SSRD: Shapes and Summaries for Race Detection in Concurrent Data Structures

Xiaofan Sun
University of California at Riverside
Riverside, USA
xsun042@ucr.edu

Rajiv Gupta
University of California at Riverside
Riverside, USA
rajivg@ucr.edu

Abstract

Concolic testing combines concrete execution with symbolic execution to automatically generate test inputs that exercise different program paths and deliver high code coverage. This approach has been extended to multithreaded programs for exposing data races. Multithreaded programs frequently rely upon concurrent dynamic data structures whose implementations may contain data races that manifest only when certain dynamic *data structure shapes*, *program paths*, and *thread interleavings* are exercised. The lack of support for exploring different data structure shapes compromises the detection of races. This paper presents a *summarization*-guided approach for concolic testing capable of efficiently exploring different dynamic data structure shapes to expose data races. Via unit testing of key functions, function summaries are generated that capture data structure shapes that cause various function paths to be exercised. The shapes are captured in the form of pointer-pointee relations among symbolic pointers. By *reusing* function summaries during concolic testing, much of the overhead of handling symbolic pointers and dynamic objects in summarized functions is avoided. The summary also contains symbolic memory accesses and synchronization events that *guide* application-level concolic testing first to identify and then confirm potential data races. We demonstrate the efficiency and efficacy of our approach via experiments with multithreaded programs performing concurrent operations on four widely used dynamic data structures - Skip List, Unrolled Linked List, Priority Queue, and AVL Tree. It increases the number of races detected from 34 to 74 in total in comparison to Cloud9, and reduces both constraints solving time and number of constraints needed to be solved via summarization.

CCS Concepts: • **Software and its engineering → Software maintenance tools**.

ISMM '24, June 25, 2024, Copenhagen, Denmark

ACM ISBN 979-8-4007-0615-8/24/06.
https://doi.org/10.1145/3652024.3665505

Keywords: concolic testing, dynamic data structures, summarization, data race detection

ACM Reference Format:
Xiaofan Sun and Rajiv Gupta. 2024. SSRD: Shapes and Summaries for Race Detection in Concurrent Data Structures. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on Memory Management (ISMM '24), June 25, 2024, Copenhagen, Denmark.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3652024.3665505

1 Introduction

The ubiquity of multicore hardware has led to widespread use of multithreading in application software. Multithreaded programs typically employ data structures for maintaining shared state and threads must coordinate accesses to the shared state for correctness. The concurrent nature of shared data structures allows multiple operations to proceed in parallel; thus, mitigating performance bottlenecks. However, concurrency often leads to bugs in form of data races. A number of static analyses have been developed for automatically detecting data races including both flow-insensitive [8, 13, 15, 28]; and flow-sensitive [12, 22, 29, 37, 39, 60, 65] methods. Though static methods can be sound, users still have to manually confirm the detected data races. Without dynamic information, static methods are hard to automatically create a test input to reproduce the bug. Dynamic testing approaches include fuzzing [16, 35, 36, 55, 63] and concolic testing [14, 18, 38]. Fuzzing approaches, unsupported by a symbolic execution engine, cannot derive and solve conditions that cause execution to exercise branch outcomes necessary for reaching the race point. On the other hand, concolic testing is a powerful technique that has been extended to multithreaded programs for race detection [18, 23]. However, existing works for concolic testing of multithreaded C/C++ progarams to uncover data races have limitations when the program makes use of concurrent dynamic data structures [9, 10, 52] (e.g., a concurrent Skip List).

To expose a data race, it is typically essential to employ a data structure with a specific shape, together with a thread interleaving, that cause racing threads to follow paths with racing reads and writes of the data structure. Thus, concolic testing must be enhanced with the ability of efficiently explore dynamic data structure shapes and thread interleavings. Cloud9 [18] extends KLEE [14], an engine designed for single thread programs, with the POSIX thread model thus

allowing development of tools for detection of concurrency bugs in C/C++ programs. However, it lacks the capability for uncovering data races in the presence of concurrent data structures [9, 10, 52] that are *dynamic* in nature (e.g., Skip List) because it cannot systematically explore data structure shapes. Although Con2colic [23] can explore values for data fields within a data structure of a given shape and size, it cannot explore different data structure shapes. While CUTE [56] can explore data structure shapes, it does not support multithreaded programs and suffers from performance issues for large programs. Note that to uncover a data race we require coordinated exercising of paths by a pair of threads. Also, exploring dynamic data structure shapes and thread interleavings requires making all linking pointers symbolic. This causes the cost of path and thread interleaving exploration of concolic testing to further increase. Thus, effectively exploring data shapes and thread interleavings for a multithreaded program is still an open problem for detecting data races.

In this paper, to address the above limitations of capability and cost, we develop an approach that is both general and efficient (i.e., it can efficiently explore concurrent dynamic data structure shapes to uncover data races in C/C++ programs). Our approach is based upon the idea of function *summarization* where concolic unit testing of a function is used to generate a summary consisting of:

- *Path Conditions & Dynamic Data Structure Shapes* that represent symbolic constraints that must be satisfied to exercise a path in a summarized function; and
- *Lock/Unlock Sets* and *Read/Write Memory Accesses* that are used to identify pairs of paths with potential data races and guide concolic testing of the full application to, if possible, confirm the presence of a data races.

The summaries play an important role in improving the *efficiency* and *effectiveness* of concolic testing in the presence of dynamic data structures in two significant ways:

- *Reusing Paths Summaries.* When exploring a program path containing a call to a summarized function, say f, the overhead of concolic execution of f is reduced by *reusing* the symbolic structure shapes and expression representations generated for f during summarization. That is, the overhead associated with symbolic pointers is reduced because all actions performed during summarization need not be repeated during concolic testing; and
- *Coordinated Exploration of Data Structure Shapes.* A data race is exposed by simultaneous execution of a pair of paths by two threads. Using the summarization produced data structure shapes for two paths, we explore *integrated single non-conflicting shapes* that enable simultaneous exercising of the two paths of interest.

We have extended Cloud9 to support symbolic pointers and shape generation for concurrent dynamic data structures.

```
1  #define MAXLEVEL 2
2  typedef struct Node {
3    key_t key;
4    val_t value;
5    struct Node* next[MAXLEVEL];
6  } Node;
7
8  void foo(Node* node, key_t k, value_t v) {
9    // M paths before calling 'bar'
10   bar(node, k, v);
11 }
12
13 void bar(Node* node, key_t k, value_t v) {
14   // N paths before the following 'if' statement
15   if (node->key == k)
16     if (node->next[0] == node->next[1])
17       node->value = v;
18 }
```

Listing 1. An example illustrating the benefits of our approach.

By incorporating summarization, summary guided shape exploration, and summary reuse we have built a powerful and efficient concolic testing system for detection of data races in multithreaded programs. Our prototype also benefits from Cloud9 supported optimizations such as parallelization [18] and state merging [38]. We have evaluated this system by uncovering data races during execution of concurrent operations on multiple dynamic data structures. Our system is both effective and efficient.

The key contributions of this paper are:

- **Function Summary.** We propose a novel representation of function summaries that capture data structure shapes, branch conditions, memory accesses, and lock/unlock operations. By reusing the summaries we enable efficient and effective concolic testing.
- **Summary-guided Testing.** Function summaries help identify potential data races and efficiently guide the coordinated exploration of non-conflicting data structure shapes, derived from data shapes available in summaries, to confirm realizable data races.
- **Prototyping and Evaluation**: We implemented our system as an extension of Cloud9. Our experiments show that our system can detect races that Cloud9 cannot and when both systems can detect a data race, our system is more efficient than Cloud9. Moreover, via use of summaries, number of calls to the constraint solver are dramatically reduced.

Remainder of the paper is organized as follows. Section 2 gives an overview of our approach using an example showing summarization construction and invocation. Section 3 presents the details of key algorithms used in invocation and guided search. Section 4, 5 and 6 present experimental results, related work and conclusion.

2 Overview

We begin by providing an overview of our approach to concolic testing of multithreaded C/C++ programs that is effective in finding data races in the presence of concurrent dynamic data structures and is also efficient.

Data races may arise due to execution of code that requires certain conditional branch outcomes that can only be achieved when certain data value conditions, data shapes, and thread interleavings are exercised.

Data value conditions usually can be solved by the SMT solver during symbolic execution. For example, in Listing 1 function `bar` line 15, assuming the expression `node->key` is a symbolic value, for the branch outcome to be `false`, the condition `node->key!=k` can be solved. However, due to the lack of ability to set up symbolic pointers, and the heavy cost of exploring data shapes, the existing concolic/symbolic execution approach cannot effectively explore the data shapes.

If a data race requires specific **data shapes**, the existing approach may miss this data race as a required branch may not be taken. The line 16 shows a condition related to two pointers `node->next[0]` and `node->next[1]`, which is an invariant value based on the input data shape of `node`. Figure 1 shows two cases: the left one can expose the data race since the condition on line 16 will evaluate true - the input shape contains pointers that `next[0]` equals to `next[1]`; thus, the line of code that cause the race will be executed. For the other case on the right, the branch with race cannot be executed. Even though manually changing the input shape can avoid the problem in this example, but if there are more data races, one single input data shape will not be enough to expose all the data races. *Thus, we require the ability to explore the data shapes to detect all races.*

The **thread interleavings** also influence the branches taken, and thus must be explored if data races under a rarely taken branch is to be exposed. Figure 2 shows an example where only a specific thread interleaving exposes the data race. Consider a variable `x` which controls the branch to call function `foo`. Only if the statement `x = 1;` is between statements `x = 0;` and `if (x!=0)` so that the latter condition would become `true` and cause the both threads are calling function `foo` which finally cause the data race. *Thus, effectively finding the suitable thread interleaving to detect the races is also important.*

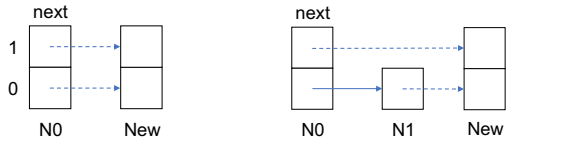

Figure 1. A data shape that can expose (left) and cannot expose (right) the race.

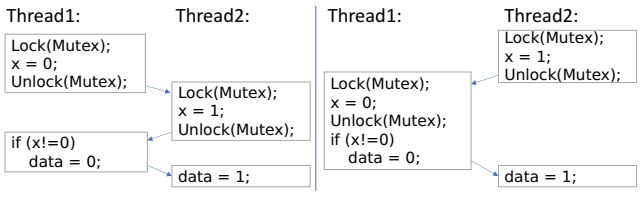

Figure 2. A thread interleaving that can expose (left) and cannot expose (right) the race.

Exploration of data shapes and thread interleavings involves performance challenges due to **large number of choices**. Consider the two functions in listing 1: `foo` with M paths, `bar` with N paths, and the `foo` will call `bar` at the end of each of the M paths. So, there are $N \times M$ paths to explore during program testing. We observe that there may be some parameter-irrelevant work if the parameter p is same or similar in different paths (e.g., symbolic expressions only have name changed). Thus, the repeated exploration of data shape and solving the constraints to exercise the path is redundant. However, if we can summarize `bar` and reuse the summary of `bar` during concolic testing of the full program, we can reduce the workload of testing `bar` from $N \times M$ paths to only N paths via the unit testing of function `bar`. The exploration of thread interleavings also has a similar issue. If there are K pairs of lock and unlock actions in $T1$, then at each of these thread $T2$ may scheduled or not leading to 2^K total interleavings. However, it may be the case that only when $T2$ is not scheduled at any of them causes the data race. If there is a directed search instead of random search for ordering the exploration of thread interleavings, desirable interleaving may be found faster.

Thus, our testing process consists of two steps, a *concolic unit testing step* followed by the *full program concolic testing step*. In the first step, using unit concolic testing *summarization* of individual functions that implement concurrent dynamic data structures is carried out. The generated summaries are also used to identify pairs of paths that contain *potential data races* for using directed search in the following step. In the second step - full program concolic testing, we start from the main function and test the whole program with the aim of generating inputs which confirm potential data races one by one. During this process, the data structure shapes contained in function summaries are *reused* to direct the exploration of non-conflicting data structure shapes for various paths in multiple threads. This directed search prunes the exploration of paths that cannot realize the potential data race. The *reuse of data structure shapes* and the *potential data races* information in summaries improves the efficiency with which summarized functions are repeatedly executed in testing.

Summarization via Concolic Unit Testing. Next, we will give the definition of function summaries. For a given function f, *concolic unit testing* of f is performed to build a decision tree model $\delta(f)$ which is the summary of f. The constructed decision tree corresponds to the *tested paths* in

the function such that each leaf node corresponds to a tested path from the start of the function to a return point.

In the *decision tree* $\delta(f)$, a node $n \in \delta(f)$ may represent a branch condition, a call to a function, a synchronization operation, or a return statement from the function. All nodes are annotated with summaries of shape $S(n)$ and memory access $M(n)$, defined as follows:

- $S(n)$ – the set of *pointer-pointee relations* among symbolic pointers representing the *data structure shape* that must be satisfied to enable the execution of $n \in \delta(f)$;
- $M(n)$ – the set of symbolic names (globals and parameters) and concrete addresses (locals) that correspond to the *read*/*write* memory accesses performed by n;

Also, branch nodes, call nodes, synchronization nodes, and return nodes are annotated with additional information $B(n)$, $C_f(n)$, $L(n)$, $V(n)$ respectively as described below.

- $B(n)$ – is the *branch condition* if n is a branch node;
- $C_{f'}(n)$ – contains name of function f' and the parameters for the call if n is a call node;
- $L(n)$ – contains *lock*/*unlock* action associated with n; and
- $V(n)$ – is the return value if n is a return node (it is empty if there is no return value).

Next, we briefly describe some key points about the unit testing that computes $\delta(f)$. First, all global variables as well as the parameters of f are treated as symbolic variables. Second, if f contains a function call, the return value of the callee function is treated as a symbolic variable and the testing of paths following the call are explored using the symbolic return value. In addition, since the callee may not be pure function (i.e., it can have side effects), the local variables that are passed as parameters to the callee are also treated as symbolic variables starting from the call point. Finally, the loops are handled by limiting the number of iterations and hence the number of paths they can generate. The code for loops in a function is augmented to limit the number of iterations.

Concolic Testing of Full Program For concolic testing of the full program that makes use of functions of the concurrent dynamic data structure, we first identify potential data races that may arise when multiple threads execute summarized functions that implement the concurrent dynamic data structure. Given a function f, and its summary $\delta(f)$, a set of *potential data races* R is computed. Each data race in R is of the form $r(\rho_i, \rho_j)$ where ρ_i and ρ_j are paths whose simultaneous execution by different threads may cause a data race according to $\delta(f)$. Symbolic variable set I represents all *user defined symbolic variables* for program P. The concolic tester, provided with I and R, explores executions in an attempt to confirm the data races in R.

Figure 3 shows the concolic testing performed to identify realizable data races in R via *search guided* by $\delta(f)$. The set of states maintained by the testing engine is shown as Φ. State set Φ contains all feasible execution states of the program P that concolic executor could reached. A state $\phi \in \Phi$ contains the current status of all threads and the complete address space for all memory objects.

Let us briefly consider how the search is carried out. Starting from the initial state ϕ_0 of the program, such as the entry point, the concolic executor explores paths, shapes, and thread interleavings when handling symbolic branch conditions and synchronization actions. For efficient path exploration using given input values I, different branch outcomes are forced and at the same time the corresponding states are pushed into the state queue. The thread interleaving exploration is guided by R as follows. A thread is made to execute a path ρ_i involved in a potential data race and another thread is made to explore all paths $\rho_j \in R$ such that $r(\rho_i, \rho_j)$ belongs to R. The data races confirmed during exploration are reported.

Our approach improves the efficiency of full program concolic testing by taking advantage of summaries. When a thread encounters a call to a function f for which summary $\delta(f)$ is available, *summary reuse* is invoked instead of calling f. This approach eliminates overhead of constructing symbolic expressions, gathering and checking constraints, and building data structure shapes that satisfy constraints. That is, some of work performed during unit testing of a function is reused instead of being repeated during each execution of the function during concolic testing.

Given the current program state ϕ just before node n in a summarized function, ϕ is updated by affecting it using the summary associated with n as follows:

1. *Shape Formation*: Given state ϕ, the shape summary $S(n)$ transforms the shape of the data structure giving state ϕ'.
$$\phi \overset{S}{\underset{n}{\Longrightarrow}} \phi'$$
2. *Memory Accesses*: A memory access summary includes reads from locations and writes to locations that copy symbolic or concrete values and changing state to ϕ''.
$$\phi' \overset{M}{\underset{n}{\Longrightarrow}} \phi''$$
3. *Updates based upon the type of node n*:
   - *Branch*- The expression $eval(B(n) = true, \phi)$ evaluates branch condition $B(n)$ and checks if it is true on ϕ. By evaluating both branches ($eval(B(n) = true, \phi'')$ and $eval(B(n) = false, \phi'')$), and adding appropriate path constraints, new states are represented as:
   $$\phi'' \overset{B=true}{\underset{n}{\Longrightarrow}} \phi''_t \quad or/and \quad \phi'' \overset{B=false}{\underset{n}{\Longrightarrow}} \phi''_f$$
   - *Call to f'*- Update state by invoking callee f', using callee's summary if available, $\phi'' \overset{C_{f'}}{\underset{n}{\Longrightarrow}} \phi'''$;

```
1  Node* search_node(Node* h, key_t k,
2                    int i, Node** pre) {
3      klee_assume(i >= 0 && i < MAXLEVEL);
4      Node* next = NULL;
5      if (h->next[i]!=NULL && h->next[i]->key<k)
6          next = h->next[i];
7      if (next != NULL)
8          return search_node(next, k, i, pre);
9      pre[i] = h;
10     if (i == 0) return h->next[i];
11     return search_node(h, k, i-1, pre);
12 }
13
14 bool insert(Node* h, key_t k, val_t v){
15     Node* prev[MAXLEVEL];
16     Node* curr = search_node(h, k, 1, prev);
17     if (curr != NULL && curr->key == k) {
18         if (curr == prev[1]->next[0])
19             curr->value = v;
20         return false;
21     }
22     Node* node = create(k, v);
23     int level = rand_level();
24     pthread_mutex_lock(&(prev[0]->mutex));
25     node->next[0] = prev[0]->next[0];
26     prev[0]->next[0] = node;
27     if (level == 1) {
28         node->next[1] = prev[1]->next[1];
29         prev[1]->next[1] = node;
30     }
31     pthread_mutex_unlock(&(prev[0]->mutex));
32     return true;
33 }
```

Listing 2. A Concurrent Skip List Example.

- *Return node-* Update state by mapping the return value to the caller $\phi'' \overset{V}{\underset{n}{\Longrightarrow}} \phi'''$; or
- *Synchronization-* Applying $L(n)$ to state ϕ'' leads to state ϕ''', $\phi'' \overset{L}{\underset{n}{\Longrightarrow}} \phi'''$, where executing threads state changes based upon the synchronization operation.

Finally, updating state ϕ due to a sequence of statements $n_i - n_j$ along a path is performed as follows.

$$\phi \overset{\delta_{i-j}}{\underset{n_i - n_j}{\Longrightarrow}} \phi' \quad = \quad \phi \overset{\delta_i}{\underset{n_i}{\Longrightarrow}} \phi_i \overset{\delta_{i+1}}{\underset{n_{i+1}}{\Longrightarrow}} \phi_{i+1} \cdots \phi_{j-1} \overset{\delta_j}{\underset{n_j}{\Longrightarrow}} \phi'$$

Updating state via use of summary is more efficient than the normal function call due to two reasons: a) The checking of constraints, creation of data shape, and construction of symbolic expressions that is carried out during concolic *unit testing* of a function is reused during concolic testing of the *full* program. b) Once symbolic expressions are simplified, some memory accesses are eliminated – if multiple writes are directed to same address, only the last write is needed. The computation of local variables may also be eliminated.

Illustration – Concurrent Skip List. Consider the code in Listing 2 which presents two operations for a skip list - `insert` (named f_i) and `search` (named f_s) for inserting in a ordered list and searching for a node corresponding to a key value. The function `search_node` (named f_{sn}) is a common

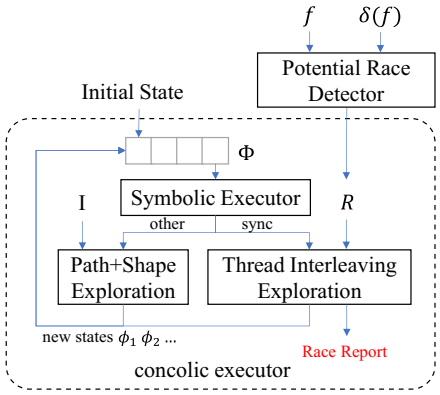

Figure 3. Exploring Path, Shape and Thread Interleaving.

function used by above functions to find the node which contains key k. Let us assume that two POSIX threads are processing those two functions correspondingly.

– *Example data race.* To allow illustration of our method, our implementation includes the following errors. During the insertion of a new node in the skip list, f_i finds the suitable position for insertion and collects all nodes that need to be modified into the list `prev` (line 15), and then enforces the change (line 25-30). In f_i, since the mutex lock only protects the write on `next[0]` field of `prev[0]` at line 26, the line 18 and 26 have a data race when `26: prev[0]->next[0]` is being written and `18: prev[1]->next[0]` is being read at the same time since `prev[0]` and `prev[1]` may represent the same node in some cases. However, the race condition for `prev[0]` and `prev[1]` pointing to the same node requires a more complex condition in f_{sn}, for which it is extremely hard to manually construct a suitable data structure shape. So that an approach can efficiently explore the data shapes to find a suitable one to pass to f_{sn} is required to detect this race. There are more data races in f_i and f_{sn} functions in different access patterns, but in our discussion we will only consider this race for illustration purposes.

– *Summary Representation.* Figure 5 shows the generated summary representation of functions f_i and f_{sn}. Note that trees include call nodes, branch conditions, synchronization operations, and return nodes. For the highlighted path `T F T` in this decision tree, the data shape generated is captured via pointer-pointee relations in Table 1 and the memory accesses summary is given in Table 2. In all cases pointer dereferencing implies a non null pointer and thus pointers point to other nodes in Table 1.

– *Summary construction.* Next, we present the construction of summaries – highlighted path `T F T` in the decision tree model for the function f_i in Figure 5 and the data shape and memory accesses information attached with each node shown in Table 1 and 2 correspondingly. This construction phase happens during the concolic unit testing of the function f_i. Before starting, we pass symbolic values to all its

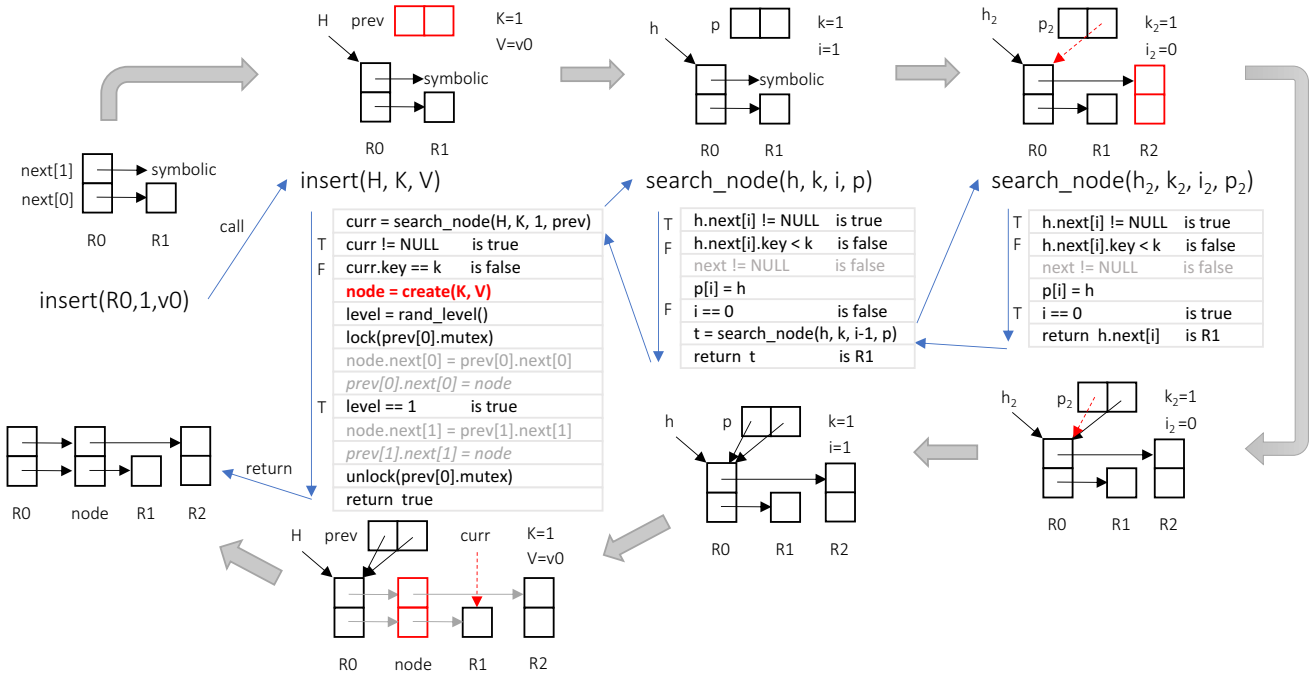

Figure 4. Data Structure Shapes at entry and return points of function invocations. *Gray statements are excluded from decision tree as their effect is captured by shape/memory access summaries or they are fully evaluated during unit testing.*

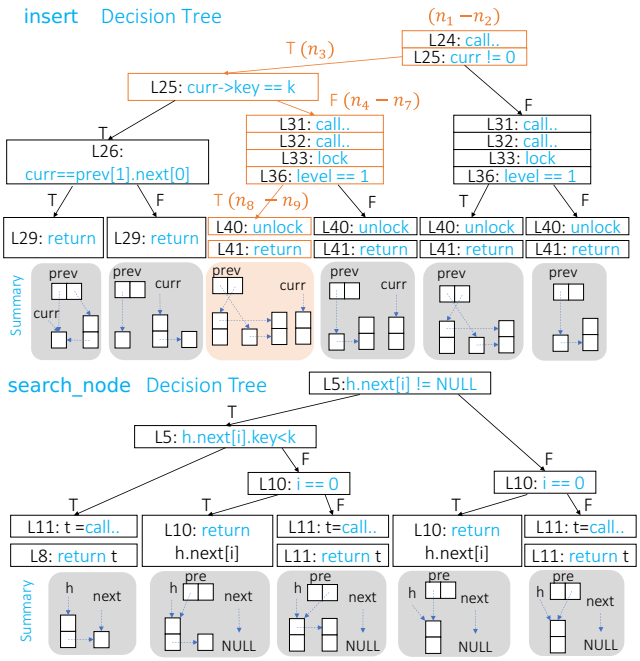

Figure 5. Summary Representations of f_i and f_{sn}.

non-pointer parameters, e.g. k and v, and adjustable symbolic pointer to its pointer parameters, e.g. h. Then, the executor will explore different paths and shapes to construct the decision tree. To make it easy to follow, since there are 9 nodes in the highlighted path (each orange box is a node; omit non-branch arrows), we name them n_1 - n_9.

When line 16 is reached, the call to f_{sn} is blocked since it is unrelated to the current decision tree. Instead, a new symbolic pointer is created as the estimated return value `curr`, and then continues to execute the current function. At the same time, we constructed a call node n_1 as the root node and annotated the information about this call in $C_{f_{sn}}(n_1)$. When line 17 is reached, it constructs the branch node n_2, and the branch condition `curr!=0` is annotated with $B(n_2)$. Similarly, the branch node n_2 is constructed and annotated with `curr->key==k` in $B(n_3)$. Note that in Table 1, a new pointee node `N0` is created during dereferencing, and the shape information is annotated in $S(n_3)$. The memory load action of symbolic address `curr->key` is recorded in $M(n_3)$, shown in Table 2. With the execution continuing, nodes n_4,n_5,n_6,n_7, and n_8 are constructed and generated with their corresponding annotations. Finally, we constructed n_9 during function f_i returning and recorded the returned value in $V(n_9)$. Now, the construction of the current highlighted path in decision tree $\delta(f_i)$ is finished. The executor will explore other paths and construct the complete summary $\delta(f_i)$ for the function f_i.

– *Summary invocation.* We illustrate the use of summaries in Figure 4 where a thread creates an initial data structure and invokes function f_i which in turn invokes summarized recursive function f_{sn}. All three function invocations use corresponding function summaries. The data structures at function call and return boundaries are shown. The changes involve mapping of symbolic names and also making changes to data structure to reflect the effect of the function via use

of summaries of data structure shapes and write memory operations. The statements along the path followed are also shown with statements that are not present in the decision tree are shown in gray such that the effect of these statements on the program state is achieved via use of shape and memory access summaries. The statements marked italics - `prev[0].next[0] = node` and `prev[1].next[1] = node` present changes to the already known pointers.

The function f_i is invoked with arguments `R0`,1,`v0` where `R0` and `v0` are set to symbolic by the user. `R0` points to a symbolic object `R0.next[1]` and a concrete object `R0.next[0]` (also `R1`). The invocations of f_i and f_{sn} start with evaluation of their decision trees and lead to actions that affect state as if functions are executed. Upon invocation of each function, the arguments are mapped to symbolic names used during unit testing. For f_i, the symbolic names `H`, `K`, `V` used in unit testing of f_i are mapped to real arguments `R0`, 1, `v0`. The evaluation of first decision tree node invokes f_{sn}. The arguments `H`, `K`, 1 and local `prev` are passed to f_{sn}. Before invocation, local memory object `prev` is allocated and used as output buffer for f_{sn}. Concolic unit testing names `h`, `k`, `i`,`p` are mapped to reals, concrete or symbolic, in the caller (i.e., `H=R0`, `K=1`, 1 and `prev`). The recursive call `search_node(h, k, i-1, p)` maps h_2, k_2, i_2, p_2 to `R0`, 1, `0`, `prev`.

In the first invocation of f_{sn}, the evaluation of branch conditions in the decision tree uses symbolic arguments (e.g., `h.next[i] != NULL` becomes `R0.next[1] != NULL`) that can be true or false, but we choose to explore the true branch first. After evaluating branch conditions, the appropriate data shape is processed to affect the current data structure. Memory object `R2` is created to satisfy the pointer-pointee relationship in the path. Finally, the memory operations are processed: the write `p[i] = h` is converted to `prev[1] = R0` and `prev[0] = R0` in the first and second invocations.

Table 1. Pointer-pointee relations and the visualized skip list shape generated along path `T F T` in `insert`.

line#	pointer	pointee
17	curr	N0
24	prev[0]	N1
25	node	N2
26	prev[0]->next[0]	N2
28	prev[1]	N3
29	prev[1]->next[1]	N2

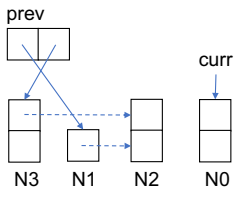

Table 2. Memory accesses along path `T F T` in `insert`.

line#	address	value	type
17	curr->key		Load
25	prev[0]->next[0]		Load
25	node->next[0]	prev[0]->next[0]	Store
26	prev[0]->next[0]	node	Store
28	prev[1]->next[1]		Load
28	node->next[1]	prev[1]->next[1]	Store
29	prev[1]->next[1]	node	Store

In the function f_i, the branch nodes are evaluated using the return value `curr` from f_{sn}. Since in the second invocation of f_{sn} returns `h.next[i]`, which is `R1`, the return value `curr` refers to `R1`. The branch conditions become `R1 != NULL` and `R1.key == k`. After calling of `create` and `rand_level`, the local variable `node` and `level` become concrete values. During the evaluation of the subsequent decision tree nodes in Figure 5, lock/unlock events are processed, and memory accesses that write to nodes `prev[0]` and `prev[1]` (which both refer to `R0`) are processed. After the invocation of f_i is complete, all the local variables and names disappear upon the pop action of the current stack frame.

Note that all of the above actions were performed using function summaries which optimizes the work performed.

3 Detailed Algorithms

Next we present some key details of our algorithms: summary invocation algorithm during full program concolic testing (Section 3.1); and summary guided search for races while exploring data shapes and thread interleavings (Section 3.2). We assume that the summary of each function is already available.

3.1 Summary Invocation Algorithm

During the concolic testing of the full program, a call to a summarized function f is replaced by invocation of its summary - which is the decision tree model $\delta(f)$. The invocation algorithm maps the symbolic and concrete values, including pointers, obtained via *concolic unit testing* to the values in the current state. The decision tree $\delta(f)$ of function f is used to determine which path is followed and the program state impacted by execution of the path is updated by storing symbolic addresses in memory. The invocation algorithm only handles the summarized functions. Unsummarized functions, or paths whose summaries are unavailable, are executed as they are by standard concolic testing.

The invocation process, presented in Algorithm 1, begins with an execution state ϕ which has been set up the list of input parameters (symbolic or concrete values) as local variables, and the root node n of the decision tree $\delta(f)$. The `InvokeSummary` function presents the actions for different node types. For all node types, at line 2, first $\phi \overset{S}{\underset{n}{\Longrightarrow}} \phi_s$ applies data shapes to the current state ϕ. Then for a branch node, we evaluate the branch condition $eval(B(n) = true, \phi)$ (line 4-6) and $eval(B(n) = false, \phi)$ (line 7-9) to decide whether branch condition is true, false, or either. Then, we apply the branch condition to the path constraints in the new state(s). We continue to process the child branches based on the evaluation results using new state(s) (at line 6 and 9). For call, synchronization, and return node types, we process memory accesses $\phi_s \overset{M}{\underset{n_p - n}{\Longrightarrow}}$ from the last non-branch node (line 11-12) since their side effects must be reflected in the

Algorithm 1: Summary Invocation

```
1  Procedure InvokeSummary(φ, n):
2      φ ⟹_n^S φ_s
3      if n is branch then
4          if eval B(n) can be true in φ then
5              φ_s ⟹_n^{B=true} φ_t
6              InvokeSummary(φ_t, n.true_branch)
7          if eval B(n) can be false in φ then
8              φ_s ⟹_n^{B=false} φ_f
9              InvokeSummary(φ_f, n.false_branch)
10     else
11         n_p ← the last non-branch node
12         φ_s ⟹_{n_p−n}^{M} φ_m ⟹_n^{C/L/V} φ' based on the type of n
13         if n is return then
14             Φ.push(φ')
15         else
16             InvokeSummary(φ', n.child_node)
17 Procedure mapObjectInit(args):
18     obj_map, value_map ← {}, {}
19     foreach unit_arg, real_arg in args do
20         if isPointer(real_arg) then
21             [b, o] ← getAddr(unit_arg)
22             obj_map[b] ← getAddr(real_arg)
23         else
24             value_map[unit_arg] ← real_arg
25     return obj_map, value_map
26 Procedure mapObject(obj_map, value_map, S):
27     foreach pointer, pointee in S do
28         real,_= convert(obj_map, value_map, pointer, _)
29         data = read(real)
30         [b, o] = getAddr(pointee)
31         obj_map[b] ← getAddr(data) - [0, o]
32         value_map[pointee] ← real
33 Procedure convert(obj_map, value_map, a, v):
34     [b, o] = getAddr(a)
35     [b', o'] = obj_map[b] + [0, exprReplace(value_map, o)]
36     return b' + o', exprReplace(value_map, v)
```

new state ϕ_m. The new state will serve as the start state for a new call, return, or a thread context switch. The state ϕ' represents the state after applying $C'_f(n)$, $L(n)$, or $V(n)$ based on the node type (line 12). Eventually, the final state ϕ' will be pushed into state queue Φ as the final result. To implement $\phi \overset{\delta(f)}{\underset{n}{\Longrightarrow}} \phi_s$, the key actions when invoking a summary are defined as follows:

(1) *Create object mapping.* Each memory object in the concolic *unit* test summary is mapped to real memory corresponding to the executing program. This mapping is used to convert the memory objects from concolic unit testing to the real memory objects in the current execution state.

(2) *Create value mapping.* The value mapping is used to convert symbolic values used in concolic unit testing to the values in the current execution state.

(3) *Converting objects.* As the decision tree path taken is identified, the memory accesses to the objects in unit testing are converted to real objects by looking up the *object mapping* and calculating the offset if the pointer is not pointing to the beginning of the objects. Observe that multiple pointers in unit testing can map to the same object. Symbolic values in memory are mapped into symbolic or concrete values in the current state using *value mapping*.

For describing the details, some utility functions need to be defined first: $[b, o] \leftarrow$ `getAddr(x)` computes the base address and offset of memory object `x`; `isPointer` determines if the current expression represents a pointer; and finally `read(`p`)` can dereference a pointer p and read its content.

In Algorithm 1, $\phi \overset{S}{\underset{n}{\Longrightarrow}} \phi_s$ can be implemented using two basic operations - `mapObject` and `mapObjectInit` functions. First, we initialize the object mapping and value mapping by calling `mapObjectInit` with the input parameter pairs, which are the parameters in current state and parameters from the unit testing. Each pair of parameters is processed and added to the object mapping and value mapping as initial information for which objects from unit testing and current state map to each other. Each time during processing $\phi \overset{S}{\underset{n}{\Longrightarrow}} \phi_s$, `mapObject` is called by passing the same object mapping, value mapping containers and the shape summary $S(n)$ which is a list of pointer-pointee relations (PPRs). First, the pointer is converted into the real memory address *real* and loaded with the value *data* that *real* points to. If *data* is concrete value, which points to a memory object, this object should be mapped to the object that pointee referenced to in the unit testing (line 28-31). For handling address being writen and stored in the memory access, the value mapping is updated (line 32).

The `convert` function converts a memory access with address a and value expression v (for write access) to a real memory object by looking up the object mapping and value mapping. The memory access $\phi \overset{M}{\underset{n}{\Longrightarrow}} \phi_m$ is implemented by converting the memory reads and writes in M from unit testing into real memory object reads and writes. Only the first read and last write for accesses to an address are needed.

3.2 Summary-Guided Race Detection

Next we provide a summary guided race detection algorithm based upon the lockset algorithm [54], hybrid race detection [44], and the Cloud9 thread scheduling algorithm [18]. Our race detection method has two phases. In the first phase, before program testing, we iterate over all the possible pairs of paths in a given summary to find potential data races in the function. We use the lockset algorithm [54] to expose potential data races from summaries for each pair of paths.

When traversing a pair of different memory accesses along two paths, if the accesses refer to the same address and do not hold the same lock, then we record a potential data race involving the accesses. During program testing, in the second phase, the thread scheduler postpones the thread when it reaches one of potential racing statements. This thread's reactivation is prevented until another thread reaches the corresponding racing statement confirming a race.

If two paths dereference pointers, the data race will only occur when both accesses refer to the same memory object. Therefore, integration of data structures for the two paths into a non-conflicting consistent shape is required for potential data race detection. Our integration algorithm accepts a pair of paths as the input and generates an integrated data shape for the paths as the output. The integration is achieved by modifying the pointer-pointee relations for the two paths. If we have two different paths of the same function, we can first start from the root pointer of all the parameters and local/global variables which has been marked as symbolic variables, and combining the pointer-pointee relationship for two paths. We illustrate this process using an example.

Next, we show the integration of data shapes generated for thread a path `T F T` and thread b path `T T T` of f_i. The line number (LN), collected path constraints (PCs), pointer-pointee relations (PPRs) of thread a, b, the corresponding data shape and their integrated result are shown in Figure 6, respectively. The integrated shape is created using the following steps: 1) Since both threads contain `prev` and `curr`, the pointer `prev[0]` is first integrated. However, a conflict of `prev[0]` pointing to `N1` in thread a and `N4` in thread b respectively is detected. The conflict is resolved by setting the pointer `prev[0]` is pointing to `N1` in thread b due to its pointee being modifiable. Observe there are multiple ways to resolve the conflict but we try to preserve the first shape as much as possible. All the pointers pointing to `N4` (`N5.next[0]` and `curr`) are modified to `N1`; 2) Then, the pointer `prev[1]` is transferred from `N5` to `N3` and the pointer `N5.next[0]` is also transferred so the PPR `N3.next[0]` pointing to `N1` is created; and 3) Because pointer `curr` in thread b points to `N1` is not modifiable, the `curr` pointer in integrated result uses the PPR `curr` pointing to `N1`.

LN	**PCs: Thread** a	LN	**PCs: Thread** b
25	curr ≠ NULL	25	curr ≠ NULL
25	curr->key ≠ k	25	curr->key = k
36	level = 1	26	curr = prev[1]->next[0]

	PPRs: Thread a			**PPRs: Thread** b	
LN	Pointer	Pointee	LN	Pointer	Pointee
25	curr	N0	25	curr	N4
33	prev[0]	N1	26	prev[1]	N5
34	node	N2	26	N5.next[0]	N4
35	N1.next[0]	N2	27	prev[0]	N4
37	prev[1]	N3			
38	N3.next[1]	N2			

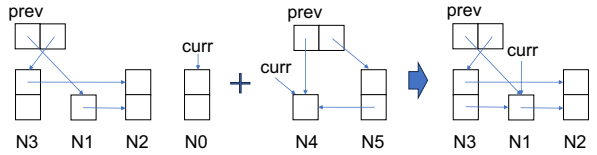

Figure 6. An example of data structure integration in thread a (left) and b (middle), with the integrated shape (right).

In the second phase of testing, the scheduler explores interleaving of threads to confirm a potential data race. A *postpone* operation is used to exercise thread interleavings. Once the current thread satisfies the conditions for a potential data race, the scheduler postpones the thread to allow another thread to be scheduled and progress to a point that realizes the data race. If a schedule is found that realizes the data race, the race is reported and thread is no longer postponed and allowed to be scheduled to search for another data race.

Algorithm 2: An algorithm for data race detection

Input: The initial state ϕ_0 and a set of potential race set R
Output: An optimized scheduling of summary invocations.

```
1  Procedure SchedulingGen(φ0, R):
2      Q = φ0
3      φ = Q.pop()
4      while Active(φ) ≠ ∅ do
5          for t ∈ Active(φ) do
6              while t ∉ postponed(φ) do
7                  t = nextThread(t, φ)
8                  if all threads are in postponed(φ) then
9                      postponed(φ) = postponed(φ) - t
10             φ' = forkThread(φ)
11             Current(φ') = t
12             runThreadUntilSync(φ')
13             Q.push(φ')
```

Algorithm 2 shows how the thread scheduler explores interleavings to confirm potential data races. It continues exploring all paths and thread interleavings combinations till testing time is exhausted. $Active(\phi)$ maintains a set of running threads. `postponed` is a set of threads which are currently postponed and cannot be scheduled yet. The algorithm checks when there are threads are running ($Active(\phi) \neq \emptyset$). For each running thread t, if t is in $Postponed(\phi)$, we postpone the thread to schedule the next one using the function `nextThread(t)` to get the next thread. If all threads are postponed, we remove the current thread from the $Postponed(\phi)$ set to make sure there is no dead lock. Then, we fork a new state to explore the thread scheduling for t and execute the thread t using `runThreadUntilSync` until it meets a thread synchronization event (for example lock, unlock, etc). Finally, we push state ϕ' in the state queue Q to schedule the next synchronization event.

Table 3. Data Race Detection Effectiveness of: AFL++ based Fuzzing; Cloud9 and SSRD based Concolic Execution.

Test Program	Function Name	# of Paths Covered			# of Data Races Detected		
		AFL++	Cloud9	SSRD	AFL++	Cloud9	SSRD
SL	insert	3	52	3779	2	2	8
	search	6	13	1479	0	0	2
ULL	insert	3	10	321	7	10	15
	delete	16	13	78	2	1	4
	search	16	16	93	0	0	3
PQ	insert	2	22	31	0	5	5
	remove	2	16	1479	0	11	19
AVL	insert	1	7942	16329	0	4	6
	delete	1	285	1322	0	1	12

Table 4. SSRD Execution Time: Without Using (WO-Sum) and With Using (W-Sum) Function Summaries + Summaries Construction Time. Negative percentages indicate reductions, and reduction in parentheses means the reduction if we do not consider summaries construction time. Analysis of Execution Time: Reduction of Constraint Solving Time Using (W-Sum) Function Summaries. # of Solved represents the number of constraints solved during execution.

Test Program	Function Name	Execution Time (s)			Constraints Solving Time		# Solved	
		WO-Sum	W-Sum (+cons. time)	Reduction (% w/o cons. time)	% of Total Time	W-Sum (% Reduction)	WO-Sum	W-Sum (% Reduction)
SL	insert	326.56	172.3 + 0.81	-47.0% (-47.2%)	45.27%	-72.69%	100,466	42,336 (-57.9%)
	search	116.19	98.29 + 0.75	-14.8% (-15.4%)	22.54%	-35.15%	45,948	28,728 (-37.5%)
ULL	insert	1092.27	87.84 + 0.39	-91.9% (-92.0%)	85.20%	-96.95%	485,212	46,655(-90.4%)
	delete	76.26	21.47 + 0.44	-71.3% (-71.8%)	64.55%	-94.38%	75,548	3,333 (-95.6%)
	search	268.32	66.31 + 0.53	-75.1% (-75.3%)	73.56%	-92.32%	164,220	7,140 (-95.7%)
PQ	insert	15.45	23.77 + 0.88	+59.5% (+53.9%)	95.64%	+55.72%	5,704	6,764 (+18.6%)
	remove	111.26	8.58 + 31.39	-64.1% (-92.3%)	87.20%	-92.27%	9,486	527 (-94.4%)
AVL	insert	41.47	18.42 + 0.76	-53.7% (-55.6%)	48.66%	-97.92%	19,625	333 (-98.3%)
	delete	212.06	122.61 + 0.98	-41.7% (-42.2%)	53.39%	-96.40%	49,887	3,856 (-92.3%)

Table 5. Summarization Benefits: The total number of memory Reads and Writes during execution in experiments; and % Reduction in memory Reads/Writes due to enabling the summarization approach.

Test Program	Function Name	# of Memory Access W-Sum	
		Read (Reduction)	Write (Reduction)
SL	insert	4,460,946 (-8.3%)	2,122,027 (-5.3%)
	search	2,606,982 (-1.9%)	1,204,355 (-1.4%)
ULL	insert	84,180 (-88.0%)	158,774 (-46.9%)
	delete	14,564 (-86.1%)	14,564 (-66.5%)
	search	38,201 (-81.1%)	38,201 (-52.9%)
PQ	insert	993 (-24.6%)	1,352 (-2.6%)
	remove	6,108 (-36.7%)	7,419 (-3.2%)
AVL	insert	28,934 (-48.0%)	34,985 (-74.4%)
	delete	1357135 (-2.5%)	703537 (-23.0%)

4 Experiments

4.1 Experimental Setup

To study the effectiveness and efficiency of SSRD, we first use a solution based on the popular general-purpose fuzzing engine AFL++ [24] combined with the Thread Sanitizer [58] to evaluate the complexity of the benchmarks. Then we compare SSRD with a representative lock-set based data race detector using Cloud9 (concolic execution+lockset) [18]. In our experiments, we tested and compared our SSRD system with Cloud9 using the same lock-based implementations of the following concurrent dynamic data structures: *Skip List(SL)*, *Unrolled Linked-List(ULL)*, *Priority Queue(PQ)*, and *AVL-Tree(AVL)*.

The codes are modified from open-source projects and augmented with statements to trigger read-write and write-write data races. When running AFL++ based and the Cloud9 data race detector, we provide additional code to create an initial shape for each kind of data structures, and further

testing of insertion, deletion, and searching is based on the initial shapes. For SSRD, we use a symbolic pointer as the handle for the automatically generated data shape for testing. To control the size of generated shapes, we limit the length of the pointer chain allowed. To exploit thread interleaving, we use POSIX thread model in Cloud9 and create a pair of threads using the pthread APIs to execute functions concurrently for testing.

4.2 Race Detection Effectiveness

We evaluated the effectiveness of our system by comparing it with the traditional Cloud9 system with a lockset-based race-detecting algorithm. Table 3 shows the path covered and the number of data races detected using each approach. The AFL++ results show that it cannot effectively find data races. From all known races, it detects only 2 out of 10 in SL, 9 out of 22 races in ULL, 0 out of 24 in PQ, and 0 out of 18 in AVL. We then note that Cloud9 can detect more data races than AFL++ but these are a subset of data races that are detected by SSRD - 2 out of 10 in SL, 11 out of 22 in ULL, 16 out of 24 in PQ, and 5 out of 18 in AVL. Note that there is no data race detected by Cloud9 but not by SSRD. The only race in ULL found via fuzzing but not in SSRD due to constraint solving problem. We observe that the number of paths explored by SSRD 1.4×(31 vs 22) to 113.7×(1479 vs 13) is greater than Cloud9. This greater exploration of search space by SSRD is responsible for uncovering many data races that are missed by Cloud9.

4.3 Exploration Efficiency

The summaries play an important role in improving the exploration efficiency of SSRD. Table 4 compares the performance difference with/without summarization. We observe that with summaries (W-Summ), the execution time is reduced by 26% to 94% across the benchmarks over without summaries (WO-Summ). The overall efficiency of SSRD is better than Cloud9. However, there are two exceptions - the insert action for priority-queue and the delete action for AVL-tree experience slow down when using summaries. The reason is the paths contained in summaries are not encountered during testing, and thus the cost incurred for generating summaries does not yield any benefits.

Table 4 also analyzes the reduction in constraints solving time and number of solved constraints due to summarization. Note that constraints solving time represents a significant portion of the total time from 2.54% to 95.64% across the benchmarks. We observe that with summaries constraints solving time is reduced by 35% to 97% across the benchmarks, while the number of constraints solved is reduced by 37.5% to 98.3%. The constraints solving time reflects the time spent on solving the branch condition and symbolic pointers. This is reduced using summaries since there are solved branch conditions and generated data shapes in the saved path summaries.

Table 5 analyzes the performance improvements of SSRD due to reductions in memory access. The number of read and write operations is reduced via summaries by 1.4% to 88.0% since there are temporary memory read and write operations that are performed during concolic unit testing that are not repeated during full testing. At the same time, since the memory access also requires solving address constraints for symbolic addresses, the reduction in memory access also reduces the number of constraints solved.

5 Related Work

Data race detection methods, static or dynamic, have been widely studied [12, 29, 37, 39, 44, 53, 55, 65]. Both static and dynamic methods have advantages and limitations.

Static data race detection methods. *Type-/Language-based* flow-insensitive race detection methods [8, 13, 15, 25, 28] are static data race detection methods incorporated in compilers. However, typically these methods require adding annotations, modification of programs, or rewriting the algorithm in another language. In contrast, our approach works directly on existing C/C++ concurrent programs. Though some methods can automatically annotate the program [3], it is hard to know what exception leads to a data race without runtime information. *Lockset* based flow-sensitive static approaches [12, 22, 29, 37, 39, 51, 60, 65] are another major kind of data race detection methods. These approaches are sound, i.e. the system finds all potential races. The primary limitation is high rates of false positives. RacerX [22] uses several heuristics to filter false warnings. RacerD [12] and RacerDX [29] are static methods with few false positives. Even then, they require the user to manually confirm the data races and ascertain race conditions as well as thread interleavings. Our approach aims to automatically generate input data and possible thread interleavings to detect the data races with its race conditions.

Dynamic data race detection methods. *Happens-before* analysis [1, 19, 26, 40] and *Lockset* analysis [2, 17, 21, 42, 54, 64] are two kinds of methods for dynamic data race detection. Happens-before analysis produces no false positives but is inefficient. Lockset analysis relies on collecting memory access information and lock/unlock event tracing for race detection. However, it can lead to false positives. Therefore, a combination of those two algorithms [20, 31, 44, 50, 71] can give a more efficient and accurate result. Some hardward-based approaches has been proposed. Kard [4] is a lightweight race detector using per-thread memory protection, Intel Memory Protection Keys. However, all these data race detection methods require other techniques to test multiple paths and thread interleavings to expose races.

Path explosion and thread interleaving explosion problems in multithreaded programs testing. *Fuzzing* [16, 35, 47, 55, 63, 67], *Symbolic/Concolic Execution* [11, 14,

18, 23, 38], and *ModelChecking* [43] are three primary testing methods for systematically exploring multiple paths and thread interleavings. Fuzzing is commonly used in test generation for large programs. However, it does not guarantee exploration of all paths and thread interleavings leaving data races hidden in a rare branch condition unexposed. Symbolic/Concolic Execution is the state-of-the-art technology to explore all paths and to generate the test input but also faces path explosion and thread interleaving explosion problems. Klee [14] is a symbolic execution engine for LLVM IR. Cloud9 [18] extends klee to POSIX environment and pthreads. State merging method [38] has been proposed to handle the path explosion problem during symbolic execution. Con2colic [23] explores the content of data structure and thread interleavings in multithreaded programs. However, it is based on random search rather than targeted search which limits scalability. Moreover, it cannot explore data shapes for pointer-based data structures. SymCC [48] and SymQemu [49] use compilation-based rather than interpreter-based technique which greatly improves performance, but do not reduce computation complexity.

Summarization is a well known method for dealing with path explosion. [27] proposes compositional automatic test generation that can scale to large programs with many feasible paths. [5] extends it by minimizing intraprocedural paths symbolically executed when forming an interprocedural path. These methods can improve symbolic execution in a compositional way but they do not contribute to data shape exploration. Incremental symbolic execution has been studied in [30, 33, 46, 66, 68, 69], and using summarization is one of the main approaches. *Interpolation-based function summaries* [59] reuse summaries during incremental verification. *Efficient summary reuse* [32] is another approach for regression verification for reusing intermediate results from the previous verification runs. FENSE [69] summarizes previously explored paths by recording the variables that may induce different incremental behaviors. However, those summaries can only reuse results between different versions and are not suitable for a single run during exploration of data shapes. *State merging* [38] solves path explosion problem by combining similar states during execution. *Path Subsumption* [70] uses an annotation algorithm for branches and statements labels, which are implied by the current state. *Chopped symbolic execution* [62] can jump over unrelated functions during symbolic execution, which can reduce the chance of forking new states. The above methods only focus on the path explosion problem but do not deal with data shape exploration and data race detection efficiency. Our approach uses summarization to help quickly explore data shapes and guide the thread scheduler and speed up race detection in exploring thread interleavings.

Model Checking is the primary method to explore thread interleavings. However, its cost is too high to cover all interleaving cases. SPL language [57] provides an extended to simplified form of sequential Java to support multithreaded program model checking using the counterexample-guided abstraction-refinement framework. *Bounded Model Checking* via lazy sequentialization [34] proposes a systematic way to check sequentially consistent C programs using POSIX threads. However, model checking requires significant effort to modify current multithreaded program and rarely supports concurrent data structures. In contrast, our approach is based on symbolic execution and requires minimal code changes. Thus, no method handles path and thread interleaving explosion for exposing data races in concurrent data structures.

Related methods for checking concurrent data structures. Handling concurrent data structures is one of the most difficult tasks in dynamic data race detection due to infinite states and different shape requirements for different paths. CDSChecker [43] provides a model-checking algorithm for modeling concurrent code under the C++ memory model. However, CDSChecker is not aimed for lock-based synchronization, and both path explosion problem and thread interleaving explosion problem is not solved by it. CDSSPEC [45] is a specification checker for the C++11 memory model. However, it requires use of a specification language to describe the data structure and still has high overhead. Shoal [6] is a system that extended SharC, by grouping objects and providing sharing rules on each group. It can avoid data races for concurrent data structures by turning data race detection problem into a sharing-rule violation detection problem. DSGEN [61] uses a data shape generation method for detecting data races in concurrent data structures of the CUDA platform. However, as the memory model and synchronization methods are different for a multicore system, it cannot be used. [41] uses a template approach in the Verified Software Toolchain [7] to prove the correctness of concurrent search structure. However, this approach requires manually written templates to verify the program which is not an automatic solution for generating test cases to detect the faults.

6 Conclusions

We presented an new approach for concolic testing of multithreaded programs that employ concurrent dynamic data structures to uncover data races. The key contributions of this work include summary computation and exploitation for efficiency, and non-conflicting shape determination and thread scheduling to guide concolic testing for exposing data races. Our evaluation shows that our approach is significantly more effective in uncovering data races than Cloud9 and reuse of summaries leads lightweight creation of objects and removal of elimination of memory accesses.

Acknowledgments

This work is supported in part by National Science Foundations grants CCF-2226448, CCF-2002554, and CCF-2028714 to University of California Riverside.

References

[1] Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H. B. Netzer. 1991. Detecting Data Races on Weak Memory Systems. *SIGARCH Comput. Archit. News* 19, 3 (apr 1991), 234–243. https://doi.org/10.1145/115953.115976

[2] Rahul Agarwal, Amit Sasturkar, Liqiang Wang, and Scott D Stoller. 2005. Optimized run-time race detection and atomicity checking using partial discovered types. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. 233–242.

[3] Rahul Agarwal and Scott D Stoller. 2004. Type inference for parameterized race-free Java. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 149–160.

[4] Adil Ahmad, Sangho Lee, Pedro Fonseca, and Byoungyoung Lee. 2021. Kard: Lightweight data race detection with per-thread memory protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 647–660.

[5] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. 2008. Demand-driven compositional symbolic execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 367–381.

[6] Zachary R Anderson, David Gay, and Mayur Naik. 2009. Lightweight annotations for controlling sharing in concurrent data structures. *ACM Sigplan Notices* 44, 6 (2009), 98–109.

[7] Andrew W Appel. 2011. Verified Software Toolchain: (Invited Talk). In *European Symposium on Programming*. Springer, 1–17.

[8] David F Bacon, Robert E Strom, and Ashis Tarafdar. 2000. Guava: A dialect of Java without data races. *ACM SIGPLAN Notices* 35, 10 (2000), 382–400.

[9] Phil Bagwell. 2001. *Ideal hash trees. EPFL*. Technical Report.

[10] Philip Bagwell and Tiark Rompf. 2011. *RRB-Trees: Efficient Immutable Vectors. EPFL*. Technical Report.

[11] Tom Bergan, Dan Grossman, and Luis Ceze. 2014. Symbolic execution of multithreaded programs from arbitrary program contexts. *ACM SIGPLAN Notices* 49, 10 (2014), 491–506.

[12] Sam Blackshear, Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. 2018. RacerD: compositional static race detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–28.

[13] Chandrasekhar Boyapati and Martin Rinard. 2001. A parameterized type system for race-free Java programs. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 56–69.

[14] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*. 209–224.

[15] Elias Castegren and Tobias Wrigstad. 2016. Reference Capabilities for Concurrency Control. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 5:1–5:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.5

[16] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. 2020. MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *29th USENIX Security Symposium (USENIX Security 20)*. 2325–2342.

[17] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. 2002. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. 258–269.

[18] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. 2010. Cloud9: A software testing service. *ACM SIGOPS Operating Systems Review* 43, 4 (2010), 5–10.

[19] Anne Dinning and Edith Schonberg. 1991. Detecting access anomalies in programs with critical sections. In *Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*. 85–96.

[20] Anne Dinning and Edith Schonberg. 1991. Detecting access anomalies in programs with critical sections. In *PADD '91*.

[21] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2007. Goldilocks: a race and transaction-aware java runtime. *ACM SIGPLAN Notices* 42, 6 (2007), 245–255.

[22] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, static detection of race conditions and deadlocks. *ACM SIGOPS operating systems review* 37, 5 (2003), 237–252.

[23] Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. 2013. Con2colic testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 37–47.

[24] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

[25] Cormac Flanagan and Stephen N Freund. 2001. Detecting race conditions in large programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 90–96.

[26] Cormac Flanagan and Stephen N Freund. 2009. FastTrack: efficient and precise dynamic race detection. *ACM Sigplan Notices* 44, 6 (2009), 121–133.

[27] Patrice Godefroid. 2007. Compositional dynamic test generation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 47–54.

[28] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and Reference Immutability for Safe Parallelism. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 21–40. https://doi.org/10.1145/2384616.2384619

[29] Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2019. A True Positives Theorem for a Static Race Detector. 3, POPL, Article 57 (jan 2019), 29 pages. https://doi.org/10.1145/3290370

[30] Shengjian Guo, Markus Kusano, and Chao Wang. 2016. Conc-iSE: Incremental symbolic execution of concurrent software. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 531–542.

[31] Jerry J. Harrow. 2000. Runtime Checking of Multithreaded Applications with Visual Threads. In *SPIN Model Checking and Software Verification*, Klaus Havelund, John Penix, and Willem Visser (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 331–342.

[32] Fei He, Qianshan Yu, and Liming Cai. 2020. Efficient summary reuse for software regression verification. *IEEE Transactions on Software Engineering* 48, 4 (2020), 1417–1431.

[33] Joran J Honig. 2020. *Incremental symbolic execution*. Master's thesis. University of Twente.

[34] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2014. Bounded model checking of multi-threaded C programs via lazy sequentialization. In *International Conference on Computer Aided Verification*. Springer, 585–602.

[35] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.

[36] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. 2022. Context-sensitive and directional concurrency fuzzing for data-race detection. In *Network and Distributed Systems Security (NDSS) Symposium 2022*.

[37] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. 2007. Fast and accurate static data-race detection for concurrent programs. In *International Conference on Computer Aided Verification*. Springer, 226–239.

[38] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient state merging in symbolic execution. *Acm Sigplan Notices* 47, 6 (2012), 193–204.

[39] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 308–319.

[40] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. 2007. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. *SIGPLAN Not.* 42, 6 (jun 2007), 22–31. https://doi.org/10.1145/1273442.1250738

[41] Duc-Than Nguyen, Lennart Beringer, William Mansky, and Shengyi Wang. 2024. Compositional Verification of Concurrent C Programs with Search Structure Templates. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 60–74.

[42] Hiroyasu Nishiyama. 2004. Detecting Data Races Using Dynamic Escape Analysis Based on Read Barrier.. In *Virtual Machine Research and Technology Symposium*. 127–138.

[43] Brian Norris and Brian Demsky. 2013. CDSchecker: Checking Concurrent Data Structures Written with C/C++ Atomics. *SIGPLAN Not.* 48, 10 (oct 2013), 131–150. https://doi.org/10.1145/2544173.2509514

[44] Robert O'callahan and Jong-Deok Choi. 2003. Hybrid dynamic data race detection. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*. 167–178.

[45] Peizhao Ou and Brian Demsky. 2017. Checking concurrent data structures under the C/C++ 11 memory model. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 45–59.

[46] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. 2011. Directed incremental symbolic execution. *Acm Sigplan Notices* 46, 6 (2011), 504–515.

[47] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. 2155–2168.

[48] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with {SymCC}: Don't interpret, compile!. In *29th USENIX Security Symposium (USENIX Security 20)*. 181–198.

[49] Sebastian Poeplau and Aurélien Francillon. 2021. SymQEMU: Compilation-based symbolic execution for binaries. In *NDSS 2021, Network and Distributed System Security Symposium*. Internet Society.

[50] Eli Pozniansky and Assaf Schuster. 2007. MultiRace: efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience* 19, 3 (2007), 327–340.

[51] Polyvios Pratikakis, Jeffrey S Foster, and Michael Hicks. 2011. LOCKSMITH: Practical static race detection for C. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 1 (2011), 1–55.

[52] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990), 668–676.

[53] Jake Roemer, Kaan Genç, and Michael D Bond. 2020. SmartTrack: efficient predictive race detection. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 747–762.

[54] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 391–411.

[55] Koushik Sen. 2008. Race directed random testing of concurrent programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 11–21.

[56] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '05)*. 263–272.

[57] Koushik Sen and Mahesh Viswanathan. 2006. Model checking multithreaded programs with asynchronous atomic methods. In *International Conference on Computer Aided Verification*. Springer, 300–314.

[58] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*. 62–71.

[59] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. 2012. Incremental upgrade checking by means of interpolation-based function summaries. In *2012 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 114–121.

[60] Nicholas Sterling. 1993. {WARLOCK}-A Static Data Race Analysis Tool. In {*USENIX*} *Winter 1993 Conference* ({*USENIX*} *Winter 1993 Conference)*.

[61] Xiaofan Sun and Rajiv Gupta. 2021. DSGEN: Concolic Testing GPU Implementations of Concurrent Dynamic Data Structures. In *Proceedings of the ACM International Conference on Supercomputing* (Virtual Event, USA) *(ICS '21)*. Association for Computing Machinery, New York, NY, USA, 75–87. https://doi.org/10.1145/3447818.3460962

[62] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering*. 350–360.

[63] Nischai Vinesh and M Sethumadhavan. 2020. Confuzz—a concurrency fuzzer. In *First International Conference on Sustainable Technologies for Computational Intelligence*. Springer, 667–691.

[64] Christoph Von Praun and Thomas R Gross. 2001. Object race detection. *Acm Sigplan Notices* 36, 11 (2001), 70–82.

[65] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 205–214.

[66] Rong Wang, Shaoying Liu, and Yuji Sato. 2021. SIT-SE: a specification-based incremental testing method with symbolic execution. *IEEE Transactions on Reliability* 70, 3 (2021), 1053–1070.

[67] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1643–1660.

[68] Guowei Yang, Suzette Person, Neha Rungta, and Sarfraz Khurshid. 2014. Directed incremental symbolic execution. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 1 (2014), 1–42.

[69] Qiuping Yi and Guowei Yang. 2022. Feedback-driven incremental symbolic execution. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 505–516.

[70] Qiuping Yi, Zijiang Yang, Shengjian Guo, Chao Wang, Jian Liu, and Chen Zhao. 2015. Postconditioned symbolic execution. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–10.

[71] Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. Racetrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the twentieth ACM symposium on Operating systems principles*. 221–234.

Received 2024-03-22; accepted 2024-05-10