-	-
CVE_2017-6346	1.40	D	100	**1.36**	D	100	1.49	D	100

TABLE III: Comparing various configurations that yield the lowest average error detection times for the three modes of **SIFT**. Timeout=500 secs. **N=3**.

a) Error detection for SV-COMP benchmarks.

Benchmark	**SINGLETONS**			**COMMON**			**ONE**		
	ET (s)	**S**	**PI**	**ET (s)**	**S**	**PI**	**ET (s)**	**S**	**PI**
race-1_1-join	-	-	-	-	-	-	-	-	-
race-1_2-join	**0.07**	D	100	**0.07**	RC	10	**0.07**	D	100
race-1_3-join	0.07	D	100	**0.06**	D	10	0.07	D	100
race-2_1-con.	**0.22**	D	100	**0.22**	D	100	**0.22**	D	10
race-2_2-con.	0.19	D	100	**0.18**	D	100	0.19	D	100
race-2_3-con.	**0.18**	D	100	**0.18**	D	100	0.19	D	10
race-2_4-con.	**0.19**	D	100	**0.19**	D	100	**0.19**	D	10
race-2_5-con.	**0.22**	D	100	**0.22**	D	10	**0.22**	D	10
race-3_1-con.	**0.21**	D	100	0.65	D	100	0.22	D	10
race-3_2-con.	**0.19**	D	100	0.58	D	100	**0.19**	D	10
race-4_1-thr.	-	-	-	-	-	-	-	-	-
race-4_2-thr.	-	-	-	**52.08**	RC	10	-	-	-

b) Error detection for real world CVE benchmarks

Benchmark	**SINGLETONS**			**COMMON**			**ONE**		
	ET (s)	**S**	**PI**	**ET (s)**	**S**	**PI**	**ET (s)**	**S**	**PI**
CVE_2009_3547	**0.14**	D	100	**0.14**	D	100	**0.14**	D	100
CVE_2011_2183	8.49	D	10	**0.91**	D	100	20.04	D	10
CVE_2013_1792	-	-	-	**8.25**	RC	100	24.62	D	100
CVE_2015_7550	0.44	D	100	**0.37**	D	100	**0.37**	D	100
CVE_2016_1972	**7.70**	D	10	13.46	D	10	7.72	D	10
CVE_2016_1973	5.36	D	10	**4.20**	RC	100	5.34	D	10
CVE_2016_7911	**0.14**	D	100	0.28	D	100	0.15	D	10
CVE_2016-9806	**0.55**	D	100	4.18	RC	100	0.56	D	10
CVE_2017_15265	5.29	D	100	**1.69**	D	10	-	-	-
CVE_2017-6346	1.47	D	100	**1.39**	D	100	1.48	D	100

the COMMON mode, SINGLETONS cannot detect errors that require multiple context switches. An example for this case is the CVE-2013-1792 benchmark that was presented in Section II. As shown in Figure 1, it requires two context switches. Therefore, as shown in Table II.b, SIFT cannot detect this error in the SINGLETONS mode while it can in the COMMON mode.

B. What is the impact of specific configurations on the partitioning modes?

As shown in Tables II and III, the depth-first search path scheduling yields the shortest error detection times. This can be explained as having more states with completed executions, and, hence, yielding a more complete picture of the dependencies among program statements. Although one may think that SIFT can be configured to use depth-first search, it should be noted that there could be cases such as the SV-COMP benchmark race-4_2-thr, where depth-first search algorithm may miss the error within the given time bound.

Although SIFT could detect the errors when $N = 2$ for most of the benchmarks, as shown in Tables II.b and III.b, SIFT could detect the error for benchmark CVE-2016-1972 only when $N = 3$. Similarly, as shown in II.a and III.a, SIFT could detect the error for SV-COMP benchmark `race-4_2-thr` only when $N = 3$.

C. What is the overhead of on-the-fly data-flow analysis in SIFT?

We have computed the time spent for on-the-fly data-flow analysis to infer the interleaving points, which includes computation of data-flow facts, read/write maps, happens-before analysis, and updating the partitions. We wanted to find out what percentage of the time to error detection is spent on the on-the-fly data-flow analysis.

It turns out that the overhead of on-the-fly data-flow analysis is higher when depth-first search is used compared to the path scheduling based on a combination of random and coverage based. For the CVE benchmarks, the overheads for the SINGLETONS, COMMON, and ONE modes with depth-first search are 18.74%, 21.20%, and 21.12%, respectively, and those with the random and coverage based scheduling are 15.37%, 14.46%, and 7.59%, respectively. For the SV-COMP benchmarks, the overheads for the SINGLETONS, COMMON, and ONE modes with depth-first search are 19.49%, 24.36%, 28.91%, respectively, and those with the random and coverage based scheduling are 16.94%, 8.82%, 6.61%, respectively. We think that the reason for the overhead difference is due to having more complete paths and more data-flow facts to glean and to process. While this increases the processing time, it also improves error detection and leads to faster detection.

Finally, SIFT could detect all the memory errors in these benchmarks including CVE-2011-2183, which could not be detected by ConVul as mentioned in [7]. We think that the success of SIFT relies on its property guided exploration of the search space. SIFT could detect this bug with all the partitioning modes, while achieving the fastest detection with the COMMON mode.

V. Related Work

It is impractical to exercise all possible scheduling scenarios to analyze multithreaded software. Previous work dealt with this challenge by designing a Domain Specific Language for developers to guide the exploration [8] and by synthesizing thread schedules from sequential tests [21] or from concurrent execution traces [10], [12]

Our approach is similar in spirit to all the approaches that analyze an execution trace to derive data-flow and synchronization dependencies among the events on an execution trace including Partial Order Reduction [22] and others [21], [12], [9]. However, SIFT's exploration is guided by property relevant interleaving points that are computed over the precise memory model provided by symbolic execution.

Maximal Causality Reduction (MCR) generates an equivalence class of traces with respect to happens-before relationship using constraint solving [10]. Although MCR avoids redundancy in generating interleaving scenarios, it is not property or failure directed. Maximal Path Causality (MPC) is combined with symbolic execution in [23] to effectively search the input and the schedule space. Our approach, on the other hand, searches the state space of input and interleaving scenarios in a systematic way by leveraging the path scheduling algorithms implemented by the symbolic execution engine. SIFT prioritizes scheduling relevant scenarios based on the property relevance.

Concolic execution is used in [12] to find scheduling-sensitive branches. The trace partitioning used in this work is complementary to our interleaving point partitioning.

An assertion guided pruning strategy is used to symbolically execute multithread programs in [9]. This approach uses an approximate weakest precondition computation for the explored states to filter out execution paths that are guaranteed not to reach the failure state. It also uses static slicing and dynamic partial order reduction to reduce the exploration stage. Our approach leverages the precise memory model of dynamic symbolic execution for points-to analysis and, therefore, it can handle non-standard pointer arithmetic that is found in systems code more precisely than standard points-to analysis techniques [24]. Also, our approach explores the state space in an incremental fashion to reduce the overhead of on-the-fly data-flow analysis.

Real execution traces, static analysis, and guided symbolic execution are combined in [17] to generate failing variants of the real execution traces and to identify root cause of the failures via differential trace analysis.

ConVul [7] analyzes dynamic execution traces to identify exchangeable events that are either not ordered with respect to a happens-before order or are transitively ordered through a small number of close proximity events. Our approach also analyzes access to shared objects when determining the interleaving points. SIFT could detect the vulnerabilities in all the CVE benchmarks that could be detected by ConVul and it can even detect the one, CVE-2011-2183, that was missed by ConVul. Furthermore, SIFT can support errors that are manifested via assertion failures in addition to the three types of memory vulnerabilities detected by ConVul.

UAF [11] analyzes dynamic execution traces to detect Use-After-Free vulnerabilities. It considers allocation, deallocation, and memory access operations as scheduling relevant and uses the maximal causality model to infer error revealing schedules. SIFT handles additional types of memory vulnerabilities as it supports a more generic definition of error relevant events.

ConCrash analyzes the stack traces of crashes to reproduce crash inducing thread interleavings [4]. SIFT, on the other hand, analyzes a state space without prior knowledge of the erronous behavior. Race detection approaches [14], [6], [13] may miss memory vulnerabilities that are due to logical errors and those that are not due to races.

VI. Conclusions

We have presented, SIFT, a property directed selective scheduling approach for symbolic execution of multithreaded code. SIFT leverages the precise memory model of symbolic execution for precise detection of property relevant data-flow and uses this information to identify interleaving points. We have equipped SIFT with three partitioning techniques to support a wide range of applications with different patterns of error revealing scheduling scenarios. SIFT has been implemented on top of the KLEE symbolic execution engine. Experimental results show that SIFT can effectively sift through a huge state space of scheduling scenarios and quickly detects memory vulnerabilities and assertion failures. In future work, we will extend SIFT with the handling of additional synchronization primitives.

VII. Acknowledgements

This work was funded by the US National Science Foundation under the CNS-1942235 award.

References

[1] CVE-2013-1792. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1792. Online; accessed 1 January 2021.

[2] Detecting Concurrency Memory Corruption Vulnerabilities. https://github.com/mryancai/ConVul. Online accessed 1 January 2021.

[3] SV-COMP: Linux Device Driver Races. https://github.com/sosy-lab/sv-benchmarks/tree/master/c/ldv-races. Online; accessed 1 January 2021.

[4] Francesco A. Bianchi, Mauro Pezzè, and Valerio Terragni. Reproducing concurrency failures from crash stacks. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 705–716. ACM, 2017.

[5] Cadar, Cristian and Dunbar, Daniel and Engler, Dawson. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224. USENIX Association, 2008.

[6] Yan Cai and Lingwei Cao. Effective and precise dynamic detection of hidden races for java programs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 450–461, 2015.

[7] Yan Cai, Biyun Zhu, Ruijie Meng, Hao Yun, Liang He, Purui Su, and Bin Liang. Detecting concurrency memory corruption vulnerabilities. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 706–717. ACM, 2019.

[8] Tayfun Elmas, Jacob Burnim, George C. Necula, and Koushik Sen. CONCURRIT: a domain specific language for reproducing concurrency bugs. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 153–164. ACM, 2013.

[9] Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, and Aarti Gupta. Assertion guided symbolic execution of multithreaded programs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 854–865. ACM, 2015.

[10] Jeff Huang. Stateless model checking concurrent programs with maximal causality reduction. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 165–174. ACM, 2015.

[11] Jeff Huang. UFO: predictive concurrency use-after-free detection. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 609–619. ACM, 2018.

[12] Jeff Huang and Lawrence Rauchwerger. Finding schedule-sensitive branches. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 439–449. ACM, 2015.

[13] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 754–768, 2019.

[14] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. Fast and accurate static data-race detection for concurrent programs. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, page 226–239, Berlin, Heidelberg, 2007. Springer-Verlag.

[15] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[16] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004.

[17] Nuno Machado, Brandon Lucia, and Luís E. T. Rodrigues. Production-guided concurrency debugging. In Rafael Asenjo and Tim Harris, editors, *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, pages 29:1–29:12. ACM, 2016.

[18] Madan Musuvathi and Shaz Qadeer. CHESS: systematic stress testing of concurrent software. In Germán Puebla, editor, *Logic-Based Program Synthesis and Transformation, 16th International Symposium, LOPSTR 2006, Venice, Italy, July 12-14, 2006, Revised Selected Papers*, volume 4407 of *Lecture Notes in Computer Science*, pages 15–16. Springer, 2006.

[19] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2005.

[20] Shaz Qadeer and Dinghao Wu. KISS: keep it simple and sequential. In William Pugh and Craig Chambers, editors, *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, pages 14–24. ACM, 2004.

[21] Malavika Samak, Murali Krishna Ramanathan, and Suresh Jagannathan. Synthesizing racy tests. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 175–185. ACM, 2015.

[22] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Efficient stateful dynamic partial order reduction. In Klaus Havelund, Rupak Majumdar, and Jens Palsberg, editors, *Model Checking Software, 15th International SPIN Workshop, Los Angeles, CA, USA, August 10-12, 2008, Proceedings*, volume 5156 of *Lecture Notes in Computer Science*, pages 288–305. Springer, 2008.

[23] Qiuping Yi and Jeff Huang. Concurrency verification with maximal path causality. In Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu, editors, *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 366–376. ACM, 2018.

[24] Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Pointer analysis for programs with structures and casting. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, page 91–103, New York, NY, USA, 1999. Association for Computing Machinery.