EFFICIENT AND SCALABLE WHOLE PROGRAM RACE DETECTION FOR JAVA AND ANDROID PROGRAMS

A Thesis

by

YANZE LI

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Shaoming Huang
Committee Members,	Jennifer Welch
	I-Hong Hou
Head of Department,	Scott Schaefer

August 2020

Major Subject: Computer Science

ABSTRACT

Races are common in all concurrency systems, including multithreaded programs, event-driven programs, distributed systems, etc. Therefore it is crucial to have tools to reveal potential races in a software system. Existing static race detection tools either cannot scale well on large programs or report too many false positives due to not reasoning happens-before relations. On the other hand, dynamic race detection tools are bound by test inputs. Thus they only have low code coverage.

This research presents an efficient race detection framework design along with two implementations, SWORD and SDROID, that can efficiently detect races on real-world Java programs and Android apps. The design leverages the state-of-the-art context-sensitive pointer analysis and uses a concept called "origin" to efficiently compute the alias information between concurrent entities (e.g., threads, events, etc.). It detects races based on a flow-sensitive lockset algorithm and a highly optimized Static Happens-Before (SHB) graph. To further support race detection on Android apps, we create an abstract thread model to enable reasoning about the behavior of Android apps and their underlying Android Runtime System. We then extend our race detection framework based on the abstract model.

Our evaluation compares SWORD with two state-of-the-art static race detectors. The results indicate SWORD achieves a 10x speedup over previous work and has the highest precision on whole program race detection for Java programs. We also use SDROID to successfully expose some previously unknown bugs in some popular Android apps.

ACKNOWLEDGMENTS

There are a number of people without whom this thesis might not have been written, and to whom I am greatly indebted.

I would like to especially thank my adviser, Jeff Huang, for bringing me into the realm of programming language and software engineering. He provided countless insightful advises and taught me how to do impactful researches.

Thanks to my other committee members, Jennifer Welch and I-Hong Hou, for contributing a great deal to my development as a graduate by giving me the benefit of their times and advises.

Thanks to my colleagues Peiming, Brad, and Bozhen for supporting me and discussing interesting research ideas with me.

Special thanks to my colleague Gang, for drinking beer with me and listening to me complaining when I was upset.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professor Jeff Huang and Jennifer Welch of the Department of Computer Science and Engineering and Professor I-Hong Hou of the Department of Electrical and Computer Engineering.

All the work conducted for the thesis was completed by the student independently.

Funding Sources

The graduate study was supported by an assistantship from Texas A&M University.

NOMENCLATURE

ART	Android Runtime
CFG	Control-flow Graph
HB Relation	Happens-Before Relation
IR	Intermediate Representation
LSP	Language Server Protocol
OSA	Origin Sharing Analysis
PTA	Points-to Analysis
PTS	Points-to Set(s)
PAG	Pointer Assignment Graph
SHB Graph	Static Happens-Before Graph

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

1.1 Data Race Detection in a Nutshell

The sharp increase in the number of cores in hardware brings up the necessity of writing parallel programs. Concurrency bugs, especially data races, are notoriously challenging to find, verify, and fix in real life, due to the complexity of concurrency and their natural non-determinism. To mitigate those issues, developers rely on data race detectors to help them find potential races.

A data race is defined as two instructions accessing the same shared memory concurrently without any synchronizations. Existing techniques to detect data races broadly fit into three categories, static analysis, dynamic analysis, and hybrid analysis.

1.1.1 Dynamic Analysis

Dynamic analysis requires program executions during the analysis process. Those tools first instrument the program (either on source code, IR, or binaries), and collect runtime information by executing the programs. They then use either an offline or online algorithm to detect races from the observed traces or even predict potential races based on current race-free traces. Previous tools such as fasttrack [2] and eraser [3], try to log shared memory accesses and lock acquisitions, and then use the *vector clock* algorithm [4] to detect races.

Dynamic tools are usually complete, meaning they may miss existing races in the programs, but all the races reported are real. The limitation of dynamic analysis, however, lies in 3 major aspects:

1. It brings significant overhead to the target programs. In most case, the slowdown is between 2x to 50x;

2. It has low code coverage. If a dynamic tool wants to discover most of the races that exist, it requires a set of fine-grained inputs and multiple rounds of execution.

3. It can only be integrated into the testing phase or the production phase. Empirical studies have shown that the later phase a bug is detected, the more expensive to get it fixed.

Consequently, dynamic analysis performs well on small or middle-sized programs with simple inputs, but it is difficult to scale to larger programs or be adopted by industry.

1.1.2 Static Analysis

Static analysis, on the other hand, does not require running the programs to detect races. Static tools usually take an Intermediate Representation (IR) of a program as input and abstract the program execution. For example, the program execution can be modeled as a directed graph, or we can create an abstract machine that interprets the target program. As a result, a program analysis problem is transformed into a graph reachability problem or state-machine transitions.

Specifically, static data race detection reasons about three properties for each pair of memory accesses within the target program:

1. If the variables currently being accessed point (refer) to a shared memory location.
2. If the two memory accesses can happen in parallel
3. If the two memory accesses are protected by a shared lock.

Previous works such as RacerX [5], Chord [6], LOCKSMITH [7] use context-sensitive and/or flow-sensitive dataflow analysis to reason about locks, while not precisely reasoning about happens-before relations. Therefore, their analysis results contain a big portion of false positives, and the expensive dataflow analysis they use leads to scalability issues in large programs.

RacerD [8] is a race detector based on separation logic [9] and can analyze programs in a compositional way, i.e., only analyzing the newly introduced code changes. However, in order to be scalable, RacerD does not rely on pointer analysis to identify shared variables and locks. Instead, it is based on syntactic pattern [1] and a simple ownership analysis to determine shared variables and assumes all lock operations are well-formed. Other than that, RacerD does not reason about happens-before relations. It uses the lock information and program annotations to infer if two functions may happen in parallel. Thus, RacerD is neither sound nor complete, but it is proved to be practical in industries due to its scalable design.

ECHO [10] is an IDE-based incremental data race detector that first introduces a static happens-before (SHB) graph to reason about happens-before (HB) relations. Upon a change in the source code (addition, deletion, or modification), ECHO only analyzes the change introduced instead of re-analyzing the whole program. For about 92% of the code changes in small/middle-sized programs, ECHO takes no more than 0.1s to detect races [10].

Based on ECHO, D4 [1] first introduces an incremental pointer analysis. It optimizes the race detection by making a more efficient SHB Graph structure and designing parallel algorithms for pointer analysis updates and race detection.

However, both D4 and ECHO are specially designed for incremental analysis. When analyzing the whole program, they can take a few hours to finish. Therefore, it is still not practical to apply ECHO and D4 as whole program data race detectors.

1.1.3 Hybrid Analysis

Hybrid analysis is a kind of analysis that combines both static and dynamic techniques.

RaceFuzzer [11] is a tool that uses a lightweight static analysis to obtain a set of po-

[1] For example, if two memory operations are both accessing `obj.f`, although the variable `obj` may refer to different heap objects, RacerD considers them as aliases due to the identical syntactic pattern.

tential data race locations. Then it fuzzes the program using a set of predefined inputs. During the fuzzing process, it controls the thread scheduling in order to reveal different thread interleavings.

Generally speaking, hybrid analysis can achieve the best precision, but it also suffers from both disadvantages of static and dynamic analysis. In order to get a smaller size of potential race locations, it requires sophisticated static analysis. To effectively expose races during the fuzzing process, it also requires a set of fine-grained inputs.

1.2 Event-Driven System

The Event-driven paradigm allows programmers to design flexible systems, and it has become popular in a range of domains. However, the underlying event processing introduces extra complexities for program analysis. Not only because event-driven systems may suffer from events being damaged, delayed, or lost, but more importantly, the events are handled in a non-deterministic way.

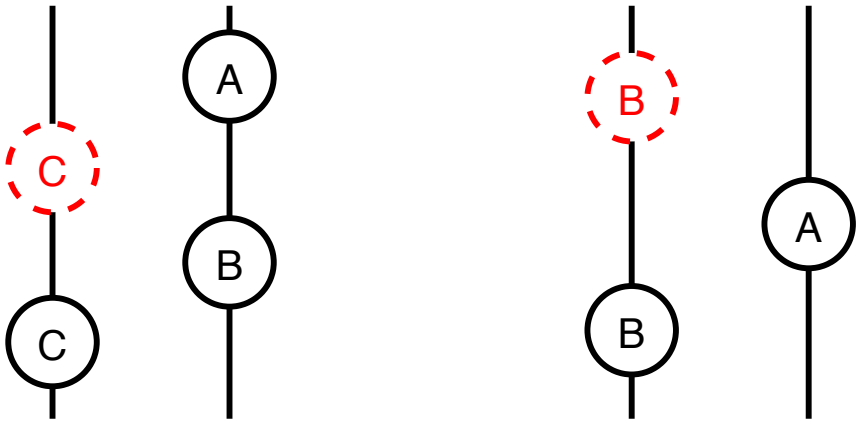
On the other hand, event-driven systems and thread-based systems share some intrinsic similarities. Back in the late 70s, Lauer and Needham compared event-driven systems with thread-based systems [12] and concluded that events and threads were intrinsically dual to each other. Later on, people argued about using threads vs. events from different perspectives [13, 14, 15].

Regardless of the pros and cons between threads and events, the connections between the two concepts are obvious – they are both an encapsulation of a unit of operations, and they both have non-determinism. Therefore, it is intuitive for a static analyzer to extend its multithreaded model to support modeling events, so that it covers more types of concurrency bugs.

1.2.1 Race Conditions

The event-driven systems are usually built around a single-threaded "event loop", that collects incoming events and dispatches them to the corresponding event handlers. The execution of events cannot overlap with each other, but the order of event sequence and the event dispatching is non-deterministic.

Therefore, in a single-threaded event-driven system, there's no "data race" we discussed above, but only "race conditions" between events. In general, we can summarize race conditions into two types:



(a) Atomicity Violation (b) Order Violation

Figure 1.1: 2 types of race conditions

1. **Atomicity Violation:** As shown in Figure 1.1a, assuming the event A, B, and C all access the same memory location, where event C unexpectedly happens between the event A and B, thus violating the atomicity between the event A and B.
2. **Order Violation:** As shown in Figure 1.1b, assuming the event A and B both access the same shared memory, and the event B mistakenly happens before event A, thus

violate the expected order between the two events. For example, reading a variable (event B) before it is initialized (event A).

To analyze an event-driven system, we first need to infer the atomicity constraints (the A-B in Figure 1.1a) and the order constraints (the A-B in Figure 1.1b) to detect potential violations. On the other hand, in a concurrent event-driven system, except for the normal data races between threads, there will also be data races between threads and event handlers.

1.2.2 Android System

Android is the most prevalent platform for smartphones and tablets. According to an empirical study from 2008 to 2013 [16], concurrency is within the top 5 causes of bugs.

Android is a concurrent event-driven system centered around the main thread (i.e., the UI thread) that manages UI responses. Other threads are mostly used for time-consuming tasks such as I/O requests or heavy computations. The UI thread runs an event loop. Event handlers are registered by developers in response to events such as user interactions or system notifications.

Previous works that use dynamic analysis to detect races in Android [17, 18, 19] require special devices to collect traces from the Android system, and may take several hours to run the offline analysis. On the other hand, they still share the input-dependent and low-coverage issues of dynamic analysis.

Existing static analysis tools [20, 21, 22, 23] either only target at generic event-driven systems, without leveraging the innate relations between Android event handler, or use expensive techniques, such as symbolic execution, to reason about races. Therefore those techniques can not scale well on large Android apps.

1.3 Pointer Analysis

Pointer analysis (i.e., points-to analysis, PTA), is a technique that computes the value of pointer variables statically, i.e., what memory addresses or heap objects can a pointer point to or a variable refer to at runtime? Virtually all interesting static program analyses rely on PTA, e.g., race detection, model checking, program optimization, security analysis, etc.

Since precise PTA is theoretically undecidable [24], certain abstractions must be made to achieve a practical PTA. Therefore, the precision of a PTA can be viewed from multiple dimensions:

Flow-sensitivity concerns if the PTA takes the program execution order into account. For a flow-insensitive PTA, a pointer can point to any heap objects assigned to it within the whole program scope, even if some assignments happen after the pointer dereference. While in a flow-sensitive PTA, the PTS at different pointer dereferences are different based on the possible values that can flow to the current location.

Context-sensitivity concerns if the PTA distinguishes different contexts of function calls. For a context-insensitive PTA, the pointers within a function call end up pointing to all arguments ever passed in from the function invocations. Since it is impossible to know how many function calls are actually made during the runtime, the identifier for a new context also varies. For example, we can distinguish contexts from the function call-site (the line where a function is invoked, call-site-sensitive), or the receiver object of the function (object-sensitive).

Field-sensitivity concerns if the PTA distinguishes different fields of a struct or class instance. This significantly influences the PTA precision for object-oriented programming languages.

Other than the dimensions listed above, PTA still varies from other perspectives, such

as the heap modeling, the branch conditions, and handling array indices. Previous researches have explored all aspects of pointer analysis [25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38], such as efficient algorithms to solve points-to constraints, different modellings of heap objects, and more effective choices of contexts, etc.

However, most previous works only discuss PTA from a generic perspective instead of a specific application. For data race detection, we find that a flow-sensitive PTA is usually infeasible for practical use. At the same time, a field-sensitive PTA is always needed in order to distinguish object fields. The context-sensitivity, however, not only affects the scalability of the PTA but also has a huge impact on analysis precision. Previous researches rarely chose contexts based on the semantics of the higher-level applications; therefore, in this thesis, we proposed am origin-sensitive PTA, which is both precise and scalable in terms of race detection.

1.4 IDE Integration and Debugging Information

Empirical studies have shown that a bug is more like to be fixed if it can be detected at an earlier phase [39]. Therefore, if we can provide programmers with sufficient debugging information within the IDE, along with their fresh memories of the current code changes, a race should be easier to understand and fix.

Among the tools we mentioned above, ECHO is the only one with IDE integration. The IDE integration of ECHO has two major limitations. First, it does not provide stack trace information. Therefore it is unclear to programmers how a race could be triggered in real execution. Second, the whole analysis runs inside the IDE's UI thread, causing a significant slow down to the IDE.

In the thesis, we propose an efficient stack trace collection algorithm along with some heuristics to shrink the size of race reports to make the bug report more readable to programmers. Then we leverage the Language Server Protocol (LSP) [40] to build a clien-

t/server mode IDE plugin. This plugin has been integrated into Eclipse, IntelliJ IDEA.

1.5 Limitations

Based on the previous researches introduced above, we summarize the limitations of current whole-program race detectors for both Java and Android:

1. Existing PTA is not fully optimized for multithreaded/event-driven programs
2. Existing tools either do not reason about HB relations or they cannot scale well on whole-program analyses for large programs
3. Existing Android race detectors cannot scale to large Android apps.
4. Existing tools do not provide stack trace information in their report.
5. The IDE Integration for race detection tools is poorly supported.

1.6 Contributions

This thesis has addressed the previous limitations for whole-program race detectors, by introducing:

1. A new context called *origin*, that unifies threads and events to support efficient PTA in both multithreaded and event-driven systems.
2. A highly optimized SHB Graph design and race detection algorithm that scales well for whole program race detection on large programs.
3. An extended multithread/event-driven model for detecting races in Android apps.
4. An efficient algorithm for collecting stack traces for the races detected.
5. An IDE integration for the race detector leveraging LSP.

Besides, the first part of this thesis (SWORD) has been accepted and presented on ICSE' 19 [41], and the second part of this thesis (SDROID) is included in a current submission to OSDI' 20. The SHB Graph design and the race detection algorithm described

in this Chapter 2 has been extended to a LLVM-based race detector for OpenMP programs, which is under a current submission to SC' 20.

1.7 Outline of Thesis

In Chapter 2, I formalize the problem of detecting data races and discuss the existing techniques in ECHO and D4, based on which I introduce origin-sensitive PTA, optimized SHB Graph, and synchronization-region-based race detection algorithm. Besides the optimizations, I also show how to extend the SHB Graph to support efficient collection of stack trace information.

In Chapter 3, I introduce the Android system, especially its event-driven part, and extend the definition of origin-sensitive PTA and the race detection algorithm to support efficient race detection in Android apps.

In Chapter 4, I first introduce how SWORD and SDROID is implemented. Then I evaluate both SWORD and SDROID against the start-of-the-arts from precision and performance. The results show SWORD can efficiently analyze real-word programs, and its happens-before reasoning can significantly eliminate false positives. SDROID successfully detects real races in Firefox that involve both threads and events. Additionally, I also present the race report and IDE integration of SWORD, some discussions are made regarding the LSP and the engineering difficulty of developing such plugins.

Chapter 5 summarizes the thesis, recaps the contributions and lists some potential future work.

In this chapter, I formally introduce my work SWORD, a scalable whole-program race detector for Java.

2.1 Target Language

In this section, we first introduce a simplified programming language ,SIMJava [10], which is a subset of Java. This language serves as the target language for this paper. The feature within the language will be handled soundly (i.e., we are able to detect all data races within SIMJava), while other features (features in Java but not in SIMJava) will either not be handled or introduce unsoundness.

The syntax of the language is defined as below:

$$
\begin{array}{rll}
\langle \textit{program} \rangle ::= & \langle \textit{definition} \rangle\ \langle \textit{expr} \rangle & \\
\langle \textit{definition} \rangle ::= & \texttt{class}\ C\ \langle \textit{body} \rangle & \\
\langle \textit{body} \rangle ::= & \texttt{extends}\ \langle \textit{cls} \rangle\ \{ \langle \textit{field} \rangle^*\ \langle \textit{method} \rangle^* \} & \\
\langle \textit{field} \rangle ::= & \langle \textit{type} \rangle\ f & \\
\langle \textit{method} \rangle ::= & \langle \textit{type} \rangle\ mn(args^*)\ \{\ e^*\ \text{return}\ z\ \} & \\
\langle \textit{type} \rangle ::= & \langle \textit{cls} \rangle \mid \texttt{int} \mid \texttt{long} & \\
\langle \textit{cls} \rangle ::= & C \mid \texttt{Object} & \\
\langle \textit{expr} \rangle ::= & x = \texttt{new}\ \langle \textit{cls} \rangle & \textbf{(allocation)} \\
\mid & x = y & \textbf{(simple assignment)} \\
\mid & x = y.f & \textbf{(field read)} \\
\mid & x.f = f & \textbf{(field write)} \\
\mid & x = o.m(arg^*) & \textbf{(method call)} \\
\mid & x = y[i] & \textbf{(array read)} \\
\mid & x[i] = y & \textbf{(array write)} \\
\mid & t.start() & \textbf{(thread fork)} \\
\mid & t.join() & \textbf{(thread join)} \\
\mid & \texttt{synchronized}\ (x)\{\langle \textit{expr} \rangle^*\} & \textbf{(lock)} \\
\mid & loop(\langle \textit{expr} \rangle)\ \{\langle \textit{expr} \rangle^*\} & \textbf{(loop)} \\
\mid & if(\langle \textit{expr} \rangle)\ \{\langle \textit{expr} \rangle^*\} & \text{(conditional branch)} \\
\mid & \langle \textit{expr} \rangle^* &
\end{array}
$$

The important statements are highlighted above. As we see, other than assignments and memory reads/writes, SIMJava supports three fundamental synchronizations, i.e. *fork*, *join*, and *synchronized*. The *fork* and *join* introduce inter-thread HB relations, while *synchronized* introduces mutual exclusions. The *loop* expression represent both for-loop and while-loop, and we need to correctly handle them because it can cause multiple thread creations. The conditional branches are ignored during our analysis (therefore it's not highlighted), because supporting path-sensitive analyses requires symbolic reasoning for condition variable, which will significantly undermine the performance. On the other hand, ignoring conditional branch can still guarantee the soundness of our analysis, because we consider the conditional branch will always be executed, therefore no race will be missed.

2.2 Pointer Analysis

Pointer analysis (also known as points-to analysis) is a static program analysis technique that determines the value of pointer variable or expressions. Since heap is the primary structure for global program data, PTA thus acts as the foundation for most of the interprocedure program analysis. The input of a PTA is a sequence of program instructions, the output of the PTA is a points-to set (PTS) for each pointer variable and a call graph for the whole program.

2.2.1 The Necessity of Pointer Analysis in Race Detection

There are two fundamental problems for static race detection to start with:

1. Identifying the method being called at each method invocation site.
2. Identifying shared memory that might be accessed concurrently by multiple threads

For the first problem, we need a complete call graph for the program we analyze. More specifically, for a method call `x.m()`, we can only figure which methods get called by first

knowing which objects x may points to. Then we can query the class hierarchy to resolve the concrete method body. For the second problem, we need to know the PTS of each variable being accessed to see if it contains any thread sharing objects.

Altogether, we can see PTA is the underlying analysis that supports the whole race detection process. Without PTA, race detectors have to leverage additional information such as optional annotation or common program patterns to infer pointer values [8], thus making the analysis unsound.

2.2.2 Context-Sensitive Pointer Analysis

In SWORD, we use a context-, field-sensitive, flow-insensitive pointer analysis. As discussed in Chapter 1, we need field-sensitivity to accurately analyze a object-oriented programming language like Java, and our PTA is flow-insensitive due to the scalability issue. Therefore, the design choice lies in how to properly choose contexts.

We first introduce the following example to show how context-sensitive may significantly help data race detection:

```
1  void main() {
2      Thread t1 = new Thread(()->{
3          process();
4      });
5      Thread t2 = new Thread(()->{
6          process();
7      });
8      t1.start();
9      t2.start();
10 }
11 // thread body
12 void process() {
13     A a = new A();
14     a.f++;
15 }
```

Figure 2.1: A simple example to explain context-sensitive PTA.

In this simple example, thread *t1* and *t2* are two threads that have the same thread body, *process*. Under a context-insensitive PTA, the method calls at line 3 and 6 are indistinguishable, therefore the PTA thinks both threads access the same object *a* at line 14. Therefore, a false data race is detected.

To overcome this issue, we can use a "callsite-sensitive" PTA, which distinguish different method calls based on their callsite. In the same example above, *process* at line 3 is different from *process* called at line 6, because they are called from different lines. Hence, PTA treats them as two different function calls by creating two different nodes in the call graph. As a result, the variable *a* in different *process* functions will point to a different object of class *A*, and therefore the memory access at line 14 is not a shared memory access.

As we can see, context-sensitivity essentially models the context-switching process in a real execution. Other than choosing "callsite" as context identifier (callsite-sensitive) [42, 43], we can also choose the receiver objects as context identifiers (object-sensivie) [44]. However, we cannot arbitrarily distinguish all method calls based on their contexts, otherwise, the number of distinguishable method calls grows exponentially, thus making the analysis unscalable. To avoid this, PTA usually will set a ***k-limit*** of the number of context it tracks. For example, for a 2-context-sensitive PTA whose *k* is 2, the analysis will only distinguish method calls by the most recent two callsites.

K-limiting brings new imprecision for PTA. The example in Figure 2.2 is extended from the previous one, where we add an identify function *id* and pass variable *a* to it before it is accessed at line 15. Now the PTS of *a* at line 14 depends on the PTS of *a* in line 23 (the return value of *id*). Now if we use 2-callsite-sensitive to analyze the code, the simplified call graph is shown in Figure 2.3a. We can see that the contexts of both threads become identical once the PTA hits function *id2*. Therefore, the PTS of variable *a* in both threads may points to both the object created at line 13 from *t1* and *t2*. And during the

```
1  void main() {
2      Thread t1 = new Thread(()->{
3          process();
4      });
5      Thread t2 = new Thread(()->{
6          process();
7      });
8      t1.start();
9      t2.start();
10 }
11 // thread body
12 void process() {
13     A a = new A();
14     a = id(a);
15     a.f++;
16 }
17 A id(A a) {
18     return id2(a);
19 }
20 // id3, id4, id5...
21 ...
22 A idN(A a) {
23     return a;
24 }
```

Figure 2.2: An extended example from Figure 2.1. Variable *a1* and *a2* are both passed into *preprocess* before being passed into *process*. The function *preprocess* is a nested identity function that directly returns its argument after N ($N > 2$) function calls.

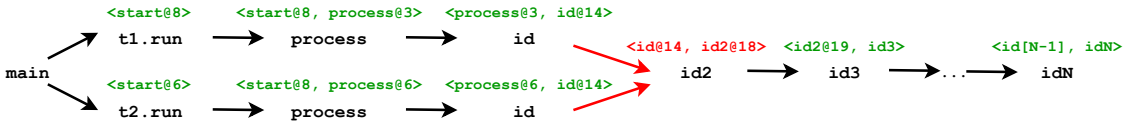
race detection, we have to conservatively consider the memory access at line 15 is a shared access thus report a data race between *t1* and *t2*.

2.2.3 Origin-Sensitive Pointer Analysis

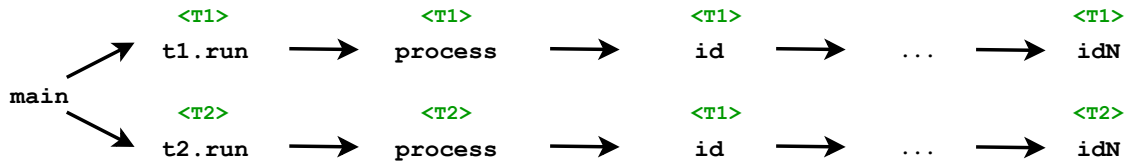
As we can see from the failing case in the last section, the limitation of k-context-sensitive PTA is it unconditionally updates the context at every function call. While the contexts of some call chains, such as the function *id*, don't contribute to the PTA precision. Moreover, for race detection, not all pointers, but only the ones that are shared across multiple threads need to be accurately computed.

Definition 1. Origin: an origin is a function that creates a thread and defines the thread entry.

Here we give a simple definition of *Origin*, and we use origin as the context identifier.



(a) Call graph under 2-callsite-sensitive PTA



(b) Call graph under origin-sensitive PTA

Figure 2.3: Call graph difference between 2-callsite-sensitive PTA and origin-sensitive PTA.

Specifically for Java, the origin includes following things:

1. The `run()V` functions from the `java.lang.Runnable` object given at a thread creation.
2. The overridden `run()V` method in any subclass of `java.lang.Thread`
3. Any API (function wrapper) that creates and/or starts a thread

Following the definition of origin, if we apply origin-sensitive PTA to the example in Figure 2.2, the `run()V` functions (written as a lambda expression for simplicity) passed to the `Thread` constructor are identified as two origins, say *T1* and *T2*. Now the call graph in PTA looks like Figure 2.3b. Since the invocation of method *idN* no longer affects the context update, each thread remains its own context across the whole analysis, therefore, the PTS for *a* at line 14 and 15 only includes the object *A* created at the current thread, and not causes any false positive for race detection.

To sum up, the origin sensitive makes the PTA thread-aware, so that it can distinguish the thread local objects from different threads. The lower frequency of context update also makes origin-sensitive PTA more more scalable than k-callsite-sensitive PTA.

2.2.4 Origin vs Other Contexts

The origin context is orthogonal to other choices of context such as object and callsites. It is a common practice to combine multiple contexts abstraction as a "hybrid context" to achieve better performance and accuracy in practice [45]. The same intuition also works for origin-sensitive PTA.

In object-oriented programming languages, such as Java, different kinds of threads are usually implemented as subclass of `java.lang.Thread` where their routines are defined within the `run()` method and the thread local variables are defined as fields of this thread class. Figure 2.4 exhibit such a program where `f` is a thread local field being accessed in the thread routine `run()`.

```
1  void main() {                    7  class A extends Thread {
2      A a1 = new A(); // o1        8      private int f = 0;
3      A a2 = new A(); // o2        9      @Override
4      a1.start();                  10     public void run() {
5      a2.start();                  11         f++;
6  }                                12     }}
```

Figure 2.4: An example showing origin-sensitive PTA cannot distinguish thread local variables while object-sensitive can.

If we mark `run()` as an origin entry, then the initialization of `A` is invoked outside of the origin, thus the field `f` of *a1* and *a2* cannot be distinguished by PTA. Now if we also consider the receiver object as a context, in addition to origin, then the PTA will distinguish the initialization calls for *a1* and *a2*, because they are on different receiver objects (*o1* and *o2*, respectively). As we can see in this simple example, origin can be combined with other contexts to achieve better precision for the PTA.

2.3 Checking Happens-Before Relation

The happens-before relation is a relation between the result of two events (even if executed out of order), so that one event should happen before the other. In Java specifically, a happens-before relationship is a guarantee that memory written to by statement *A* is visible to statement *B*, that is, that statement *A* completes its write before statement *B* starts its read [46].

The happened-before relation is formally defined as the least strict partial order on events such that:

1. If events a and b occur on the same process, $a \rightarrow b$ (a happens-before b) if the occurrence of event a preceded the occurrence of event b;

2. If event a is the sending of a message and event b is the reception of the message sent in event a, $a \rightarrow b$.

Like all strict partial orders, the happened-before relation is transitive, irreflexive and antisymmetric.

The most famous algorithm of computing happens-before relations in distributed system should be the vector clock from Lamport timestamps [4], which provide a partial ordering of events with a high overhead. And this concept later was adopted by program analysis to check data races. Fasttrack [2] leverages the vector clock and records only the $epoch$ of the last read to reduce the number of events need recording. iFT [47] is an efficient algorithm, that uses only the epochs of the access histories, which requires $O(1)$ operations to maintain the access history. For static analysis tool, [10] first abstract the program traces into a static happens-before graph, thus transform the reason of happens-before relations into a graph reachability problem.

SWORD is also a static analysis tool based on the concept of a static happens-before

Table 2.1: Original SHB Graph.

Intra-thread SHB Graph Construction	
Statement	***SHB Node & Edge***
❶ $x.f = y$	$\forall\langle o, \mathbb{O}_k\rangle \in pts(\langle x, \mathbb{O}_i\rangle)$, **write**($\langle o, \mathbb{O}_k\rangle.f$)
❷ $x = y.f$	$\forall\langle o, \mathbb{O}_k\rangle \in pts(\langle y, \mathbb{O}_i\rangle)$, **read**($\langle o, \mathbb{O}_k\rangle.f$)
❸ $x[idx] = y$	$\forall\langle o, \mathbb{O}_k\rangle \in pts(\langle x, \mathbb{O}_i\rangle)$, **write**($\langle o, \mathbb{O}_k\rangle.*$)
❹ $x = y[idx]$	$\forall\langle o, \mathbb{O}_k\rangle \in pts(\langle y, \mathbb{O}_i\rangle)$, **read**($\langle o, \mathbb{O}_k\rangle.*$)
① $x = y.m(a_1, ..., a_n)$	$\forall\langle m', \mathbb{O}_i\rangle \in dispatch(\langle y, \mathbb{O}_i\rangle, m)$, *add HB edge*: **call**($\langle m, \mathbb{O}_i\rangle$) $\Rightarrow$ $\mathbf{m}_{\textbf{first}}(\langle m, \mathbb{O}_i\rangle)$
❺ $synchronized(x)\{$ $\ldots\}$	$\forall\langle o, \mathbb{O}_k\rangle \in pts(\langle x, \mathbb{O}_i\rangle)$, **lock**($\langle o, \mathbb{O}_k\rangle$), **unlock**($\langle o, \mathbb{O}_k\rangle$)
Inter-thread SHB Graph Construction	
Statement	***SHB Node & Edge***
❻ $x.entry(c_1, ..., c_n)$	$\forall\langle entry, \mathbb{O}_j\rangle \in dispatch(\langle x, \mathbb{O}_i\rangle, entry)$, *add HB edge*: **entry**($\mathbb{O}_j$) $\Rightarrow$ $origin_{first}(\mathbb{O}_j)$
❼ $x.join()$	$\forall\langle join, \mathbb{O}_j\rangle \in dispatch(\langle x, \mathbb{O}_i\rangle, join)$, *add HB edge*: $origin_{last}(\mathbb{O}_j)$ $\Rightarrow$ **join**($\mathbb{O}_j$)

graph, while SWORD leverages some observation from general concurrent program to optimize the design of SHB Graph, thus achieves a significant performance improvement.

In this section, we start from introducing the naive SHB Graph used in ECHO, and discuss its limitation on large programs. Then we elaborate the optimized SHB Graph used in SWORD.

2.3.1 Static Happens-Before Graph

The SHB Graph is essentially a static trace of the target program. Each instruction corresponds to an *event*, which is the node of the SHB Graph. Events are connected by directed edges to represent their happens-bebfore relations. For example, an edge $A \rightarrow B$ means event *A* happens-before event *B*.

Table 2.1 reflects the mapping between instructions and events. The construction of SHB Graph starts from the `main()` method of a program. The program will be rep-

resented as a topologically sorted control-flow graph (CFG), where each node is a Basic Block (a sequence of instructions without jumps). The instructions within each basic block are then traversed sequentially. This traversal scheme guarantees the SHB Graph conservatively preserves the program execution order, except that both branches of a if-condition will be "executed".

Construction rule ❶ to ❺ handles inter-thread instructions, where all memory reads and writes are interpreted as read events and write events in the SHB Graph and all synchronization code blocks are interpreted as a lock event at the start together with a unlock event at the end. More importantly, two consecutive events are connected by happens-before edge, i.e. the event of the previous instruction has an directed edge pointing towards the current event.

Rule ❻ and ❼ handles thread fork/join instructions. The handling of thread fork/join is a fix-point algorithm, where we keep a list of "static threads" that we haven't traversed. At beginning, only the entry of the main thread is pushed to the list (i.e., the `main()` function). Upon the encounter of a thread fork, we query the call graph to identify the thread entry (i.e., the overridden `run()` method) and push the thread fork site into the list. At a thread join site, we query the PTA to figure out the thread object who is calling the `join()` method, and keep this information for later use. At next iteration, we pop a thread entry from the list, and draw an edge from its fork site to the head of the current static trace. Then we traverse the instructions within this child thread following the rules described above. When we hit the end of the current thread, we draw an HB edge from the end of the trace to the join site the its parent thread, based on the information we previously collected. The interative process keeps going until no new thread fork is found.

There are several corner cases need special handling to guarantee the termination and soundness of the above traversal:

Recusion. Recursive function calls will cause endless traversal during the SHB Graph

construction. To handle this situation, we maintain a static *call-stack* that records the current functions we are traversing. Every time we enter a function call, we check the call-stack to make sure we won't re-enter a recursive function.

Thread Creation in Loops. Some programs will spawn a set of threads within a loop, or recursively spawn threads and use condition checks to terminate. From source code perspective, there's only one thread fork site, while in real execution, a set of threads will be spawned. The soundness of our analysis will be undermined if the thread races with itself. To accommodate such situation, we unroll the loop or recursion once so that at least two static threads are created. Hence, our analysis can detect the self-races of those threads.

2.3.2 An Optimized SHB Graph

The SHB Graph described in Section 2.3.1 is similar to the one proposed in ECHO. In D4, in order to further optimized the incremental update of the SHB Graph, uses a more compact graph storage. Specifically, D4 constructs a unique subgraph for each method-/thread and connect the subgraphs with different happens-before edges instead of duplicate them.

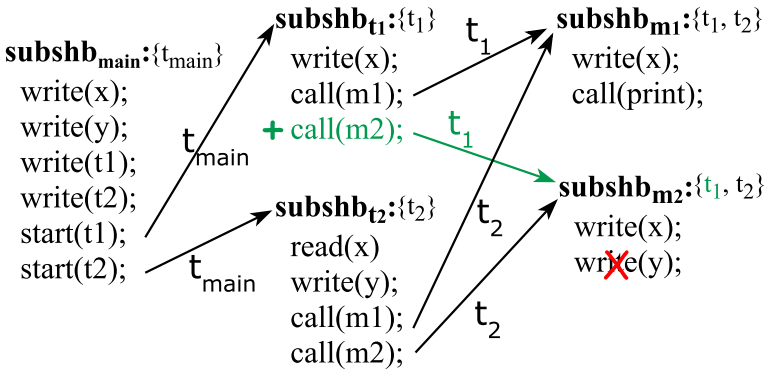


Figure 2.5: An SHB Graph constructed by D4 [1]

Figure 2.5 demonstrates an SHB Graph constructed by D4, which contains the sub-SHB Graph for thread *t1* and *t2* and method *m1* and *m2*. Such design enables minimal updates on the SHB Graph regarding a source code change, therefore it speed up the incremental race detection dramatically.

Both ECHO and D4 suffer from performance issue when trying to do a whole-program analysis upon large programs. This is mainly because their connectivity checking is almost a naive Depth-First-Search. For large programs, the number of events within the SHB Graph can be millions, therefore it is extremely inefficient to check the connectivity between two event. Meanwhile, since the number of edges involved in the graph enormous, it is also hard for us to effectively cache some pre-computed result, due to the low cache hit rate.

To address the performance issue above, we proposed an SHB Graph that is fully optimized for whole-program race detection. The key observation is that, the majority of the HB edges are intra-threaded, which can be represented in a much efficient way, while only the less common inter-threaded HB edges are critical. Therefore, we represent these two different types of edges in different ways:

Intra-thread HB edges are now represented as monotonically increasing IDs associated with each event. To achieve this, we have to duplicate the events in a method that is called in multiple places. This process can also be viewed as inlining all the method calls so that the trace of each thread is just a sequence of event, without inter-procedure calls. Now if we assign an monotonically increasing IDs for each event we create, for example, we assign the first event we create with ID 1 and the second with ID 2. The unique ID acts like a "static timestamp", where the events with smaller IDs happens-before the ones with larger IDs. This way, we turn the intra-thread connectivity checking into a constant time integer comparsion.

Inter-thread HB edges remain unchanged. But since the thread forks/joins only take

a small portion of all the events, we only need to keep much fewer explicit edges as we previously do.

2.3.3 Checking Happens-Before Relations

Now that checking the HB relations within a single thread is trivial, we only need to consider the inter-thread scenarios. The algorithm for checking HB relations is shown in Algorithm 2. The high level idea of this algorithm is we only check the connectivity between those nodes connected by inter-thread HB edges (i.e., thread fork/join sites).

Algorithm 1: CheckHappensBefore

input : $e1, e2$ - events, each associated with their belonging thread id
shb - SHB Graph, the trace for each thread is index by the thread id
output: $true/false$ - whether the two events has a happens-before relation

```
1  trace1 ← shb[e1.tid]
2  trace2 ← shb[e2.tid]
3  start1 ← findNextOut (trace1, e1)
4  end2 ← findLastIn (trace2, e2)
5  if canReach (shb, start1, end2) then
6  |   return true
7  else
8  |   end1 ← findLastIn (trace1, e1)
9  |   start2 ← findNextOut (trace2, e2)
10 |   return canReach(shb, start2, end1)
```

For a pair of events, say *e1* and *e2*, we first find the immediate succeeding outgoing node ($start1$) of *e1* (either a thread fork or the end of the trace) and the immediate preceding incoming node ($end2$) of *e2* (either a thread join or the start of the trace) (line 1-4). Then we checked the connectivity from $start1$ to $end2$. This is equivalent to checking the connectivity from *e1* to *e2*, as *e1* happens-before $start1$ and $end2$ happens-before e2 (the transitivity of happens-before relations). If *e1* can reach *e2* (line 5), then we have proven

e1 happens before e2. otherwise, we check the connectivity from the other direction (line 8-10). The `canReach` function in the pseudo-code is a DFS.

In this algorithm, all connectivity checks are around the synchronization related nodes. Since there are only a small number of those nodes, we can efficiently cache the synchronization nodes within each thread trace to speedup `findNextOut` and `findLastIn` and the connectivity result between those nodes to speedup `canReach`.

2.4 Origin Sharing Analysis

Another core part for race detection is identifying all the shared memory locations and the corresponding memory access events on them. To obtain such information, we extend the previous work, thread sharing analysis [48] (TSA), to origin sharing analysis (OSA). Comparing to the escape analysis that is widely used in compiler optimizations [49]. OSA and TSA care about if an object is accessed by multiple origins/threads, while escape analysis cares about whether the lifetime of an object exceeds the current scope. More specifically, four major limitations of escape analysis are listed in the TSA paper [48]:

1. a thread-escaped object may not be accessed by multiple threads.
2. an object being thread-escaped does not mean all data associated with the object are shared.
3. standard escape analysis algorithms do not directly work for array accesses.
4. the immutable objects that are only read after initialization should not be viewed as origin-sharing objects as they do not cause data races (while they can still be thread-escaped).

Except for the limitations listed above, classic escape analysis requires constructing an *Information FLow Graph* for the whole program and propagating the shared nodes in the graph until reaching a fix point. This process is expensive, and existing tools cannot finish

within 30 minutes on toy programs.

On the other hand, the OSA algorithm presented in this section naturally fits into the SHB Graph construction process, and only requires an efficient extra check to identify all potential thread sharing objects in the target program.

Comparing to the original TSA algorithm, there are two subtle changes. First, we extend the concept of being "thread sharing" to "origin sharing". Therefore, the algorithm can naturally be used for detecting shared objects across all kinds of origins. Second, the OSA algorithm fits into the SHB Graph process and collects all read/write events into their corrsponding *Reads Map* and *Writes Map*. These maps provide the inputs for data race detection.

In OSA, the shared memory is represented by the abstract objects constructed by PTA. For each abstract object, we maintain a ***write map*** and a ***read map***, where we build a mapping between the static threads and the read/write access events on the abstract object.

Algorithm 2 traverses the read and write events within each origin (line 2-3). For each read/write event, we query the PTA for the abstract object the current event is accessing (line 4). Then we push the current origin and the events to the read/write maps of the objects being accessed (line 7-8 and 11-12). Once the read/write maps are collected, we can use Algorithm 3 to identify shared abstract objects. The intuition behind Algorithm ref3 is that the *key set* of each read/write map relect the number of origins reading/writing the object. Therefore, we can conclude an object is origin-sharing under three conditions:

1. there are more than one origins writing to the object (line 4-5).
2. there are one origin writing the object and more than one origins reading the object (line 6-7).
3. there are exactly one origin writing the object and a different origin reading it (line 8-10).

Algorithm 2: OriginSharingAnalysis

```
   input  : PTA - pointer analysis
            CG - program call graph
   output: SharedObjects - a set of shared abstract objects
 1 origins ← all discovered origins
 2 for o ∈ origins do
 3     for event ∈ o do
 4         pts ← getPointsToSet(event)
 5         if isWrite(event) then
 6             for obj ∈ pts do
 7                 wmap ← getWritesMap(obj)
 8                 wmap ← (o, event)
 9         else if isRead(event) then
10             for obj ∈ pts do
11                 rmap ← getReadsMap(obj)
12                 rmap ← (o, event)
13 FindSharedObjects()
```

Algorithm 3: FindSharedObjects

```
   input  : AbstractObjects - all abstract objects encountered
   output: SharedObjects - a set of shared abstract objects
 1 for obj ∈ AbstractObjects do
 2     rmap ← getReadsMap(obj)
 3     wmap ← getWritesMap(obj)
 4     if wmap.size > 1 then
 5         SharedObjects ← obj
 6     else if wmap.size = 1&&rmap.size > 1 then
 7         SharedObjects ← obj
 8     else if wmap.size = 1&&rmap.size = 1 then
 9         if writes and reads are on different thread then
10             SharedObjects ← obj
```

2.5 Lockset Tracking

SWORD uses a fairly straightforward approach to handle locks. Each read/write event in the SHB Graph will be associated with a *lockset* at its creation. To compute the *lockset*, SWORD maintains a global list during the SHB Graph construction. The list consists of the locks being hold at the current event (represeted by the abstract object the lock variables points to). Hence, upon the creation of an event, we make a copy of the global list and associate it with the event, as its *lockset*.

Once a lock/unlock event is encountered, we query the PTA to see which abstract objects the lock variable may point to, and we update the *lockset* by pushing abstract objects into or popping the abstract object out of the global list.

To check if two events shares the same locks, we do an intersection between the two locksets associated with the events. If the intersection contains at least one abstract object, these two events are protected by the same locks.

Using canonical lockset ID. The above algorithm brings significant overhead on both memory and performance when the number of events is huge. Storing a list on each event and repeatedly doing the intersection are expensive. We observe that the number of different combinations among mutexes is much smaller than the number of conflict memory accesses we need to check for races. Therefore, we assign each combination of mutexes (including the empty lockset) a *canonical ID* and associate each event with an ID. This not only reduces the memory overhead for storing lock information in the SHB graph, but also speeds up the lockset checking process. All memory accesses with an identical lockset ID, or different IDs corresponding to overlapping locksets, are protected by the same lock(s), and the intersection IDs between two locksets can be cached for later checks.

2.6 Data Race Detection

Based on the three components we introduced. OSA enables SWORD to detect a set of object shared between thread along with their corresponding memory access events. SHB Graph enables SWORD to check the HB relation between a pair of events. The lockset tracking helps SWORD reason about the protection of locks.

For the last step, detecting data race, all we need to do is enumerate each pair of shared memory accesses and check their HB relations and lock sharing states respectively.

Algorithm 4: CheckDataRace

```
   input : SharedObjects - all shared objects within the target program
   output: races - a set of detected data races returned from EnumerateRaces
1  for obj ∈ SharedObjects do
2      rmap ← getReadsMap(obj)
3      wmap ← getWritesMap(obj)
4      for o, writes ∈ wmap do
5          for o', reads ∈ rmap do
6              if o ≠ o' then
7                  races ←EnumerateRaces (writes, reads)
8          if wmap.size() > 1 then
9              for o', writes' ∈ wmap do
10                 if o ≠ o' then
11                     races ← EnumerateRaces (writes, writes')
```

Algorithm 4 describes the overall process for checking data races. We first iterate over all shared objects and get their read maps and write maps (line 1-3). Since a data race must have at least one write event involved, we then start iterating over the origins and the write events within the origin on the write map (line 4). Finally, we try to find another set of write events or read events from different origin (on the same shared object) and pass

Algorithm 5: EnumerateRaces

```
   input : writes - a set of write events
           xs - a set of read or write events
           shb - the SHB Graph
   output: races - a set of detected data races
1  for w ∈ writes do
2      for x ∈ xs do
3          if checkHB (shb, w, x) = false then
4              if shareLock (w, x) = false then
5                  races ← (w, x)
```

the two sets of events to `EnumerateRace` to check if any pairs of those memory access events can be potential races (line 5-11).

The logic of `EnumerateRace` is rather simple, we enumerate over the two sets of given memory access events. For each pair of events, we consider them as a potential race if: 1. they don't have an HB relation; 2. they do not share a common lock.

2.6.1 Synchronization-Region-Based Race Detection

The data race checking algorithm can be inefficient if there's intensive reads/writes upon each shared object, which brings an explosion on the pair of memory accesses we need to enumerate. We observe that a huge number of enumerations can be omitted by a technique called *Synchronization-Region-Based Race Detection*. To explain the idea, we first give a formal definition of *synchronization regions*.

Definition 2. Synchronization Region: An event sequence in the same origin with no synchronization status changes.

More specifically to our SHB Graph, it means a sequence of read/write events whose lockset id is the same, and there's no thread fork/join in between. In the context of data race detection, all the reads and writes within the same synchronization region can be

regarded as a single read and write. This is because there are only two properties that determine if a memory access potentially races with another (assuming they are accessing the same object):

1. its immediate succeeding outgoing node and the immediate preceding incoming node in the SHB Graph;
2. its lockset.

For the memory accesses within the same synchronization region, the above two properties remain the same. Therefore they appear no difference in race detection. In almost all programs, we observe that a synchronization region protects a large sequence of memory accesses on the same origin-shared object(s). Therefore this technique significantly reduce the number of memory access pairs we need to enumerate. Algorithm 6 shows a simple way to shrink all read/write events within the same synchronization region into the first event of this region.

Other than shrinking the number of memory access events we need to enumerate, the concept of synchronization region also gives us a hint on how to simplify the SHB Graph. Logically, we "inline" all the method calls we have encountered during the SHB Graph construction, thus the frequently called methods will generate a large amount of events. Now that we know in a synchronization region, all the reads/writes on the same object are equivalent to data race detection, we can simply skip the repeated method calls that happen in the same synchronization region, thus reduce the size of the SHB Graph[1].

2.7 Collecting Stack Traces

Collecting stack trace information for the reported races is essential for people to understand how a race can be triggered.

[1]In a sense, this is an unsound optimization, because the pointer information in a method differs based on the calling environment. However, a synchronization region cannot cross origins and our PTA is flow-insensitive, thus making this optimization correct in our analysis

Algorithm 6: ShrinkMemoryAccess

input : xs - a set of read or write events on the same shared object within an origin (thread)
tid - the thread id which xs belongs to
shb - the SHB Graph

output: $xs^{'}$ - a set of read or write events in different synchronization-regions

```
1  trace ← shb[tid]
2  prev ← null
3  prevLastIn ← null
4  prevNextOut ← null
5  for x ∈ xs do
6      lastIn ← findLastIn (trace, x)
7      nextOut ← findNextOut (trace, x)
8      if prev = null then
9          xs' ← x
10     else if prev.lockset ≠ x.lockset ∨ prevLastIn ≠ lastIn ∨ prevNextout ≠
           nextOut then
11         xs' ← x
12     prev ← x
13     prevLastIn ← lastIn
14     prevNextOut ← nextOut
```

To support the collection of stack trace, we only need to make a small extension to our SHB Graph design. In our current SHB Graph, we will create a method call event at each method call (rule ① in Table 2.1). These call events help us to generate a list of method called, and their unique IDs suggest when they are called. However, the call event list does not contain any information about when each method is returned. To accommodate this, our solution is to associate a *return ID* with each call event. The return ID is the unique ID of the last event within the method, and this can be simply achieved by memorizing the current call event when we enter a method call, and set the return ID to the last event ID once we finish traversing all the instructions within a method.

Algorithm 7: CollectStackTrace

input : e - the event whose stack trace we want to collect
tid - the thread id which xs belongs to
shb - the SHB Graph
output: st - a list of call events representing the stack trace of e

```
1 trace ← shb[tid]
2 callEvents ← getCallEvents(trace)
3 for call ∈ callEvents do
4     if call.id > e.id then
5         break
6     else if call.rid ≥ e.id then
7         st ← call
```

The algorithm for computing the call stack trace for an event *e* is presented as Algorithm 7. The `getCallEvents` will return all the call events within the static trace of the current thread, sorted by the event ID (line 2). The we traverse the method calls in order. If the ID of a call is smaller than the target event *e*, yet its return ID ($call.rid$) is larger than *e*, this mean at the point *e* is executed, *call* has not returned yet (line 6). Therefore, we should insert the *call* to the stack trace list (line 7). We can stop the traversal once we

meet a call whose ID is larger than e, because all the following calls will happen after e (line 4-5).

2.8 Related and Future Work

Race detection has been considered as an important research topic for decades. Both static and dynamic algorithm has been proposed to tackle the problem. One of the key challenges is how to compute and represent *Happens-Before* Relationships. Offset-span labeling [50], which is an online scheme that labels threads in a fork-join graph, labels each task with a vector of tuples. Vector Clock [51, 4] records a clock for each thread in the system, and the virtual clock is increased upon every synchronization event. Two events are considered to be parallel if the two vector clock are not ordered. Flanagan et al [2] improve the vector clock algorithm by replacing heavyweight vector clocks with adaptive lightweight representation as they find the full generality of vector clocks is unnecessary in most cases.

Dynamic Race Detection Tools. Google's Thread Sanitizer [52], also known as TSAN, proposed a hybrid algorithm that uses both happens-before and lockset to detect data races. TSAN has been used to find hundreds of races in real world application. Helgrind [53] is a tool based on Valgrind [54]. Helgrind only detects happens-before relationships and it supports a subset of the dynamic annotations in TSAN. Intel's Inspector [55] is another dynamic data race detection tool that uses Intel PT [56] to trace the program. It uses a Concurrent Provenance Graph to record control, data and schedule dependencies.

Static Race Detection Tools. Chord [6], ECHO [10], D4 [1], RacerD [8] are static race detection tools on Java. Chord is based on object-sensitive, flow-insensitive alias analysis and escape analysis. RacerD leverages separation logic to detect races. It abandons the alias analysis and uses syntactic patterns to check alias information to achieve scalability. ECHO, SWROD, and D4 use field-sensitive but context-insensitive PTA. ECHO and

D4 primarily focus on the incremental race detection, hence the SHB Graph design for handling function calls is different from SWORD.

LOCKSMITH [7], RELAY [57] are two static race detectors for C that both focus on precise lockset reasoning. RELAY uses a context-sensitive bottom-up algorithm leveraging the function summaries and symbolic analysis. LOCKSMITH uses a context-, flow-sensitive correlation analysis to infer the protection of locks and applies a sharing analysis to rule out thread local variables.

3. SDROID: A SCALABLE WHOLE PROGRAM RACE DETECTOR FOR ANDROID

3.1 Android Background

Android applications are a representative class of modern software that contains complex interations between threads and events. For instance, in Android apps, there are hundreds of different types of events that can be created from the Activity lifecycles, UI interactions, or the system services [58]. Meanwhile, the app logic may create any number of normal Java threads and AsyncTask to improve performance.

Android Platform. All the Android apps run on top of the Android Framework, which orchestrates the control flow between apps' components and mediates all the intra-app, inter-app, and hardware-software communications. Typically, Android apps are written in Java (or partly in C/C++ throught Java Native Interface) and compiled into Dalvik bytecode that either executes on top a Dalvik virtual machine (Android version < 5.0) or then gets translated to native code and run on the Android Runtime (Android version ≥ 5.0).

Android app architecture. An app consists four types of components: (1) Services, for performing long-running background operations, (2) Content Providers, for managing data accesses, (3) Activities, for managing user interfaces, and (4) Broadcast receivers, for responding broadcast messages from the system or other applications.

Activities are the most critical components. Android apps are usually a collection of Activities and they switch between activities upon user interactions. For example, in the Youtube Android app, the "Home" screen corresponds to the `HomeActivity`. When the user clicks the "Search" box, the app switches to the `MediaSearchActivity`. And once the user click the setting button, the app goes to the `SettingActivity`. The lifetime of Activities is a state machine, where each state is associated with an event

so programmers can register callback functions to update the interfaces, e.g., upon the Activity creation, the callback function `onCreate()` is invoked to initialize the content of UIs, and upon Activity destruction `onDestroy()` is called to properly destruct the UIs. On the other hand, GUI components are placed within each Activity to form the specific UI layout. Each UI component can also be bound with callbacks in response to user interactions. Components are strongly isolated, the only way for components to communicate with each other is through message passing (in Android Framwork, this is called *Intent*).

Threads. There are three kinds of threads in Android system: 1. looper thread; 2. background thread; 3. binder thread. *Looper threads* is a long-running thread with a message loop. The looper thread receives messages and pushes them into a message queue, which later dispatches the message to its corresponding message handler. In other word, the looper thread is the "event-loop" we described in Chapter 1. The "main" thread of each Android app, also known as the UI thread, is nothing but a looper thread that handles all the interactions with UI components. Since each message is an "event", the messages can be pushed into the queue in different order, but each message is processed atomically. *Background threads* is just a normal thread that is typically used for background computation. *Binder threads* are used for inter-process communication between apps and Services, each Service maintains a pool of binder threads so that it can handle multiple service calls simultaneously.

Typically, the majority of the executions happen within the UI thread. When encountering the app encounters some heavy computations, they can be offloaded to background thread.

3.2 Harness Creation

In Android apps, there isn't an explicit main method as in other Java programs, that can be used as the analysis entry. Instead, the main entry of an Android app is the *Main Activity* (*i.e.*, the home screen) declared by the developers and instantiated by the Android Runtime (ART). Since the creation process of the main Activity is implicit from the source code of an Android app, we need to craft a harness that soundly models the process of the Activity creation. To do so, we first recognize the main activity by parsing and reading the *AndroidManifest.xml* within each Android apk to find the declared main Activity. Then an anlysis harness is generated as the code shown below. The lifecycle event handlers (`onCreate()`, `onDestory()`, etc.) for an activity construction and destruction are called in a sequence before and after all normal event handlers. Their order of invocation must be the same as the code below. Hence, the happens-before relations among lifecycle events are correctly retained. The normal event handlers are called in between the lifecycle event handlers in arbitrary order, because there is no execution order enforced by ART.

Code 1: An example harness for Andorid activities.

```
1 public static void harness() {
2   // a newly starting activity
3   XXXActivity a = new XXXActivity();
4   //lifecycle event handlers
5   a.onCreate();
6   a.onStart();
7   a.onResume();
8
9   // normal event handlers
10  a.onKeyDown(...);
11  a.onMenuItemSelected(...);
12  ...
```

```
13
14    // lifecycle event handlers
15    a.onPause();
16    a.onStop();
17    a.onDestory();
18  }
```

This harness allows us to start analyzing an Android apps from its main Activity. During the analysis, other Activities can be created by `startActivity()`. We create a harness using the same template for every Activity we meet, and resume our analysis by entering the harness method.

3.3 Event as Origin

The normal Java threads can be created the same way in Android apps. However, we also need to model the behavior of event handlers to detect races related to non-deterministic event handling. As we mentioned in Chapter 1, events and threads are inherently similar. Therefore, we extend our definition of *Origin* to unify the concept of thread and event.

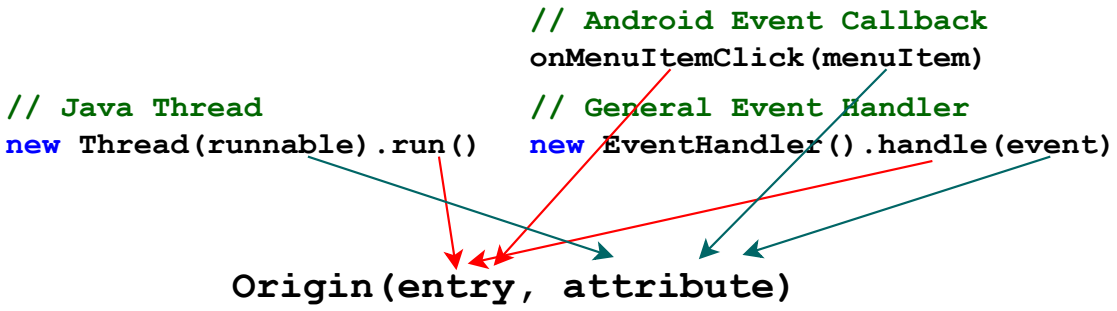


Figure 3.1: An "origin" view of threads and events.

Figure 3.1 shows a generalized definition of an origin where the *entry* corresponds to a

thread start or event handler, and the *attribute* corresponds to the data passed to the thread or event handler.

Table 3.1: The origin entries for SWORD and SDROID (incomplete).

Threads	**Events**
java.lang.Thread.start()	onClick(. . .)
java.lang.Runnable.run()	onMenuItemClick(. . .)
java.util.concurrent.Callable.call()	onLocationChanged(. . .)
. . .	. . .

* Due to the enormous number of possible entries, we only list some representitive ones.

Some Representative entries for origins are listed in Table 3.1. Because projects may have their customized thread creation APIs or event handlers, the actual selection of origin entries are subject to change, for example, some less important event handlers can be ignored.

In SDROID, we select a list of commonly used event handlers as origin entries (*e.g.,* the entries listed in Table 3.1) and handle them similarly as threads. More specifically, since there are too many event handlers exist in the Android framework, we categorize them into two types, namely *Entry Callbacks* and *Posted Callbacks* [21] ,and explain how we model each one of them in detail. Figure 3.2 displays an overview of how we model each type of event handlers (callbacks).

Entry Callbacks. Entry callbacks are event handlers externally invoked by ART. In other words, entry callbacks are those event handlers we invoked in the analysis harness [1]. These callbacks are bound with Activity lifecycles and UI interactions. We do not consider the lifecycle event handlers as origin entries, because their invocations are bounded to

[1]This is also the reason why we need to create a harness for them, because they are implicitly invoked by ART during the real executions.

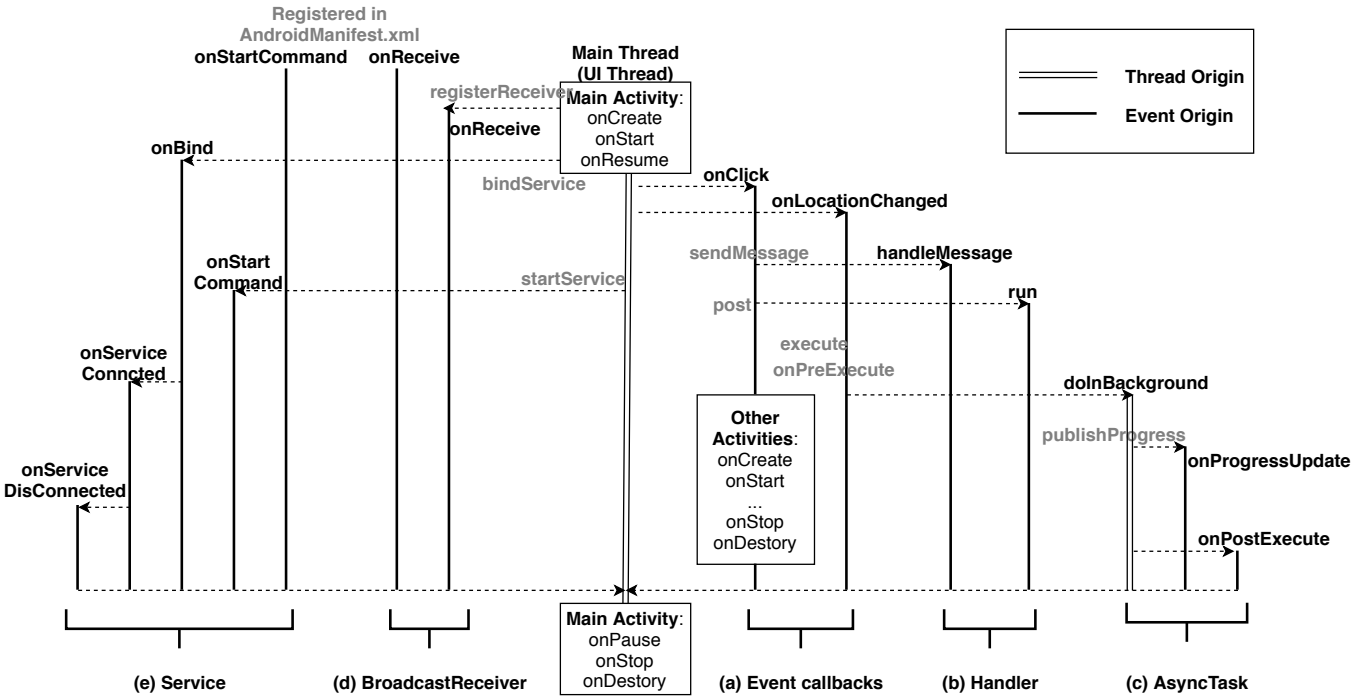


Figure 3.2: An overview of the Android thread model.

Activity's lifecycle and cannot be triggered by users in a non-deterministic way, therefore they reside in the same thread where the Acitivity is created (*e.g.,* the main thread in Figure 3.2). On the other hand, since the normal event handlers (*e.g.,* `onClick`) are triggered in arbitrary order, we consider them as origin entries. In other words, we see them as a kind of "special threads" during the race detection (Figure 3.2(a)).

Posted Callbacks. Posted callbacks are event handlers internally triggered by Android apps. They are related to Android components other than Activities: *Handler*, *AsyncTask*, *BroadcastReceiver* and *Service*.

First, the *Handler* is associated with a looper thread (a thread with a message queue), and it provides two APIs to either send a message to the message queue (`sendMessage`) or dispatch a `Runnable` task for the looper thread to execute (`post`). The `sendMessage` method delivers a message object to the message queue, and later this message

will be dispatched to the `handleMessage` method. The `post` method enqueues a `Runnable` object which will be executed later by the looper thread. Both methods post an event to the receiving looper thread. As shown in Figure 3.2(b), SDROID models them as threads created by the caller of these methods. This model guarantees the HB relation between the instructions in the caller and the instructions in the event handler are properly retained.

Second, the Android framework provides another high-level concurrency construct, *AsyncTask*, with which the main thread can easily communicate. Once the main thread called `execute`, `onPreExecute` will first be invoked on the main thread. Then the *AsyncTask* will execute its `doInBackground` method in a background thread. Once the background task is finished, the main thread will finally call `onPostExecute`. During the background processing, *AsyncTask* can update the progress of the task to the main thread by calling `publishProgress`. Then the event handler `onProgressUpdate` will be invoked in the main thread. Since all event handlers but `onPreExecute` are invoked non-deterministically. We recognize all the other three event handlers as origin entries, and create a static thread for them to detect races, as shown in Figure 3.2(c). A difference between `doInBackground` with other callbacks is that it is executed in a background thread, so it can cause both race conditions and data races.

Third, the *BroadcastReceiver* responds to the system-wide broadcasting. It only has one event handler `onReceive`, which will be triggered by `sendBroadcast(Intent)`. Ideally, we should identify and analyze the target `onReceive` method every time we hit a `sendBroadcast`, and recognize it as an origin entry. However, `sendBroadcast` does not specify the receiver that handles this broadcasted message, and the receiver can be registered based on the message type (a field of the `Intent` object passed to `sendBroadcast`) either in *AndroidManifest.xml* or by calling `registerReceiver` API. Finding the target receiver for each `sendBroadcast` requires precisely knowing the

value of the `Intent` object, which is generally undecidable. To overcome this technical limitation, we directly look at the the registration sites of receivers, and for each receiver registration, we create a new origin by calling their `onReceive` method, as shown in Figure 3.2(d).

Fourth, The *Service* handles background processing. There are two ways to interact with a *Service*:

1. You can explicitly invoke a service by calling `startService(Intent)`. The service will be created if it hasn't been yet, and the lifecycle event handler `onCreate` will be called. After that, the handler `onStartCommand` will be invoked to performance the actual service logic. To handle this kind of service invocation, we can craft a harness for a service the same way we did for activities. The harness only consists of `onCreate` and `onStartCommand`, and the harness should be considered as an origin entry, because the service operations happen in the background threads.

2. Another approach to interact with service is to bind an application to it. Once the API `bindService` is called, the event handler `onBind` and `onServiceConnected` will be called subsequently. After the connection between the client and service is established, the application can access the data from the service synchronously through a *Binder* object return from `onBind`. If later the connection is lost, the callback `onServiceDisconncted` will be invoked. As shown in Figure 3.2(e), we recognize all three callbacks as origin entries and analyze them one after another to reflect their happens-before relations.

The accurate analysis on *Services* also requires the reasoning of the value of `Intent` object. Because the *Services* are also registered in the *AndroidManifest.xml*, we can apply the same scheme for handling *Receivers*.

3.4 Race Detection for Android

Since our goal is to extend the race detection framework introduced in Chapter 2 to support the thread model in Android. The race detection process is mostly unchanged. Upon each origin entry, SDROID generates a static trace for that origin. The static traces between origins are connected by inter-origin edges (*i.e.,* the inter-thread edges).

The only difference is we somehow still needs to distinguish between thread origins and event origins, because events essentially can only be executed on a single thread (the looper thread), therefore they cannot have data races with other events. To accommodate that, we introduce a global lock for event origins. The global lock will be acquired once we enter an event origin and released when we exit. This approach guarantees that no data races will be detected between the event handlers, because they all run on the UI thread.

As a result, our analysis framework is able to model such a concurrent event-driven system, and seamlessly detect data races between origins. As for detecting race conditions, since race conditions are mostly semantic bugs that highly depend on specific code logic, we cannot easily detect them without user specification. However, we can apply different heuristics (i.e., bug patterns) to detect a subset of those race conditions. For example, if one event only write to the an shared object with value `null`, while another event only reads this object, then it is very likely to cause a *Null Pointer Exception*. Such heuristics are easy to implement in our race detection framework, and they can further cooperate with user specifications to detect a larger variations of race conditions we introduced in Chapter 1.

3.5 Related and Future Work

Races in event-driven programs have attracted much attention recently [59, 18, 17, 19, 20, 60, 22, 21]. Event-based races are even more challenging to detect than thread-based races because most events are asynchronous and the event handlers can be triggered

in many different ways. Moreover, the difficulty in detecting event-based races can be exacerbated by interactions between threads and events, which are common in distributed systems and mobile apps.

The state-of-the-art dynamic race detectors [52, 61] do not perform well in detecting event-based races, due to the large space of causal orders among event handlers and threads, as well as the large number of events and event sequences.

Dynamic race detectors that specifically targeting at Android apps [59, 18, 19, 17] all need a special device to log the execution traces for Android apps, thus making the tool even harder to use in the development process. Moreover, to effectively expose bugs in the traces, the user needs to run the app with a sequence of predefined interactions to properly trigger those events, as a result, it is extremely hard for those techniques to expose previously unknown bugs. To accurately analyze the collected traces, all previous tools require several seconds to several hours to process the traces offline. Comparing with these tools, SDROID does not require a logging device, and it usually requires only several minutes to analyze a large size Android app. Furthermore, SDROID can detect many more potential bugs exist in the Android apps.

For static analyzers, FSCS [60] only support event-driven system in general, without any support for Android system. DEvA [22] is a static technique that requires manual description of the framework, and it detects "event anomaly" in general. Therefore it is much harder to use and can cause false negatives due to the incomplete description. Besides, this technique in a sense is orthogonal to SDROID as we mentioned in Section 3.4. Adopting certain levels of user specification can help SDROID to better detect potential semantic bugs between events (i.e., event anomalies). nAdroid [21] is a static tool that focus on detecting null pointer exceptions in Android apps. It is the first work trying to view events and threads thus enabling using existing data race detector to analyze Android apps. However, nAdroid uses Chord [6] as it's underlying framework, which is pretty old

and has no HB relation reasoning. In consequence, nAdroid has to apply a set of unsound heuristics to filter out detected bugs. Moreover, the Android model proposed by nAdroid is incomplete, while SDROID first proposed a complete model to abstract all kinds of possible behavior in an Android app and considering more inherent HB relations between Android events. SIERRA [20] uses symbolic execution to analyze Android apps statically. In general, symbolic execution cannot scale to large programs, thus SDROID has a much better performance over SIERRA. On the other hand, SIERRA considers some more sophisticated event handling scenarios, such as pausing an Activity and then re-enter this Activity multiple times, causing a sequence of `onPause` and `onResume` events to be triggered. Symbolic analysis enables SIERRA to accurately analyze these cases SDROID does not handle accurately, thus reporting less false positives in general. We consider those advantages in SIERRA as our future research directions.

In summary, SDROID leverages a data race detection framework that has been proven successful in Java programs and extends this framework to the Android system by proposing a complete Android thread model that conservatively models most kinds of potential behavior in an Android app. There are still some corner cases we do not handle. For example, users can associate a `Looper` object to any Java thread and turn it into a looper thread just like the UI thread. Our model currently does not handle this case. Moreover, we fail to find some good heuristics to detect common race conditions between events in practice, it is possible to carry out more case studies to understand race conditions in real-world Android app as well as incorporating user provided specifications.

4. EVALUATION [1]

We implemented both SWORD and SDROID based on ECHO, which is based on the program analysis framework WALA [62]. We evaluated SWORD on a collection of 12 different real-world Java applications from DaCapo-9.12 [63], petablox [64] and Github. In this section, we report the results of our experiments and make some discussion based on the experimental results.

4.1 Methodology

For SWORD, we mainly compare our tool with the most recent two tools available, RacerD and D4. RacerD is inherently compositional and require annotations to increase its coverage. Therefore, for small benchmark, we records the classes being analyzed and manually annotated then as *@ThreadSafe*, so that we guarantees both tools achieve the same code coverage. Since for large benchmarks, we are unable to precisely annotate every class analyzed, this experiment is only performed at 7 benchmarks. Besides, since RacerD does not consider external libraries, we also configured SWORD to skip analyzing all external library. Since D4 uses the same framework as SWORD, we simply run D4's initial analysis (which analyzes the whole program) and record the execution time to show the performance improvement achieved by SWORD. As for the number of races, D4 and SWORD technically reports the same number of races, but they organize the race report in a different way, so we skip recording those numbers as they are not providing useful information.

For Android race detection, RacerD is the only publicly available static race detector that supports Android apps. However, RacerD can only be integrated into the command-

[1] © [2019] IEEE. Adapted, with permission, from Y. Li, B. Liu, and J. Huang, "Sword: A Scalable Whole Program Race Detector for Java" in 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 75–78, IEEE, 2019.

Table 4.1: Performance and accuracy for SWORD and RacerD on different benchmarks.

		RacerD		SWORD	
Program	LOC	Alarms*	Runtime†	Alarms	Runtime
tsp	454	6/44	1.2s	46	1s
elevator	1088	1/52	2.8s	45	1s
weblech	1322	10/53	2s	13	1.4s
sor	7176	44/64	3.3s	8	1.1s
sunflow	24713	59/1632	10.3s	1150	135s
lusearch	48128	222/2786	25s	1420	13s
avrora	70057	79/4788	30s	27	74s

* *# of alarms without annotations / # of alarms with annotations*
† *RacerD has similar performance with or without annotations*

line building process, and we are unable to successfully integrating RacerD for most of the open-source project. Therefore, we are only able to evaluate SDROID alone and discuss its accuracy and performance based on the statistics provided from previous papers. All open-source Android apps are collected from F-Droid [65]. In addition, we also pick two well-known commercial Android apps (Chrome and Zoom) to test if our analyzer can scale to common commercial programs.

4.2 Evaluation of SWORD

Table 4.1 shows the whole-program analysis results for RacerD and SWORD. The table is collected from SWORD's original paper [41], therefore the performance results cannot fully reflect the optimizations we discussed in Chapter 2. Nevertheless, this experimental result is sufficient to show the effectiveness of SWORD on whole-program race detection.

Table 4.2 shows the performance comparision between D4 and SWORD. As we can see, SWORD outperforms D4 in all benchmarks and achieves an overall 10x speedup.

Table 4.2: Performance comparison between SWORD and D4.

Program	KLOC	D4(1-Origin) Runtime	SWORD Runtime	SWORD Speedup
Batik	191	14.93s	3.35s	4.45x
Eclipse	239	8.03s	2.3s	3.49x
Pmd	104	13.32s	0.91s	14.63x
Sunflow	169	26.01s	2.34	11.12x
Xalan	175	35.73s	3.33s	10.73x
Zookeeper	108	271.20s	41.39s	6.55x

4.2.1 Performance

For small benchmarks, SWORD and RacerD both manifest a fast detection speed. When the programs grow large, RacerD tends to have a better performance on average, because it utilizes a cheap *ownership analysis* to replace PTA. This design makes the performance of RacerD mainly depends on program sizes and is not affected by complexity. In contrast, SWORD performs a origin-sensitive PTA, which is much more precise but also more expensive. Nevertheless, SWORD can complete the detection within 3 minutes for all the benchmarks. Another interesting observation is the performance of SWORD is disproportionate to the program size. For example, *sunflow* has smaller program size than *lusearch*, but it contains more abstract threads and shared memory accesses, which require significantly more computation (135s vs 13s) to determine the happes-before relations and locksets.

Comparing to D4, SWORD exhibit a significant performance improvement, this is expected, since SWORD is fully optimized for whole program race detection. However, the optimizations on SWORD cannot easily adopted by D4. This is because D4 do not "inline" each method call, thus the thread local trace is not a sequence. This results in a much more expensive HB checking and less efficient memory access shrinking.

4.2.2 Precision

The *Alarms* columns in Table 4.1 show the reported race number of SWORD and RacerD. The soundness of SWORD is guaranteed by PTA and SHB graph design. In order to verify the accuracy for all benchmarks, we compared SWORD's reports with RacerD's reports on annotated benchmarks, because they had the same code coverage. we manually checked all reported races in the first 4 benchmarks and confirmed all extra alarms reported by RacerD were false positives. The rest 3 large benchmarks all contain more than a thousand alarms, therefore we checked them by sampling. We categorized all reported races by their classes. For each class, we randomly pick three alarms to check:

1. The alarm is reported by RacerD but not by SWORD;
2. The alarm is reported by SWORD but not by RacerD;
3. The alarm is reported by both tools.

After checking all the sample alarms, we conclude that, for the first type of alarms, all of them are false positives. For the second type, there exist some real races, which means RacerD has false negatives. The last type of alarms consists of both false positives and real races.

4.2.3 Case Study

We use two cases to illustrate the imprecision in RacerD that can be easily addressed by SWORD's HB relation checking and precise PTA. The first example below is from *Weblech*, which is a multithreaded web crawler implemented in Java.

```
1 // TextSpider.Java: main function
2 public static void main(String[] args) {
3     ...
4     Spider spider = new Spider();
5     spider.readCheckpoint();
6     ...
7     spider.start();
8 }
```

```
9
10 // Spider.java: Spider class implementation
11 Class Spider extend Runnable {
12     private queue;
13
14     public void readCheckpoint() {
15         ...
16         // write on queue, RACE
17         queue = (DownloadQueue) ois.readObject();
18         ...
19     }
20
21     public void run() {
22         ...
23         // read on queue, RACE
24         synchronized(queue) {...}
25         ...
26     }
27
28     public void start() {
29         ...
30         Thread t = new Thread(this, "thread 1");
31         t.start()
32         ...
33     }
34 }
```

`Spider` is a Runnable class, and will spawn multiple threads and assign itself as the thread routine in the `start` function. The method `readCheckpoint` will only be called once in `main` before `Spider.start` is called. Therefore there is no concurrent access with field `queue` in this method. Since RacerD does not reason about happens-before relation, it can only assume `readCheckpoint` *may happen in parallel* with `Spider.run` (the thread routine), thus reporting a read-write race on `queue`. This false alarm can be easily eliminate by SWORD due to its HB checking.

The second case comes from a simplified version of *sor*, which is a micro-benchmark from petablox.

```
1 class Sor {
2     // shared variables
3     static black = new float[M][N];
4     static red = new float[M][N];
5     ...
6     public static void main(String[] args) {
7         for (i=0;i<proc;i++) {
8             // start and end are indices to black and red
9             new sor_first_row(start,end).start();
```

```
10              ...
11          }
12      }
13  }
14
15  class sor_first_row {
16      float[][] black_ = Sor.black;
17      float[][] red_ = Sor.red;
18      ...
19      public void run() {
20          ...
21          for (i = 0; i < Sor.iterations; i++) {
22              for (j = start; j <= end; j++) {
23                  // RACE due to overlapping indices
24                  black_[j][k] = red_[j-1][k] + red_[j+1][k];
25                  red_[j][k] = black[j-1][k] + black_[j+1][k];
26              }
27          }
28          ...
29      }
30  }
```

`sor_first_row` is a class extends `Thread`. Argument *start* and *end* are consecutive ranges (e.g. 1 to 5; 6 to 10). The shared static array *black* and *red* are assigned to thread local variables *black_* and *red_*. All threads perform reads and writes on *black_* and *red_*, and the index of these array accesses will overlap at indices $j-1$ and $j+1$. However, RacerD treats *black_* and *red_* as pure thread local variables instead of recognizing they refer to the static shared arrays due to missing precise PTA. Therefore, RacerD fail to detect the real races on *black_* and *red_* (line 24 and 25), no matter it runs with or without annotations.

4.3 Evaluation of SDROID

We mainly show the performance evaluation of SDROID on different kinds of Android apps. Table 4.3 compares how effective is SDROID using origin-sensitive PTA to analyze android programs. Since checking the race report of all these Android apps is very hard [2], we picked Firefox-Focus as a Representative app and did a comprehensive study on its bug report, the conclusion is discussed in the precision section.

[2]It's hard to manually reason about potential user interactions. Besides, most of the commercial Android apps are not open-sourced.

Table 4.3: The performance result for SDROID on different Android apps(Time: s).

	0-Ctx		Origin			1-CFA		2-CFA		1-Obj		2-Obj	
App	**PA**	**Total**	**#O**	**PA**	**Total/SD**	**PA**	**Total/SD**	**PA**	**Total/SD**	**PA**	**Total/SD**	**PA**	**Total/SD**
ConnectBot	*2.40*	2.49	11	*5.45*	5.57 /124%	*23.85*	23.99/8.63x	*3512.66*	3512.93/1409x	*>4h*	-	*>4h*	-
Sipdriod	*5.80*	16.02	15	*31.48*	228.33/1327%	*14.33*	40.88/1.55x	*3436.02*	3452.33/215x	*>4h*	-	*>4h*	-
K-9 Mail	*6.56*	8.59	23	*14.73*	19.49/127%	*30.88*	33.32/2.88x	*4284.43*	4288.31/498x	*>4h*	-	*>4h*	-
Tasks	*6.90*	7.10	7	*12.72*	12.90/82%	*117.63*	117.77/15.59x	*8080.92*	8081.12/1137x	*>4h*	-	*>4h*	-
FBReader	*6.66*	7.49	15	*20.16*	23.33 /211%	*45.26*	52.79/6.05x	*2.97h*	3.10h/1482x	*>4h*	-	*>4h*	-
VLC	*5.35*	5.39	4	*46.40*	46.44/762%	*25.40*	25.44/3.72x	*3234.61*	3234.68/599x	*>4h*	-	*>4h*	-
FireFox Focus	*3.84*	4.08	8	*15.46*	15.76/286%	*17.96*	18.34/3.50x	*>4h*	-	*>4h*	-	*>4h*	-
Telegram	*20.82*	41.76	134	*199.79*	372.93/793%	*83.31*	171.42/3.10x	*>4h*	-	*>4h*	-	*>4h*	-
Zoom	*36.77*	37.62	15	*148.01*	149.01/296%	*198.59*	200.47/4.33x	*>4h*	-	*>4h*	-	*>4h*	-
Chrome	*6.14*	7.35	34	*108.76*	111.79/1421%	*18.43*	22.72/2.09x	*>4h*	-	*>4h*	-	*>4h*	-

"-": the corresponding pointer analysis runs out of time.

4.3.1 Performance

As Table 4.3 has shown, origin-sensitive analysis brings a significant performance overhead against context-insensitive analysis. This is expected, however, we find the performance overhead brought by origin-sensitive analysis is more significant when analyzing Android apps. Because most of the program logic are written within event handler (origins), as a result, there are more events in origins, which incurs more expensive PTA.

Regardless, our analysis can still finish analyzing all Android apps within 10 minutes. This is much more efficient than all other context-sensitive analyses such as 2-cfa and 1-/2-object sensitivity. Comparing to all other static tools, we also see a superior performance, and all other tools did not show their capability of analyzing large commercial programs like Zoom and Chrome.

4.3.2 Precision

At this moment, we are not able to fully verify the precision of our Android race detector, mainly due to two reasons. First, a concurrent program an Android app may be, most existing apps are single-threaded, therefore we cannot find many data races. Second, as for race conditions, they require a deep understanding of the apps' logic, and can be

very hard to manually check without extensive help from developers. As a result, we pick Firefox-Focus as a representative and mainly focus on studying the data races we detect. Among the 15 races detected, 2 of them are confirmed to be real races, one of which will be elaborated in the next section [3]. Other 13 false positives are mainly due to lacking the reasoning of path-condition and the imprecision in PTA caused by container object such as *ArrayList* or *HashMap*. Those are all expected false positive cases, and we consider handling them as our future work.

4.3.3 Case Study

SDROID is able to finish in 15s on FireFox Focus 8.0.15 (a privacy-focused mobile browser), and detected two previously unknown bugs (both reported in Bug-1581940) confirmed by developers from Mozilla. By the time I'm writing this thesis, one of the bug has already been fixed in the newer version, therefore we use it as an case study. A simplified code snippet is presented below:

```
1  // called from Gecko background thread
2  public synchronized IChildProcess bind() {
3      ...
4      Context ctx = GeckoAppShell.getAppCtx();//RACE
5      ...
6  }
7  // called from MainActivity.onCreate()
8  @UiThread
9  public void attachTo(Context context) {
10     ...
11     Context appCtx = context.getAppCtx();
12     if (!appCtx.equals(GeckoAppShell.getAppCtx())) {
13         GeckoAppShell.setAppCtx(appCtx);//RACE
14         ...
15     }
16     ...
17 }
```

The code involves both FireFox Focus and FireFox's browser engine, Gecko. Upon the app initialization, `getAppCtx` and `etAppCtx` are called without synchronizations, one from Android UI thread (through `onCreate` event handler), the other from Gecko

[3]The two races have similar root causes, therefore we only elaborate one.

engine's background thread. Although in reality, the creation order between UI thread and Gecko background thread keeps the race from happening, it is possible for Gecko engine to read an uninitialized application context thus leads to crash.

5. CONCLUSION

This thesis presents a fully optimized framework for doing whole-program race detection on Java and Android programs.

To achieve a scalable whole-program race detection framework, we first propose a new context abstraction called origin for pointer analysis. This context abstraction allows PTA to distinguish thread sharing objects versus thread-local object precisely, thus providing a sufficient precision while maintaining reasonable performance. Besides, we leverage the typical characteristics of the multithreaded program traces to effectively represent the HB relations and reduce the overall SHB Graph size, which further enables efficient caching to optimize the race detection performance greatly. We also propose a fast algorithm for collecting stack-trace for our bug reports, thus allowing developers to understand the race alarms better. Our evaluation shows SWORD achieves overall a 10x speedup over previous work D4.

Next, we extend our efficient race detection framework to the Android system. To achieve this, we unify the concept of events and threads as origins and propose a comprehensive Android thread model that allows us to analyze Android as a standard multithreaded program. As a result, we successfully extend our Java race detection framework to Android and have found several previously unknown bugs.

Besides the technical contribution, we also make some engineering efforts to integrate the race detectors with popular IDEs such as Eclipse and Intellij IDEA leveraging Language Server Protocol.

In summary, this thesis presents the SWORD and SDROID, which support efficient and scalable race detection on real-world Java and Android programs. In the short term, I will continue this line of research, especially supporting path-sensitive analysis to improve

analysis accuracy. As for Android, it is still unclear whether our model soundly considers all potential behaviors of an Android app. Therefore it will be useful to further formalize the behavior of the Android system. Besides, effectively detecting harmful race conditions in event-driven systems can still be explored, especially under the circumstance where user specifications are not available. In the longer term, I'm more interested in more advanced type systems and especially the incorporation between type system and runtime checks (such as the contract system). I also believe a more advanced type system can bridge the gap between static analysis and formal verification. I'm also interested in applying static analysis/formal verification techniques into some less-studied systems such as layout rendering systems.

REFERENCES

[1] B. Liu and J. Huang, "D4: fast concurrency debugging with parallel differential analysis," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pp. 359–373, 2018.

[2] C. Flanagan and S. N. Freund, "Fasttrack: Efficient and precise dynamic race detection," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, (New York, NY, USA), pp. 121–133, ACM, 2009.

[3] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, pp. 391–411, Nov. 1997.

[4] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," vol. 21, (New York, NY, USA), pp. 558–565, ACM, July 1978.

[5] D. Engler and K. Ashcraft, "Racerx: Effective, static detection of race conditions and deadlocks," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, (New York, NY, USA), pp. 237–252, ACM, 2003.

[6] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for java," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, (New York, NY, USA), pp. 308–319, ACM, 2006.

[7] P. Pratikakis, J. S. Foster, and M. Hicks, "Locksmith: context-sensitive correlation analysis for race detection," *Acm Sigplan Notices*, vol. 41, no. 6, pp. 320–331, 2006.

[8] S. Blackshear, N. Gorogiannis, P. W. O'Hearn, and I. Sergey, "Racerd: compositional static race detection," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–28, 2018.

[9] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pp. 55–74, IEEE, 2002.

[10] S. Zhan and J. Huang, "Echo: Instantaneous in situ race detection in the ide," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, (New York, NY, USA), pp. 775–786, ACM, 2016.

[11] K. Sen, "Race directed random testing of concurrent programs," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, (New York, NY, USA), pp. 11–21, ACM, 2008.

[12] H. C. Lauer and R. M. Needham, "On the duality of operating system structures," *ACM SIGOPS Operating Systems Review*, vol. 13, no. 2, pp. 3–19, 1979.

[13] J. Ousterhout, "Why threads are a bad idea (for most purposes)," in *Presentation given at the 1996 Usenix Annual Technical Conference*, vol. 5, San Diego, CA, USA, 1996.

[14] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[15] J. R. Von Behren, J. Condit, and E. A. Brewer, "Why events are a bad idea (for high-concurrency servers).," in *HotOS*, pp. 19–24, 2003.

[16] B. Zhou, I. Neamtiu, and R. Gupta, "Experience report: How do bug characteristics differ across severity classes: A multi-platform study," in *2015 IEEE 26th Interna-*

tional Symposium on Software Reliability Engineering (ISSRE), pp. 507–517, IEEE, 2015.

[17] P. Maiya, A. Kanade, and R. Majumdar, "Race detection for android applications," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 316–325, 2014.

[18] P. Bielik, V. Raychev, and M. Vechev, "Scalable race detection for android applications," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 332–348, 2015.

[19] Y. Hu, I. Neamtiu, and A. Alavi, "Automatically verifying and reproducing event-based races in android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 377–388, 2016.

[20] Y. Hu and I. Neamtiu, "Static detection of event-based races in android apps," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 257–270, 2018.

[21] X. Fu, D. Lee, and C. Jung, "nadroid: statically detecting ordering violations in android applications," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pp. 62–74, 2018.

[22] G. Safi, A. Shahbazian, W. G. Halfond, and N. Medvidovic, "Detecting event anomalies in event-based systems," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 25–37, 2015.

[23] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang, "Static data race detection for concurrent programs with asynchronous calls," in *Proceedings of the 7th joint meeting of the European software engineering conference and the acm sigsoft symposium on the foundations of software engineering*, pp. 13–22, 2009.

[24] G. Ramalingam, "The undecidability of aliasing," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1467–1471, 1994.

[25] L. O. Andersen, "Program analysis and specialization for the c programming language," tech. rep., 1994.

[26] J. Dietrich, N. Hollingum, and B. Scholz, "Giga-scale exhaustive points-to analysis for java in under a minute," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, (New York, NY, USA), pp. 535–551, ACM, 2015.

[27] D. Grove and C. Chambers, "A framework for call graph construction algorithms," *ACM Trans. Program. Lang. Syst.*, vol. 23, pp. 685–746, Nov. 2001.

[28] B. Hardekopf and C. Lin, "The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, (New York, NY, USA), pp. 290–299, ACM, 2007.

[29] G. Kastrinis and Y. Smaragdakis, "Hybrid context-sensitivity for points-to analysis," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, (New York, NY, USA), pp. 423–434, ACM, 2013.

[30] J.-s. Yur, B. G. Ryder, and W. A. Landi, "An incremental flow- and context-sensitive pointer aliasing analysis," in *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, (New York, NY, USA), pp. 442–451, ACM, 1999.

[31] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for java," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, pp. 1–41, Jan. 2005.

[32] B. G. Ryder, "Dimensions of precision in reference analysis of object-oriented programming languages," in *Proceedings of the 12th International Conference on*

Compiler Construction, CC'03, (Berlin, Heidelberg), pp. 126–137, Springer-Verlag, 2003.

[33] Y. Smaragdakis and G. Balatsouras, "Pointer analysis," *Found. Trends Program. Lang.*, vol. 2, pp. 1–69, Apr. 2015.

[34] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras, "Introspective analysis: Context-sensitivity, across the board," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, (New York, NY, USA), pp. 485–495, ACM, 2014.

[35] J. Späth, L. N. Q. Do, K. Ali, and E. Bodden, "Boomerang: Demand-driven flow- and context-sensitive pointer analysis for java," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)* (S. Krishnamurthi and B. S. Lerner, eds.), vol. 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 22:1–22:26, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016.

[36] M. Sridharan and R. Bodík, "Refinement-based context-sensitive points-to analysis for java," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, (New York, NY, USA), pp. 387–400, ACM, 2006.

[37] M. Sridharan and S. J. Fink, "The complexity of andersen's analysis in practice," in *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, (Berlin, Heidelberg), pp. 205–221, Springer-Verlag, 2009.

[38] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, (New York, NY, USA), pp. 131–144, ACM, 2004.

[39] "The true cost of a software bug." https://www.celerity.com/the-true-cost-of-a-software-bug.

[40] "Language server protocol." https://microsoft.github.io/language-server-protocol/.

[41] Y. Li, B. Liu, and J. Huang, "Sword: A scalable whole program race detector for java," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 75–78, IEEE, 2019.

[42] M. Sharir, A. Pnueli, *et al.*, *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences . . . , 1978.

[43] O. Shivers, *Control-flow analysis of higher-order languages*. PhD thesis, PhD thesis, Carnegie Mellon University, 1991.

[44] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to and side-effect analyses for java," in *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pp. 1–11, 2002.

[45] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: understanding object-sensitivity," in *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 17–30, 2011.

[46] "Memory consistency error." https://docs.oracle.com/javase/tutorial/essential/concurrency/memconsist.html, 2019.

[47] O.-K. Ha and Y.-K. Jun, "An efficient algorithm for on-the-fly data race detection using an epoch-based technique," vol. 2015, (New York, NY, United States), pp. 13:13–13:13, Hindawi Publishing Corp., Jan. 2015.

[48] J. Huang, "Scalable thread sharing analysis," in *Proceedings of the 38th International Conference on Software Engineering*, pp. 1097–1108, 2016.

[49] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff, "Escape analysis for java," *Acm Sigplan Notices*, vol. 34, no. 10, pp. 1–19, 1999.

[50] J. Mellor-Crummey, "On-the-fly detection of data races for programs with nested fork-join parallelism," in *Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pp. 24–33, IEEE, 1991.

[51] F. Mattern *et al.*, *Virtual time and global states of distributed systems*. Citeseer, 1988.

[52] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: Data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, (New York, NY, USA), pp. 62–71, ACM, 2009.

[53] A. Jannesari, K. Bao, V. Pankratius, and W. F. Tichy, "Helgrind+: An efficient dynamic race detector," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, pp. 1–13, IEEE, 2009.

[54] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[55] J. Thalheim, P. Bhatotia, and C. Fetzer, "Inspector: data provenance using intel processor trace (pt)," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pp. 25–34, IEEE, 2016.

[56] A. Kleen and B. Strong, "Intel processor trace on linux," *Tracing Summit*, vol. 2015, 2015.

[57] J. W. Voung, R. Jhala, and S. Lerner, "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*, ESEC-FSE '07, (New York, NY, USA), pp. 205–214, ACM, 2007.

[58] W. Song, X. Qian, and J. Huang, "Ehbdroid: beyond gui testing for android applications," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 27–37, IEEE, 2017.

[59] C.-H. Hsiao, S. Narayanasamy, E. M. I. Khan, C. L. Pereira, and G. A. Pokam, "Asyncclock: Scalable inference of asynchronous event causality," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, (New York, NY, USA), p. 193–205, Association for Computing Machinery, 2017.

[60] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang, "Static data race detection for concurrent programs with asynchronous calls," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, (New York, NY, USA), p. 13–22, Association for Computing Machinery, 2009.

[61] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye, "Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, (New York, NY, USA), p. 162–180, Association for Computing Machinery, 2019.

[62] WALA, "T. j. watson libraries for analysis (wala)." http://wala.sourceforge.net/, 2017.

[63] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN*

conference on Object-Oriented Programing, Systems, Languages, and Applications, (New York, NY, USA), pp. 169–190, ACM Press, Oct. 2006.

[64] "Petablox benchmark." `https://github.com/petablox/petablox-bench`, 2018.

[65] "F-droid." `https://f-droid.org/`, 2020.