2020

Finding Data Races in Software Binaries with Symbolic Execution

Nathan D. Jackson
Wright State University

FINDING DATA RACES IN SOFTWARE BINARIES WITH SYMBOLIC EXECUTION

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

by

Nathan D. Jackson
B.S.C.S, Wright State University, 2015

2020
Wright State University

WRIGHT STATE UNIVERSITY

GRADUATE SCHOOL

04/30/20

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Nathan D. Jackson ENTITLED Finding Data Races in Software Binaries with Symbolic Execution BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science.

_____

Junjie Zhang, Ph.D.
Thesis Director

_____

Mateen Rizki, Ph.D.
Chair, Department of Computer Science and Engineering

Committee on Final Examination:

_____

Junjie Zhang, Ph.D.

_____

Meilin Liu, Ph.D.

_____

Michelle Cheatham, Ph.D.

_____

Barry Milligan, Ph.D.
Interim Dean of the Graduate School

ABSTRACT

Jackson, Nathan D. M.S., Department of Computer Science, Wright State University, 2020. Finding Data Races in Software Binaries with Symbolic Execution.

Modern software applications frequently make use of multithreading to utilize hardware resources better and promote application responsiveness. In these applications, threads share the program state, and synchronization mechanisms ensure proper ordering of accesses to the program state. When a developer fails to implement synchronization mechanisms, data races may occur. Finding data races in an automated way is an already challenging problem, but often impractical without source code or understanding how to execute the program under analysis. In this thesis, we propose a solution for finding data races on software binaries and present our prototype implementation *BINRELAY*. Our solution makes use of symbolic execution to maximize program coverage without requiring specific inputs to be passed to the binary. Currently, *BINRELAY* is limited to the detection of data races on global variables, but with future work, our system could detect data races on variables in other scopes.

Contents

List of Figures

List of Tables

Listings

Acknowledgment

I would like first to thank my wife, Heather, and my son Reid for supporting me through my master's program and primarily through my thesis research. Without their support, it would have been impractical for me to write this thesis and complete this phase of my education. I want to thank my parents for always believing in me and supporting my learning over the years. I would also like to thank my thesis advisor Dr. Junjie Zhang. I have learned a lot from Dr. Zhang, and I sincerely enjoyed our various prototyping sessions over the past year. Finally, I would like to thank Dr. Meilin Liu and Dr. Michelle Cheatham for taking the time to evaluate my thesis.

Abbreviations

CFG — Control Flow Graph

CPU — Central Processing Unit

CVE — Common Vulnerabilities and Exposures

CWE — Common Weakness Enumeration

DAG — Directed Acyclic Graph

ID — Identifier

JIT — Just In Time

NIST - National Institute of Standards and Technology

OS — Operating System

Introduction

We propose a solution for finding data races on global variables in software binaries. Our solution, *BINRELAY*, makes use of symbolic execution to exercise the code inside of a binary. During symbolic execution, we build a graph structure to model the interactions between simulated threads. We later use this model and other bookkeeping information to determine whether or not two memory accesses can happen concurrently and if there are proper synchronization mechanisms in place.

1.1 Motivation

Modern software programs frequently perform more than one task simultaneously to promote efficient use of hardware resources and to provide responsiveness using a technique called multithreaded programming. Since software developers must implement multithreaded programming, it is subject to programmer error. Data races are a common bug that can occur when two threads try to access a shared variable or memory location without a synchronization mechanism[1]. By their nature, data races are unpredictable and can have effects ranging from benign to fatal (in safety-critical systems)[2], so detecting them is of utmost importance.

While there are existing tools that can detect data races, they are generally limited in some fashion. These data race detection tools fall into either the dynamic or static analysis category. Dynamic tools can only analyze the executed code in the program under analysis. With dynamic data race detection approaches, code not executed is code not analyzed for

data races. Static data race detection tools, on the other hand, can analyze all of a program's code, but we found that existing static race tools require source code and are prone to false positives without filtering. Our desire is for a tool to find data races within a program without having to drive the program to specific bits of code and in cases where source code analysis is impossible or impractical.

1.2 Background

We utilize several technical concepts in this thesis. In this section, we describe these concepts.

1.2.1 Multithreaded Programming

In a multithreaded program, concurrent tasks run on different threads. Threads are an abstraction for units of CPU utilization that are managed by the operating system[3]. A process running in an operating system may have one or more threads. To utilize more processing power offered by modern hardware or promote application responsiveness, software developers create more than one thread in an application. However, multithreaded programming comes with its own set of challenges for developers. Developers implementing multithreaded applications must use synchronization mechanisms to ensure each thread does not interfere with another thread's execution or data shared amongst multiple threads.

1.2.2 Locks

One frequently used technique for synchronization is known as a mutex lock[3], or lock for short. Locks control access to a single resource shared amongst multiple threads. When using a lock, a thread takes the lock, performs the access on the shared resource, and then releases the lock. While the thread holds the lock, other threads may not access the protected shared resource. Other threads trying to access the shared resource must wait

until the lock is unlocked. Software developers must properly utilize the lock creation, acquisition, and release functions, and when they fail to do so, a data race may occur.

1.2.3 Data Races

Data races occur when one or more threads access a shared memory location without proper use of a lock or other synchronization mechanism[1]. Data races are a particularly dangerous type of software defect because they may go unnoticed under the right circumstances[4]. Adverse side effects of data races can have serious implications leading to reliability and security concerns [5][6].

Listing 1.1: Example Data Race

```
1  int x = 0;
2
3  void foo()
4  {
5      x += 1;
6  }
7
8  void bar()
9  {
10     x += 2;
11 }
12
13 int main()
14 {
15     thread t1 = create_thread(foo);
16     thread t2 = create_thread(bar);
17
18     join(t1);
19     join(t2);
20
21     puts(x); // what is the value of X?
22 }
```

In listing 1.1, we have an example program with a data race. The example creates two threads that add some value to the global variable X. If we follow this code, we would

expect X to be equal to 3. If we executed this code many times, the value might sometimes come out to be 1, 2, or 3 because the ”+=” operators in each thread constitute a read and write, are executed concurrently, and are subject to the OS scheduler, timing, or other factors since there is no locking mechanism in place to ensure only one thread may access the shared variable X at any given time.

1.2.4 Symbolic Execution

Symbolic execution is a technique where a program is analyzed to determine how an input affects the control flow of a program[7]. Usually, a program takes inputs (e.g., numbers, strings) from a file, command-line, user interaction, network, or other hardware devices; program inputs invoke specific paths in the program. Instead of using a concrete input from an external source, symbolic execution assigns symbolic values to the inputs and then executes the program in an interpreter. Since the inputs are now symbolic, the interpreter can execute both sides of a branch if they have a condition that relies on a symbolic input. Symbolic execution can cover more branches of a program because it does not need real input to reach various parts of the program [8].

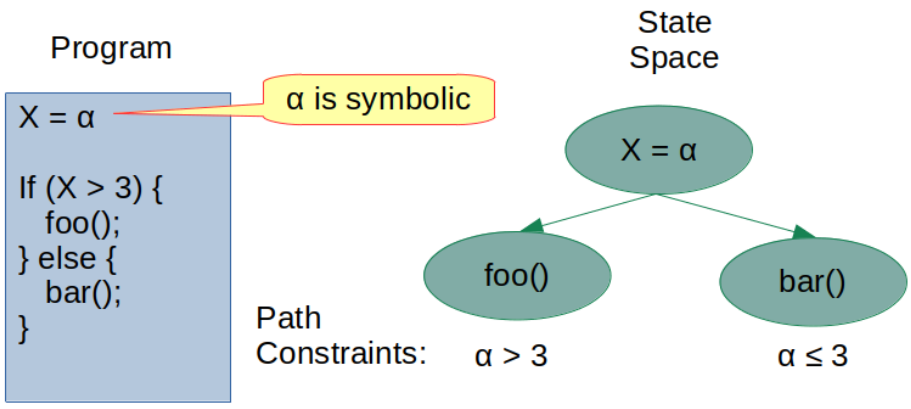

Figure 1.1: Symbolic Execution Example

In figure 1.1, we provide a very simple example to help explain symbolic execution. Before the symbolic execution starts, we create an initial state ready to start the program. In

the example program, we assign a symbolic variable α to X. Since X is now symbolic, the hypothetical symbolic execution engine executes both blocks in the if-else to produce two new states. For each state, the symbolic execution engine assigns different path constraints to α. The path constraint tells us what α must satisfy to reach the state. Constraint solvers can be used to generate solutions that satisfy a given state's constraints[8]. In this case, if we wanted to execute bar(), a constraint solver might return 3 as a possible solution for α.

A well-known problem with symbolic execution is path explosion. Path explosion is a difficult problem to overcome and is usually the limiting factor of many symbolic execution tools[8]. Path explosion occurs because the number of distinct paths taken inside of a program may grow exponentially. Each branch or conditional jump inside of a program introduces a new path followed by the interpreter. Over time, as paths accumulate, the memory of the host running the symbolic interpreter may be exhausted. Despite these limitations, symbolic execution can find software bugs, such as buffer overflows. We are hoping to use symbolic execution to find a different type of bug known as a data race.

1.3 Goals

In this thesis, we developed *BINRELAY*, a tool for discovering data races on global variables in userspace Linux binaries. A data race occurs when multiple threads access a memory location happen concurrently[3]. In this thesis, we focus on detecting races on global variables because all threads have direct access to global variables[9].

A race can only exist between two operations if they can run concurrently. Multi-threaded applications may launch threads that run concurrently or may wait for another thread to finish before moving on. Therefore, our system needs to be capable of determining if two operations ever run on different threads and which instructions in the binary may run concurrently.

A mutex lock can protect memory addresses accessed by more than one thread. If

the program uses the proper locking operations, concurrent memory accesses happen sequentially and thus eliminate the possibility of a data race. Our system needs to be able to determine which locks (if any) are held by threads.

Data races can be difficult to reproduce or notice, but we want to find them no matter how often they are manifest or regardless of how they affected a program's output. Therefore we should not rely on external factors for data race detection such as timing, user input, or system state.

1.4 Organization

Chapter 1 describes the motivation, background, and goals of this thesis. Chapter 2 describes work related to our system and where applicable, how the work differs from our approach. Chapter 3 describes the design of the system in this thesis. Chapter 4 describes our prototype implementation. Chapter 5 details our results and the limitations of our system and design and also identifies future work to improve our system. Finally, chapter 6 concludes this thesis.

Related Work

Academia, industry, and open-source projects have implemented race detection tools. In this section, we discuss a subset of existing approaches to data race detection and how they differ from our work, and the existing binary analysis used to prototype our system. In some of the work presented here, some authors may refer to race conditions instead of data races. In this section, we refer to race conditions and data races interchangeably.

Savage et al.[10] developed a dynamic data race detection tooled named *Eraser*. *Eraser* takes as input a binary to analyze and then produces a new instrumented binary that makes callbacks into the *Eraser* runtime to keep track of the data necessary for data race detection. Specifically, *Eraser* hooks thread management, lock acquisitions and releases, and memory accesses. With these hooks, *Eraser* keeps track of the locks held by each thread during their memory accesses. By checking which locks a thread holds during access, *Eraser* infers which locks protect shared variables. When access with no lock occurs on a shared variable, *Eraser* emits a warning. Since *Eraser* relies on tracking accesses from executing a binary, it is subject to the limitations of dynamic analysis: it can only report race errors from the executed code, not from code never reached.

In our approach, we do not modify the binary under analysis. However, *Eraser*'s approach to gathering the information needed to reason about the threads, locks, and memory accesses in a program is still valuable for our system. Our system similarly hooks the thread management, lock and unlock calls, and memory accesses, but in a symbolic execution environment instead of a modified binary.

A similar approach used in *Eraser* is implemented in the dynamic analysis framework called *Valgrind*[11]. *Valgrind* provides *Helgrind* which is a thread error detector that can detect data races[12]. A key difference between *Helgrind* and *Eraser* is that instead of producing a modified binary, an emulated environment provided by *Valgrind* hosts and instruments the binary. *Helgrind*'s race detection algorithm checks to see if there is a happens-before relation between two accesses and, if not, emits a warning. Our work is similar to *Helgrind*'s implementation in that we instrument the binary during symbolic execution. We do not make use of the happens-before relation; instead, we use a graph to reason about whether or not events can happen concurrently.

Engler et al. [13] developed *RacerX*, a static analysis tool that finds race conditions and deadlocks on large codebases. Our primary interest in *RacerX* in the race condition detection algorithm for this thesis. *RacerX* works by performing a depth-first traversal of a program's CFG. For each function, *RacerX* creates an empty lockset and then traverses the statements in the function. If a lock or unlock statement is encountered, then the lockset is updated accordingly.

After constructing the locksets, then the deadlock and race detection algorithms are run. The race detection algorithm runs on each statement and uses the locksets for each statement to determine if a race could occur. When operating in its most basic mode, *RacerX* looks for global variable accesses that are unprotected by a lock. In the more sophisticated approach, *RacerX* infers which non-global variables need to be protected and by which locks. While these checks are relatively simple, *RacerX* makes a few simplifying approximations against variables and function pointers that may lead to false positives. By ranking the race warnings, *RacerX* reduces the impact of false positives. Race warnings are assigned point values generated from heuristics. The point values enable ranking from most severe to least severe. However, *RacerX*'s approach could not be adapted as-is to work against software binaries. *RacerX* relies on information that can only reliably found in source code such as function call targets and source code annotations to tell the

system which functions are involved in taking locks. First, since there is no source code in the analysis we to perform, source code annotations cannot be applied to a binary-only program. Secondly, we found that our relatively simple test binaries make frequent use of indirect control flow instructions such as indirect branches or indirect subroutine calls. Indirect control flow often depends on values known only at runtime, such as the value of a register or memory location.

Blackshear et al. developed a static race condition detection system called *RacerD*[14]. *RacerD* constructs access paths to approximate the underlying memory location. Access paths are derived from source code and act as an alternative to pointer analysis. Instead of directly checking if two accesses are accessing the same memory, *RacerD* considers two syntactically similar access paths as a reference to the same memory. The authors found that this approach was enough to find real race conditions, but false negatives could occur due to pointer aliasing. *RacerD*'s approach is interesting because the use of access paths generated from source code enables interprocedural analysis without relying on dynamic analysis techniques or symbolic execution. Unfortunately, access paths are a source code-centric approach that obviously cannot apply with only software binaries.

RELAY by Voung et al. is another approach to race condition detection; it generates per-function summaries that contain relative locksets and guarded accesses[15]. Relative locksets are two sets of locks that show which locks were taken or released since the start of a function. Guarded accesses are tuples of an accessed variable, whether or not the access was a read or write, and the locks held at the time of access. *RELAY* uses symbolic execution before computing relative locksets and the guarded access sets, to ensure that the per-function data best represents the calling context of functions. After symbolic execution and lockset computation, *RELAY* determines if a race condition may occur by performing intersections of the locks between the guarded accesses if they refer to the same underlying memory. If the lockset resulting from the intersection is empty, then a warning is emitted. The authors tested *RELAY* against the Linux kernel source code to find both true and false

positives. *RELAY* inspired our work, but we focus on userspace binaries instead of source code.

Another paper by John Mellor-Crummey presented a dynamic data race detection technique that makes use of a fork-join graph. Fork-join graphs are DAGs used to model thread creation and join operations in a program[4]. We use a similar approach to determine if two accesses run concurrently, but we generate the DAG structure using symbolic execution instead of dynamic analysis.

We found one other work by Andreas Ibing that also finds data races with the help of symbolic execution, but source code is required[16]. In this approach, the author uses symbolic execution to explore the program before switching to concrete execution and using ThreadSanitizer to perform the data race detection. Our system assumes no source code is available and performs symbolic execution from the outset. Ibing tested his approach against the race condition test binaries from the Juliet dataset and successfully found all races. We also tested our system against the Juliet dataset and produced comparable results.

In this thesis, symbolic execution and binary analysis techniques are a means to an end. We built our system using *angr*, which was created by Shoshitaishvili et al.[8]. *angr* is a binary analysis framework that implements many typical analysis techniques in Python. *angr*'s well-designed framework provides for natural extension and re-use of components, as seen in the design section of this thesis. There are various other components included with *angr* that enable binary loading and lifting to an IR that our system implicitly uses, but are not the focus of our work.

Design

Next, we describe the overall design of our system and the challenges we had to overcome to detect data races in software binaries.

3.1 Problem Formulation

Data races occur when two memory accesses occur on the same underlying memory location, at least one of the accesses is a write, and the accesses are unsynchronized[4]. Multithreaded programs share global variables, so we focused on finding data races on global variables first[9]. Since data races like any other bug can exist in parts of the code that require specific inputs to execute, we wanted to find data races without requiring upfront knowledge about what input to supply to the binary. Therefore, we designed our system *BINRELAY* to find data races on the global variables in software binaries without requiring a user to have upfront knowledge about how to run the program.

3.2 Symbolic Execution

One goal for our system was to find data races in all parts of the program binary under analysis regardless without requiring upfront knowledge about how to execute the program. To this end, we used symbolic execution to implement *BINRELAY*. When a computer executes a program, there is one path that the program follows based on what inputs the program received. Symbolic execution, on the other hand, assigns symbolic

variables to inputs and then executes the program. For conditional branches that depend on symbolic input, a symbolic execution engine executes both sides of the branch[8].

The challenge for us was determining which parts of a binary run concurrently while executing the binary in a symbolic execution engine. Ordinarily, threads are started by the parent thread by invoking a thread creation routine and supplying as a parameter, the function to be executed by the new thread. The parent thread will then typically wait for the thread it created to complete. These thread creation and wait operations ultimately invoke the OS. During symbolic execution, we are simulating the binary, so we cannot defer to the operating system's implementation. To provide an approximation of what a thread does to the simulated state, we intercept the thread creation functions.

To simulate threads during our symbolic execution, we keep track of some additional bookkeeping information in each state managed by the symbolic execution engine. The bookkeeping information keeps track of the current, previous, and next simulated thread ID and a directed graph that models the active threads at different points throughout the state's history. A symbolic execution engine updates the bookkeeping information at specific points during the simulation.

We maintain a lockset on each state. To keep track of the locks, we create an empty lockset in the initial state. When taking a lock, the lock's address gets added to the state's lockset. When releasing a lock, the lock's address gets removed from the state's lockset.

Finally, we record each memory access during symbolic execution. We add memory accesses to an access set maintained in the current state. A tuple of the following values represents memory accesses: the current simulated thread id, current program counter value, whether the access was a read or write, the lockset at time of access, and the current node representing the active simulated threads from the directed thread state history graph.

3.3 Fork-Join Graph

We make use of a DAG structure to model the threads that may be active at a given point in time during normal execution of the program under analysis. The DAG structure is called a fork-join graph. Each node is a set of thread IDs that ran concurrently. The graph's edges represent the events that created a new node, such as a fork (thread creation) or join[4]. Each simulation state has a fork-join graph. When the symbolic execution engine encounters indirect control flow, it creates a copy of the fork-join for each new resulting state. States also store references to the last node added to the graph. The last added node contains the threads that are known to be executing concurrently in the current state.

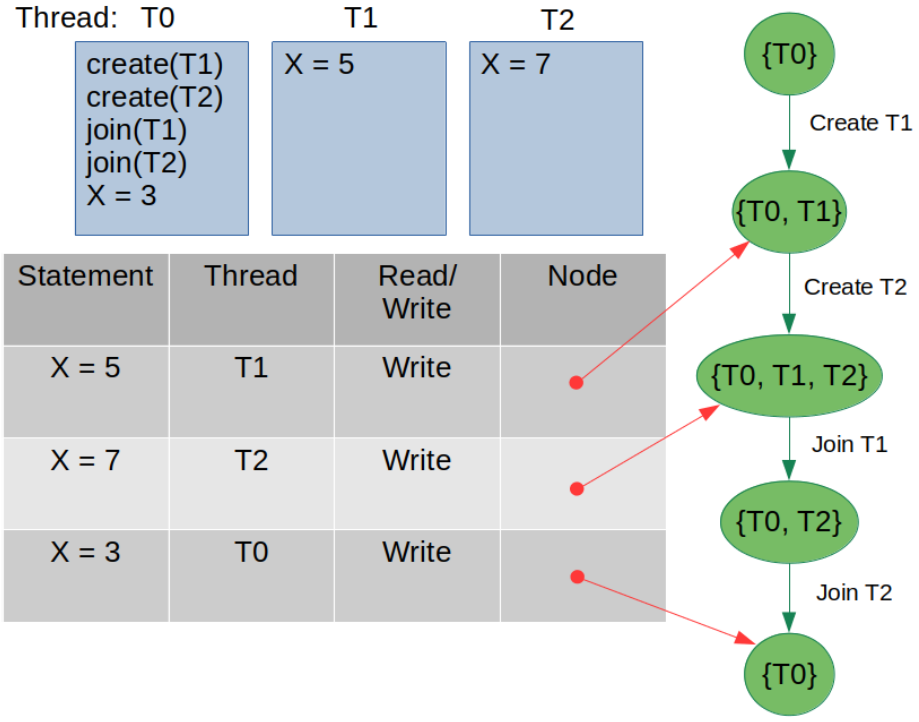

Figure 3.1: Fork-Join DAG Example

Building the fork-join graph is a core part of our approach to detecting data races because, with it, we can determine if two events can happen concurrently. Events of interest (memory accesses for data race detection) can take a reference to the node last added to the fork-join graph. For each pair of events that we'd like to check for concurrency, we first

find the edge in the graph that created the event's thread. Using this edge's destination node and the other event's recorded node in the fork-join graph, we check to see if there is a path between the two nodes and if there is, that there is no join edge on the first event's thread ID. If both of these checks pass, then the events can happen concurrently. If the first event's thread creation edge was not found (as in the case of the main thread) or the path check fails, then the events are swapped, and we rerun the checks. If both cases fail, then the events do not happen concurrently. Note that the structure of the fork-join graph stored within a state assumes a list-like structure.

Figure 3.1 gives an example fork-join graph. Initially, the graph starts with one node. When the example program creates threads 1 and 2, new nodes are created and connected by the annotated create edges. When executing the program outside of the symbolic execution environment, the thread would still run concurrently with the parent or other threads, so we do not update the graph again until we see a join operation. When the parent thread joins on threads 1 and 2, we also add annotated join edges. Using this graph structure, we can determine if the two statements may run concurrently. For example, we can see the statements X =5 and X = 7 may be executed in parallel because their nodes referenced in the table have a path without a join on thread 1 or 2. However, X = 3 and X = 5 cannot run concurrently because of the join edge on thread 1. Listing 3.1 contains the pseudocode for the concurrency check algorithm.

Listing 3.1: Concurrency Check Pseudocode

```
1 Find-Create-Edge(G, t)
2     For edge in G.edges
3         If edge.type == create and edge.tid == t:
4             Return edge
5
6 Reachable(c, t, a)
7     while c != null
```

```
        if c == a.threadset_node
            if t in a.threadset_node
                Return true
            Else
                Return false
        Else
            E = c.next_edge
            If E.attrib == join(t)
                Return false
            Else
                C = E.dest_node

Check-For-Concurrency(a1, a2)
    Edge1 = null
    Edge2 = null

    If a1.tid not T0
        Edge1 = Find-Create-Edge(a1.tid)
    If a2.tid not T0
        Edge2 = Find-Create-Edge(a2.tid)

    If edge2 != null and Reachable(Edge2.dest_node, a2.tid, a1)
        Return true
    If edge1 != null and Reachable(Edge1.dest_node, a1.tid, a2)
        Return true
    Return false
```

3.4 Event Instrumentation

To collect information from symbolic execution, we instrument certain events in the binary. When the symbolic execution engine executes the events, it runs the associated

callback code. In *BINRELAY*, we are concerned with several events: thread creation and join function calls, lock acquisition and release function calls, and memory accesses.

When the program under analysis calls a thread creation function, we keep track of the fact that we are going to be simulating a new thread, and since there is no operating system to handle the thread creation, we simulate this process. To implement the thread creation simulation, we update bookkeeping variables. We set previous thread ID to the current thread ID, current thread ID to the next thread ID, and then increment the next thread ID. Since we increment the next thread ID on every thread creation call, each simulated thread gets a unique ID within a simulation state. We also update the fork-join graph by inserting a new node. Recall that nodes in the fork-join graph hold sets of active thread IDs, so to construct a new node, we create a new set with all of the thread IDs from the last inserted node's set and add the new thread ID. We then create an edge between the last inserted node and the new node. The new edge has an annotation that indicates the edge exists as a result of thread creation. Once we've updated all of the bookkeeping information and the fork-join graph, we call the function passed into the thread creation routine by the binary. When this function returns, we restore the current thread ID with the previous thread ID and continue execution after the thread creation call.

Thread join operations are simpler to model and only require us to update the fork-join graph. We also create a new node in the fork-join graph for joins. The new node's set as a result of the join operation has all of the thread IDs from the last insert node except for the joined thread ID. We create an edge between the last inserted node and the new node. The new edge has an annotation that indicates the edge exists as a result of the thread join operation. If the program under analysis does not contain a join operation for a thread, the thread is assumed to run forever starting from its creation but this is not a problem for our system. The fork-join graph would show that the thread could run concurrently with any code executed after the creation call and the fork-join graph would reflect this.

Lock acquisition and release events only require updating the locksets on each sim-

ulated state. When taking a lock, the address of the lock gets stored in the current state's lockset. When the lock is released, the lock's address gets removed from the current state's lockset.

We also instrument memory access events. When memory access occurs, an access tuple is created that contains the following items: current simulated thread id, current simulated program counter, the accessed address, whether the access is a read or write, the locks held during access, and a reference to the last inserted node in the access graph. The access tuple gets added to the current simulation state's access set. The fields part of an access tuple are primarily a snapshot at the point of access of the data collected during other instrumentation callbacks, as described in previous sections.

3.5 Warning Generation

Once the symbolic execution completes, we can examine all of the resulting states. Each state has an access set and a fork-join graph used to find any data races. Since threads are simulated by calling the threads' entry functions and updating the bookkeeping information, we can look for races between memory accesses within states, and we don't have to consider comparing accesses across different states. Within the state, we take every combination of two access tuples and use a set of rules to determine if the accesses were involved in a data race. Each rule must pass to emit a data race warning.

The first rule is that accesses must occur on different threads. Secondly, at least one of the two accesses must modify the memory. Third, there must be no lock addresses in the intersection of both accesses' locksets. Finally, we check to see if the accesses can run concurrently by using the reachability algorithm described in the fork-join graph section and treat the accesses as events. Assuming each of these checks passed, we emit a warning that there is a possible data race between the two memory accesses.

Since the symbolic execution produces multiple resulting states that may have data

races, more than one state may have encountered the same data race. To reduce the amount of output to something more easily interpreted by a human, we only report unique races, which can be identified by the program counter values of the conflicting memory accesses.

Implementation

We present a proof-of-concept implementation to validate our approach to finding data races. To this end, we describe the implementation of *BINRELAY*, a tool for finding data races in software binaries. *BINRELAY* accepts a userspace Linux binary as input and produces warnings for any data races found during the analysis. With *BINRELAY*'s output, a user may be able to fix the program if they have access to the source code or the ability to patch the program binary or report the issue to the software vendor.

4.1 Binary Analysis Framework

Since our focus is data race detection, we chose to implement our system on top of an existing framework that enables the symbolic execution of software binaries. We chose the *angr* binary analysis framework because it provides not only the symbolic execution framework, but it is also modular and provides other functionality such as binary loading that our system implicitly requires. Since we used *angr*, we wrote *BINRELAY* in Python.

There are performance-related downsides to using Python and *angr* by extension. Python suffers from a problem known as the global interpreter lock, which makes actual parallel programming in Python difficult[17]. The global interpreter lock only allows one thread to run at any given time. Practically this means that parallelizing our system would be difficult without significant effort. Furthermore, Python is an interpreted language, and this means the performance of the interpreter limits our system. Despite the performance issues, we think that the ease of use of Python and modularity of angr far outweigh the

performance issues for prototyping and evaluation of the core algorithm.

4.2 Symbolic Execution

Our system makes use of *angr*'s symbolic execution engine. Since symbolic execution is a means to an end for *BINRELAY*, we will only briefly describe how *angr*'s symbolic execution capabilities work to give background and context for our implementation. At the core of *angr*'s symbolic execution is the SimState class[18]. SimState represents the entire state of a program, such as memory and registers. During symbolic execution, the step method advances a state. When a state is advanced, the symbolic execution engine executes the basic block that is referenced by the state. The result of advancing the state is a collection of successor states that are part of a SimSuccessors object. States within the SimSuccessor object may be satisfied or unsatisfied. Satisfied states were states whose constraints were satisfied. Unsatisfied states are states that were not satisfied. With symbolic execution, states may have more than one satisfiable successor states, which means that more than one successor state also needs to be followed.

Since stepping a state may result in more than one successor state, managing these successor states would be cumbersome if not for *angr*'s SimulationManager class. A SimulationManager object keeps track of the states and is also responsible for advancing the states. In our work, we rely entirely on a SimulationManager object to perform symbolic execution, and we never directly step states. A SimulationManager keeps states in lists called stashes. Generally speaking, the simulation manager steps an entire stash and then moves the states between stashes depending on certain conditions. For example, the two stashes that we are most interested in are the ”active” stash and the ”deadended” stash. The ”active” stash contains the states we are actively advancing. The ”deadended” stash contains the states that resulted in the program under analysis terminating. When a state terminates, it gets moved from the ”active” stash to the ”deadended” stash. For *BINRELAY*,

we analyze states that ended up in the deadended stash, or other words, states where the program ran until it terminated.

For our symbolic execution, we construct an initial state such that we are ready to start executing from the program's entry point. Since we would like to assume as little as possible about the binary under analysis, we construct six symbolic command-line arguments and pass them as input into the program under analysis. Using symbolic arguments allows the symbolic execution to consider paths that require specific command-line arguments such as a flag or file input.

4.3 Bookkeeping

As mentioned earlier, *angr* is extremely modular. One of the key areas where this modularity shines is augmenting SimState objects with extra information using plugins. SimStatePlugins can be registered with a state and then propagated to the successor states during the normal state stepping process. In *BINRELAY*, we need to keep track of our bookkeeping data described in Chapter 2. To do this, we created a SimStatePlugin named ThreadInfo. The ThreadInfo plugin keeps track of the fork-join graph, the last node added to the fork-join graph, the lockset containing locks that the state currently holds, the access set, and previous, current, and next thread IDs. To update the bookkeeping data, we use hooks for various events during the symbolic execution. As the symbolic execution engine generates successor states, the state plugin's copy method copies data into new instances of the plugin.

We implemented the fork-join graph with the NetworkX Python library because it is already a dependency for *angr*. To find a path between two nodes in the fork-join graph during the concurrency check described in Chapter 2, we use the NetworkX's shortest path method. If a path exists, then we verify that none of the edges the path represents a join operation on one of the threads.

Listing 4.1: ThreadInfo SimState Plugin

```
class ThreadInfoPlugin(angr.SimStatePlugin):
    def __init__(self):
        super(ThreadInfoPlugin, self).__init__()

        self.prev_thread_id = None
        self.current_thread_id = 0
        self.next_thread_id = 1

        self.TG = nx.DiGraph()
        self.cn = frozenset([0])
        self.TG.add_node(self.cn)

        self.locks_held = set()

        self.accesses = {}

    @angr.SimStatePlugin.memo
    def copy(self, memo):
        result = ThreadInfoPlugin()
        result.prev_thread_id = self.prev_thread_id
        result.current_thread_id = self.current_thread_id
        result.next_thread_id = self.next_thread_id

        result.TG = copy.deepcopy(self.TG)
        result.cn = copy.deepcopy(self.cn)

        result.locks_held = copy.deepcopy(self.locks_held)

        result.accesses = copy.deepcopy(self.accesses)
        return result
```

4.4 Function and Event Hooks

To collect the required bookkeeping information stored in the ThreadInfo plugin, we use *angr*'s ability to hook symbols and insert breakpoints into symbolic execution. To hook functions with *angr*, one creates a SimProcedure class and then hooks a symbol to an instance of the SimProcedure. Registering the SimProcedure has the effect of replacing a function call with a call to the SimProcedure's run method. Inside the run method, the SimProcedure can change control flow or modify the current SimState. In our implementation, we are focused primarily on Linux binaries, so we targeted the thread creation and mutex functions that are part of the POSIX thread library: pthread_create, pthread_mutex_lock, and pthread_mutex_unlock[9].

To simulate thread creation, we implemented a SimProcedure that replaces the pthread_create function. *angr* includes a pthread_create SimProcedure, but it simulates thread creation by creating two successor states, a state that started the thread and a state that remained on the parent thread. Unfortunately, we could not use this SimProcedure as-is, and we had to replace it with our implementation to maintain the bookkeeping data. Details of the thread creation simulation are in Chapter 2, so we do not detail that here.

Listing 4.2: pthread_create Hook

```
1 class _pthread_create(angr.SimProcedure):
2     """
3     A Sim Procedure for pthread_create
4     """
5
6     def run(self, nt, attr, start_routine, arg):
7         thread = self.state.solver.eval(nt)
8
9         self.state.thread_info.prev_thread_id = ...
              self.state.thread_info.current_thread_id
```

```
10          self.state.thread_info.current_thread_id = ...
                self.state.thread_info.next_thread_id
11          self.state.thread_info.next_thread_id += 1
12
13          logger.debug("enter thread: %d -> %d",
14                       self.state.thread_info.prev_thread_id,
15                       self.state.thread_info.current_thread_id)
16
17          src_node = self.state.thread_info.cn
18
19          tmp = set(src_node)
20          tmp.add(self.state.thread_info.current_thread_id)
21          dest_node = frozenset(tmp)
22
23          self.state.thread_info.TG.add_edge(src_node, dest_node,
24              create=self.state.thread_info.current_thread_id)
25          self.state.thread_info.cn = dest_node
26
27          self.state.mem[thread].uint64_t = ...
                self.state.thread_info.current_thread_id
28          self.call(start_routine, (arg,), 'on_return')
29
30      def on_return(self, thread, attr, start_routine, arg):
31          prev = self.state.thread_info.current_thread_id
32          self.state.thread_info.current_thread_id = ...
                self.state.thread_info.prev_thread_id
33          self.state.thread_info.prev_thread_id = prev
34
35          logger.debug("leave thread: %d -> %d",
36                       self.state.thread_info.prev_thread_id,
37                       self.state.thread_info.current_thread_id)
38          self.ret(self.state.solver.BVV(0, self.state.arch.bits))
```

Thread join operations are also implemented as SimProcedures and replace the pthread_join function. Chapter 2 details how our system simulates the join operation.

Listing 4.3: pthread_join Hook

```
class _pthread_join(angr.SimProcedure):
    def run(self, thread, retval):
        joined_id = self.state.solver.eval(thread.to_claripy())
        logger.debug("Join %d", joined_id)

        src_node = self.state.thread_info.cn
        tmp = set(src_node)
        tmp.remove(joined_id)
        dest_node = frozenset(tmp)

        self.state.thread_info.TG.add_edge(src_node, dest_node, ...
            join=joined_id)
        self.state.thread_info.cn = dest_node
```

SimProcedure objects simulate lock acquisition and release calls for the pthread_mutex_lock and pthread_mutex_unlock functions. In pthread_mutex_lock's SimProcedure, we add the address of the lock passed to the SimProcedure to the locks_held set in the ThreadInfo plugin on the current state. During pthread_mutex_unlock calls, we remove the address of the lock from the locks_held set in the ThreadInfo plugin.

Listing 4.4: Lock and Unlock SimProcedure Implementation

```
class _pthread_mutex_lock(angr.SimProcedure):
    """
    A simprocedure that is executed when a lock (mutex) is taken.
    """

```

```
6       def run(self, mutex):
7           logger.debug("Thread %d is locking mutex @ %s" %
8                        (self.state.thread_info.current_thread_id, ...
                            mutex))
9           mutex_address = self.state.solver.eval(mutex.to_claripy())
10          self.state.thread_info.locks_held.add(mutex_address)
11
12
13  class _pthread_mutex_unlock(angr.SimProcedure):
14      """
15      A simprocedure that is executed when a lock (mutex) is released.
16      """
17
18      def run(self, mutex):
19          logger.debug("Thread %d is releasing mutex @ %s" %
20                       (self.state.thread_info.current_thread_id, ...
                            mutex))
21          mutex_address = self.state.solver.eval(mutex.to_claripy())
22          self.state.thread_info.locks_held.remove(mutex_address)
```

Finally, we also hook the memory accesses that happen during symbolic execution. In contrast to the thread and lock functions, we do not implement memory access hooks using SimProcedures. Instead, we can define breakpoints in the symbolic execution engine for memory reads and writes that invoke callback functions. Callback functions for memory accesses accept the current state as an argument, and the details related to the memory access are available as part of the angr SimInspector state plugin. Inside our callback functions, we construct an access tuple that contains the current simulated thread ID in the ThreadInfo plugin, the current program counter value, the access address, whether the read or write access, a snapshot of the locks currently held by our state, and the last added node to the fork-join graph.

Listing 4.5: Memory Access Callbacks

```
def _mem_read_callback(state):
    ip = state.ip
    from_addr = state.inspect.mem_read_address
    length = state.inspect.mem_read_length
    logger.debug("Thread %d is reading from %s at %s" %
                 (state.thread_info.current_thread_id, from_addr, ...
                    state.ip))

    if int != type(from_addr):
        from_addr = state.solver.eval(from_addr)
    if int != type(length):
        length = state.solver.eval(length)
    if int != type(ip):
        ip = state.solver.eval(ip)

    for i in range(length):
        addr = from_addr + i
        access = (state.thread_info.current_thread_id, ip, addr, ...
            READ,
                  frozenset(state.thread_info.locks_held), ...
                     state.thread_info.cn)
        if addr not in state.thread_info.accesses:
            state.thread_info.accesses[addr] = set()
        state.thread_info.accesses[addr].add(access)

    logger.debug("thread=%d pc=0x%X addr=0x%X rw=r locks=%s tsn=%s",
                 state.thread_info.current_thread_id, ip, from_addr,
                 state.thread_info.locks_held, state.thread_info.cn)


def _mem_write_callback(state):
```

```
29      ip = state.ip
30      to_addr = state.inspect.mem_write_address
31      length = state.inspect.mem_write_length
32      logger.debug("Thread %d is reading from %s at %s" %
33                   (state.thread_info.current_thread_id, to_addr, ...
                        state.ip))
34
35      if int != type(to_addr):
36          to_addr = state.solver.eval(to_addr)
37      if int != type(length):
38          length = state.solver.eval(length)
39      if int != type(ip):
40          ip = state.solver.eval(ip)
41
42      for i in range(length):
43          addr = to_addr + i
44          access = (state.thread_info.current_thread_id, ip, addr, ...
                WRITE,
45                    frozenset(state.thread_info.locks_held), ...
                        state.thread_info.cn)
46          if addr not in state.thread_info.accesses:
47              state.thread_info.accesses[addr] = set()
48          state.thread_info.accesses[addr].add(access)
49
50      logger.debug("thread=%d pc=0x%X addr=0x%X rw=w locks=%s tsn=%s",
51                   state.thread_info.current_thread_id, ip, to_addr,
52                   state.thread_info.locks_held, state.thread_info.cn)
```

4.5 Race Warning Emission

Once symbolic execution has completed, we need to find the resulting data races, if any using the bookkeeping data that we stored in the ThreadInfo state plugin. We focused on only terminated SimStates (states moved to the deadended stash in the SimulationManager object) as states can only be active, terminated, or stopped due to an error with the default SimulationManager. We may miss races that occured on a state that encountered an error during symbolic execution, but we think that states that terminated are a better approximation of how the program would behave outside symbolic execution. For each of the terminated states, we generated all combinations of two memory access tuples recorded in the ThreadInfo state plugin. For each combination, we then evaluated our simple rules: the thread IDs for each access must be different, at least one access had to be writing memory, the intersection of the locksets on each access must be empty, and the referenced nodes in the fork-join graph must be reachable according to the algorithm described in Chapter 2. If all of these rules pass, we then emit a race warning between the two accesses.

Listing 4.6: Sample 6 Program Output

```
1  ...
2  Starting symbolic execution
3  Symbolic execution terminated
4  <SimulationManager with 2 deadended>
5  Checking for race conditions
6  possible race: (0x401184, 0x40123A)
7  possible race: (0x40118D, 0x40123A)
8  possible race: (0x40118D, 0x401231)
9  Race Analysis Complete
10 ...
```

Evaluation

We evaluated *BINRELAY* against a set of x86-64 userspace Linux binaries. We compared our results against the *Helgrind* thread error detector tool.

5.1 Dataset

To test the functionality of our system, we developed a set of sample binaries in C. Each sample binary has zero or more data races and branches that depend on command-line arguments. We placed the shared variables in the global scope of each test program so that the underlying memory falls in the ".data" section of the binary.

We also tested our system against binaries built from the NIST Juliet Test Cases for C/C++[19]. The programs in the Juliet dataset contain various bugs categorized into CWEs. CWEs are categories of common weaknesses in software and hardware that have security ramifications [20]. Specifically, we used the binaries that have bugs categorized under "CWE-366: Race Condition within a Thread". Each of the Juliet binaries has a slightly different CFG, but have at least one data race. We tested a total of 32 Juliet binaries. Half of the CWE-366 binaries have data races on global variables, and the other half have data races on variables in memory allocated from the stack. While we were targeting global variables specifically, we performed another test where we removed the filter for memory accesses to anywhere but the .data and .bss sections to see how our algorithm would work as it currently is.

Once we validated our algorithm using our sample binaries and the Juliet binaries, we

attempted to use our system against non-trivial binaries that have known data races. To find software with known data races, we tried to use the CVE database, but many reported data race bugs required a specific system state to be triggered or private bug trackers hide the details of the race from public view. Since finding a source of binaries with confirmed data races was challenging, we attempted to modify the "genisoimage" binary from Ubuntu 18.04, but we quickly found limitations in our system that prevented us from analyzing the patched binary.

5.2 Effectiveness

Unfortunately, evaluating our system against a non-trivial binary has been unsuccessful. We very quickly encountered either path explosion or cases where one thread was waiting on another thread to modify the global memory in a certain way and would otherwise stay in an infinite loop in the symbolic execution environment. We discuss these limitations in greater detail in Chapter 6.

Despite the problems we encountered while testing against real software binaries, we were successful in testing our system against the global variable data races in the Juliet Dataset. In the first test with the global variable filter in place, we found all of the data races in the Juliet binaries if the variable on which a data race occurred was globally scoped. We had no false positives in either the global variable or stack variable binaries. As expected, we missed the data races where the shared variable was passed into the thread by reference and allocated from the stack. In the test where we removed the global variable filter, we identified all of the data races but had many false positives. After analysis of the false positives, we found that our algorithm was identifying these data races because of a deficiency in modeling a thread's execution, but our rules to determine if the system should emit a race warning are sound. We discuss the modeling deficiency in Chapter 6.

All of the Juliet CWE-366 binaries followed a similar pattern where each thread ex-

Table 5.1: Juliet CWE-366 Evaluation Results

	Expected Races	True Positive	False Positive
BINRELAY	72	36	0
BINRELAY (Non-Global Filter Disabled)	72	72	2448
Helgrind	72	72	0

ecuted a loop for one million iterations. On each iteration, a read and write to the same shared variable occurred. While we were able to detect the data races on the full execution, our tool spent much time simulating each iteration when we had already collected the required bookkeeping data after the first iteration. We were able to implement a shortcut for these binaries to reduce the overall runtime to just a few seconds. For our analysis, there is not much value in executing every iteration of the loop, assuming we've reached each instruction inside the loop. For Juliet binaries, the loop bodies are simple and have no additional branch conditions. Thus, once we executed the loop once with symbolic execution, we had already captured the needed information regarding memory accesses and could force the symbolic execution engine to jump out of the loop and continue execution.

As expected with the sample binaries, we identified all data races with no false positives with the non-global filter in place since our system was heavily tested and designed against these samples. However, we were able to compare the effectiveness of our system compared to *Helgrind*. Both tools correctly identify the data races, but with *Helgrind*, there is a caveat. For *Helgrind* to detect a data race, the data race must occur during execution. Some of our sample binaries accept command-line arguments. If we don't pass enough arguments or the incorrect values are passed into the binary, *Helgrind* won't detect the data race. Since we use symbolic execution, we can detect the data races without supplying specific inputs to the binary. *Helgrind* requires the command-line arguments to be known ahead of time. Refer to Sample 6 in Appendix A for an example.

Table 5.2: Sample Binary Evaluation Results

	Expected Races	True Positive	False Positive
BINRELAY	12	12	0
BINRELAY (Non-Global Filter Disabled)	12	12	33
Helgrind	12	9	0
Helgrind (No Arguments)	12	5	0

Discussion

Given that *BINRELAY* can successfully find data races on binaries from our dataset, we believe that the core of our algorithm is sound for data race detection. Unfortunately, we had trouble finding a significant quantity of non-trivial programs because the details of confirmed data races were challenging to obtain since the details of acknowledged CVEs were either difficult to obtain or were already patched in the publicly available binaries. To overcome this, we attempted to modify the ”genisoimage” binary from Ubuntu 18.04 to create an artificial data race, but this binary revealed a limitation with our current system. With our current implementation, we have two challenges left for potential future work.

First, we need to improve the modeling of multithreaded applications within a symbolic execution environment. Currently, our system produces false positives when the non-global variable filter is disabled. In the Juliet binaries, where the shared variable is on the stack, our system correctly identifies real data races on stack variables but also produces false positives. After analyzing the false positives, we found that the reason for false positives is because our simulated threads are sharing a stack. When the thread finishes execution, we return to the parent thread. When creating new threads or making subsequent function calls on the parent thread, the program under analysis reuses stack addresses. The reuse of stack addresses in this way causes *BINRELAY* to record memory access tuples that do not accurately model what would happen during a real execution of the binary. In a real execution of the program, the threads would not share a stack. Instead, the operating system would assign each thread a separate stack. In this way, most memory accesses to the

stack would be on different memory addresses, except when the program explicitly passes stack variables between threads. To remove these false positives, we should assign each simulated thread to a different region of memory for the stack.

Secondly, we found that our system does not handle cases where a thread is waiting for an event to occur on another thread. In the ”genisoimage” test, we were able to execute the part of the binary with the artificial data race, but it ended up in an infinite loop because it was waiting on another thread to change a value used in a loop condition. Since *BINRELAY* simulates threads by running them sequentially, the change to the value that the loop encountered never happened, and the loop would run forever. A possible approach to overcome this limitation is to assign symbolic variables to the memory locations used in the loop condition to allow the symbolic execution engine to find a successor state that exits the loop but still allows us to record the memory accesses in the loop body.

Conclusion

In this thesis, we have described our data race detection tool: *BINRELAY*. We demonstrated that our system could find data races in our sample binaries and the NIST Juliet dataset. While our initial goal was to detect data races on global variables, we believe we've shown that with additional work, our system could detect races on non-global variables without false positives. We've also identified future work to improve the symbolic execution to handle cases where simulated threads are waiting for an event to occur on another thread or external event. With these improvements, we believe that *BINRELAY* could find data races in non-trivial binaries.

Bibliography

[1] John Regehr. Race Condition vs. Data Race. https://blog.regehr.org/archives/490, 2011.

[2] Nancy G. Leveson and Clark S. Turner. An Investigation of the Therac-25 Accidents. *Computer*, 1993.

[3] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., ninth edition, 2013.

[4] John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, page 24–33, New York, NY, USA, 1991. Association for Computing Machinery.

[5] CVE-2019-5796. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5796.

[6] CVE-2018-12633. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-12633.

[7] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 1976.

[8] Giovanni Shoshitaishvili, Yan and Wang, Ruoyu and Salls, Christopher and Stephens, Nick and Polino, Mario and Dutcher, Audrey and Grosen, John and Feng, Siji and Hauser, Christophe and Kruegel, Christopher and Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[9] pthreads(7) - Linux man page. https://linux.die.net/man/7/pthreads.

[10] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. *Operating Systems Review (ACM)*, 31(5):27–37, 1997.

[11] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, 42(6):89–100, 2007.

[12] Helgrind: A Thread Error Detector.

[13] Dawson Engler and Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. *ACM SIGOPS Operating Systems Review*, 2003.

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

[15] Jw Voung, R Jhala, and S Lerner. RELAY: static race detection on millions of lines of code. *ESEC/FSE '07 Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007.

[16] Andreas Ibing. Efficient data-race detection with dynamic symbolic execution. In *IEEE Software Engineering Workshop*, September 2016.

[17] Global Interpreter Lock. https://wiki.python.org/moin/GlobalInterpreterLock.

[18] angr API Documentation. https://angr.io/api-doc/.

[19] https://samate.nist.gov/SARD/testsuite.php.

[20] About CWE. https://cwe.mitre.org/about/index.html.

Appendix A

Program Code

Listing A.1: find_races.py

```
#!/usr/bin/env python3

import argparse
import logging

import angr
import claripy

import binrelay

logger = logging.getLogger(name=__name__)
logger.setLevel(logging.INFO)

MAX_ARG_BYTES = 20

if __name__ == "__main__":
    ap = argparse.ArgumentParser(
        description="Find race conditions on a given binary.")
    ap.add_argument("binary", help="the binary to analyze")
    ap.add_argument("-d", "--disable-filter", action="store_true",
                    default=False, help="Disable global variable ...
                       filter (experimental)")
    ap.add_argument("-u", "--unicorn", action="store_true",
                    default=False, help="Use the unicorn engine")
    ap.add_argument("-l", "--loop-hooks", action="store_true",
                    default=False, help="Use loop hooks")
    args = ap.parse_args()

    logger.info("Finding races in %s" % (args.binary))

    project = angr.Project(args.binary, auto_load_libs=False)

    binrelay.utils.hook_pthread_exit(project)
```

```
    if True == args.loop_hooks:
        binrelay.utils.hook_loops(project)

    arg_size = project.arch.byte_width * MAX_ARG_BYTES

    analysis_args = [
        project.filename,
        claripy.BVS("arg1", arg_size),
        claripy.BVS("arg2", arg_size),
        claripy.BVS("arg3", arg_size),
        claripy.BVS("arg4", arg_size),
        claripy.BVS("arg5", arg_size),
        claripy.BVS("arg6", arg_size)
    ]
    state = project.factory.entry_state(args=analysis_args)
    if True == args.unicorn:
        for opt in angr.options.unicorn:
            state.options.add(opt)

    race_finder = project.analyses.RaceFinder(
        initial_state=state, ...
           disable_global_filter=args.disable_filter)
```

Listing A.2: binrelay/race_analysis.py

```
import copy
import itertools
import logging

import angr

import networkx as nx

from .utils import pthread_exit

logger = logging.getLogger(name=__name__)
logger.setLevel(logging.INFO)

READ = "read"
WRITE = "write"


class ThreadInfoPlugin(angr.SimStatePlugin):
    def __init__(self):
        super(ThreadInfoPlugin, self).__init__()

        self.prev_thread_id = None
        self.current_thread_id = 0
        self.next_thread_id = 1

        self.TG = nx.DiGraph()
        self.cn = frozenset([0])
        self.TG.add_node(self.cn)
```

```
29
30          self.locks_held = set()
31
32          self.accesses = {}
33
34      @angr.SimStatePlugin.memo
35      def copy(self, memo):
36          result = ThreadInfoPlugin()
37          result.prev_thread_id = self.prev_thread_id
38          result.current_thread_id = self.current_thread_id
39          result.next_thread_id = self.next_thread_id
40
41          result.TG = copy.deepcopy(self.TG)
42          result.cn = copy.deepcopy(self.cn)
43
44          result.locks_held = copy.deepcopy(self.locks_held)
45
46          result.accesses = copy.deepcopy(self.accesses)
47          return result
48
49
50  class _pthread_create(angr.SimProcedure):
51      """
52      A Sim Procedure for pthread_create
53      """
54
55      def run(self, nt, attr, start_routine, arg):
56          thread = self.state.solver.eval(nt)
57
58          self.state.thread_info.prev_thread_id = ...
                self.state.thread_info.current_thread_id
59          self.state.thread_info.current_thread_id = ...
                self.state.thread_info.next_thread_id
60          self.state.thread_info.next_thread_id += 1
61
62          logger.debug("enter thread: %d -> %d",
63                       self.state.thread_info.prev_thread_id,
64                       self.state.thread_info.current_thread_id)
65
66          src_node = self.state.thread_info.cn
67
68          tmp = set(src_node)
69          tmp.add(self.state.thread_info.current_thread_id)
70          dest_node = frozenset(tmp)
71
72          self.state.thread_info.TG.add_edge(src_node, dest_node,
73                                             create=self.state.thread_info.current_t
74          self.state.thread_info.cn = dest_node
75
76          self.state.mem[thread].uint64_t = ...
                self.state.thread_info.current_thread_id
77          self.call(start_routine, (arg,), 'on_return')
78
79      def on_return(self, thread, attr, start_routine, arg):
```

```
80          prev = self.state.thread_info.current_thread_id
81          self.state.thread_info.current_thread_id = ...
                self.state.thread_info.prev_thread_id
82          self.state.thread_info.prev_thread_id = prev
83
84          logger.debug("leave thread: %d -> %d",
85                       self.state.thread_info.prev_thread_id,
86                       self.state.thread_info.current_thread_id)
87          self.ret(self.state.solver.BVV(0, self.state.arch.bits))
88
89
90  class _pthread_join(angr.SimProcedure):
91      def run(self, thread, retval):
92          joined_id = self.state.solver.eval(thread.to_claripy())
93          logger.debug("Join %d", joined_id)
94
95          src_node = self.state.thread_info.cn
96          tmp = set(src_node)
97          tmp.remove(joined_id)
98          dest_node = frozenset(tmp)
99
100         self.state.thread_info.TG.add_edge(src_node, dest_node, ...
                join=joined_id)
101         self.state.thread_info.cn = dest_node
102
103
104 class _pthread_mutex_lock(angr.SimProcedure):
105     """
106     A simprocedure that is executed when a lock (mutex) is taken.
107     """
108
109     def run(self, mutex):
110         logger.debug("Thread %d is locking mutex @ %s" %
111                      (self.state.thread_info.current_thread_id, ...
                            mutex))
112         mutex_address = self.state.solver.eval(mutex.to_claripy())
113         self.state.thread_info.locks_held.add(mutex_address)
114
115
116 class _pthread_mutex_unlock(angr.SimProcedure):
117     """
118     A simprocedure that is executed when a lock (mutex) is released.
119     """
120
121     def run(self, mutex):
122         logger.debug("Thread %d is releasing mutex @ %s" %
123                      (self.state.thread_info.current_thread_id, ...
                            mutex))
124         mutex_address = self.state.solver.eval(mutex.to_claripy())
125         self.state.thread_info.locks_held.remove(mutex_address)
126
127
128 def _mem_read_callback(state):
129     ip = state.ip
```

```
130     from_addr = state.inspect.mem_read_address
131     length = state.inspect.mem_read_length
132     logger.debug("Thread %d is reading from %s at %s" %
133                  (state.thread_info.current_thread_id, from_addr, ...
                        state.ip))
134
135     if int != type(from_addr):
136         from_addr = state.solver.eval(from_addr)
137     if int != type(length):
138         length = state.solver.eval(length)
139     if int != type(ip):
140         ip = state.solver.eval(ip)
141
142     for i in range(length):
143         addr = from_addr + i
144         access = (state.thread_info.current_thread_id, ip, addr, ...
                    READ,
145                   frozenset(state.thread_info.locks_held), ...
                        state.thread_info.cn)
146         if addr not in state.thread_info.accesses:
147             state.thread_info.accesses[addr] = set()
148         state.thread_info.accesses[addr].add(access)
149
150     logger.debug("thread=%d pc=0x%X addr=0x%X rw=r locks=%s tsn=%s",
151                  state.thread_info.current_thread_id, ip, from_addr,
152                  state.thread_info.locks_held, state.thread_info.cn)
153
154
155 def _mem_write_callback(state):
156     ip = state.ip
157     to_addr = state.inspect.mem_write_address
158     length = state.inspect.mem_write_length
159     logger.debug("Thread %d is reading from %s at %s" %
160                  (state.thread_info.current_thread_id, to_addr, ...
                        state.ip))
161
162     if int != type(to_addr):
163         to_addr = state.solver.eval(to_addr)
164     if int != type(length):
165         length = state.solver.eval(length)
166     if int != type(ip):
167         ip = state.solver.eval(ip)
168
169     for i in range(length):
170         addr = to_addr + i
171         access = (state.thread_info.current_thread_id, ip, addr, ...
                    WRITE,
172                   frozenset(state.thread_info.locks_held), ...
                        state.thread_info.cn)
173         if addr not in state.thread_info.accesses:
174             state.thread_info.accesses[addr] = set()
175         state.thread_info.accesses[addr].add(access)
176
177     logger.debug("thread=%d pc=0x%X addr=0x%X rw=w locks=%s tsn=%s",
```

```
                    state.thread_info.current_thread_id, ip, to_addr,
                    state.thread_info.locks_held, state.thread_info.cn)


def find_create_edge_dest(G, t):
    for _, d, attrs in G.edges(data=True):
        if "create" in attrs and t == attrs["create"]:
            return d
    return None


def reachable(G, c, tid, a):
    if c == a[5]:
        return True
    path = nx.shortest_path(G, source=c, target=a[5])
    for s, t in zip(path, path[1:]):
        attrs = G.get_edge_data(s, t)
        if "join" in attrs and tid == attrs["join"]:
            return False
        if t == a[5]:
            return True
    return False


def check(G, a1, a2):
    c1 = None
    c2 = None
    if 0 != a1[0]:
        c1 = find_create_edge_dest(G, a1[0])
    if 0 != a2[0]:
        c2 = find_create_edge_dest(G, a2[0])

    if None != c2 and True == reachable(G, c2, a2[0], a1):
        return True
    if None != c1 and True == reachable(G, c1, a1[0], a2):
        return True
    return False


class RaceFinder(angr.Analysis):
    """
    RaceFinder is the point of this entire project!
    """

    def __init__(self, initial_state=None, ...
       disable_global_filter=False):
        # Save off the SimProcedures so they can be restored ...
           post-analysis.
        orig_hooks = copy.deepcopy(self.project._sim_procedures)

        # Setup BINRELAY's SimProcedures.
        # XXX: Need to do this in a cross-platform way, right now ...
           we assume
        # Linux binaries.
```

```
229         self.project.hook_symbol("pthread_create", _pthread_create())
230         self.project.hook_symbol("pthread_mutex_lock", ...
                _pthread_mutex_lock())
231         self.project.hook_symbol("pthread_mutex_unlock",
232                                  _pthread_mutex_unlock())
233         self.project.hook_symbol("pthread_exit", pthread_exit())
234         self.project.hook_symbol("pthread_join", _pthread_join())
235
236         # Setup the symbolic execution.
237         checked_ranges = set()
238         for section in self.project.loader.main_object.sections:
239             if section.name == ".data" or section.name == ".bss":
240                 checked_ranges.add((section.vaddr, section.memsize))
241
242         if None == initial_state:
243             initial_state = self.project.factory.entry_state()
244         initial_state.register_plugin("thread_info", ...
                ThreadInfoPlugin())
245
246         # Setup breakpoints for memory accesses
247         initial_state.inspect.b("mem_read", when=angr.BP_AFTER,
248                                 action=_mem_read_callback)
249         initial_state.inspect.b("mem_write", when=angr.BP_AFTER,
250                                 action=_mem_write_callback)
251
252         simmgr = ...
                self.project.factory.simulation_manager(initial_state)
253         simmgr.use_technique(angr.exploration_techniques.Spiller())
254         logger.info("Starting symbolic execution")
255         simmgr.run()
256         logger.info("Symbolic execution terminated")
257
258         checked_ranges = set()
259         for section in self.project.loader.main_object.sections:
260             if section.name == ".data" or section.name == ".bss":
261                 checked_ranges.add((section.vaddr, section.memsize))
262
263         logger.info(simmgr)
264
265         logger.info("Checking for race conditions")
266
267         races = set()
268
269         for st in simmgr.deadended:
270             for addr in st.thread_info.accesses.keys():
271                 if False == disable_global_filter and True == ...
                        min([addr < rng[0] or rng[0]+rng[1]-1 < addr ...
                        for rng
272                                                                  in ...
                                                                     checked_ran
273                     continue
274
275                 combinations = itertools.combinations(
276                     st.thread_info.accesses[addr], 2)
```

```
277
278                 for combo in combinations:
279                     a0 = combo[0]
280                     a1 = combo[1]
281
282                     if a0[0] == a1[0]:
283                         continue
284                     if a0[3] == READ and a1[3] == READ:
285                         continue
286                     if len(a0[4].intersection(a1[4])) > 0:
287                         continue
288
289                     result = check(st.thread_info.TG, a0, a1)
290                     if True == result:
291                         logger.debug("possible race on 0x%X", addr)
292                         logger.debug("thread=%d, pc=0x%X ...
                                addr=0x%X rw=%s locks=%s tsn=%s",
293                                      a0[0], a0[1], a0[2], a0[3], ...
                                         a0[4], a0[5])
294                         logger.debug("thread=%d, pc=0x%X ...
                                addr=0x%X rw=%s locks=%s tsn=%s",
295                                      a1[0], a1[1], a1[2], a1[3], ...
                                         a1[4], a1[5])
296                         race = tuple(sorted([a0[1], a1[1]]))
297                         races.add(race)
298
299         for race in races:
300             logger.info("possible race: (0x%X, 0x%X)", race[0], ...
                    race[1])
301         logger.info("Race Analysis Complete")
302
303         # Restore sim procedures
304         self.project._sim_procedures = orig_hooks
305
306
307 # Register the RaceFinder with angr.
308 angr.AnalysesHub.register_default("RaceFinder", RaceFinder)
```

Listing A.3: binrelay/utils.py

```
1  import logging
2
3  import angr
4
5  logger = logging.getLogger(name=__name__)
6  logger.setLevel(logging.DEBUG)
7
8
9  class pthread_exit(angr.SimProcedure):
10     """
11     Simulates pthread_exit by performing a no-op.
12     """
13
```

```
14      def run(self, exit_code):
15          self.ret()
16
17
18  class loop_hook(object):
19      def __init__(self, dest_addr):
20          self.hit_count_for_tid = {}
21          self.dest_addr = dest_addr
22
23      def __call__(self, state):
24          if state.thread_info.current_thread_id not in ...
                self.hit_count_for_tid:
25              self.hit_count_for_tid[state.thread_info.current_thread_id] ...
                    = 0
26
27          if ...
                self.hit_count_for_tid[state.thread_info.current_thread_id] ...
                ≥ 10:
28              state.ip = self.dest_addr
29              logger.debug("tid = %d jumping out of loop, ip = %s" %
30                           (state.thread_info.current_thread_id, ...
                                state.ip))
31
32          self.hit_count_for_tid[state.thread_info.current_thread_id] ...
                += 1
33
34
35  def hook_loops(proj, max_iters=10):
36      proj.analyses.CFGFast()
37
38      loop_finder_result = proj.analyses.LoopFinder()
39
40      for loop in loop_finder_result.loops:
41          edge = loop.break_edges[0]
42          logger.debug(edge)
43          src_block = proj.factory.block(edge[0].addr)
44          jmp_out_addr = src_block.instruction_addrs[-1]
45          logger.debug("jump out addr = 0x%X" % (jmp_out_addr))
46          proj.hook(jmp_out_addr, hook=loop_hook(edge[1].addr))
47
48
49  def pthread_exit_hook(state):
50      state.callstack.ret()
51      state.ip = state.callstack.current_return_target
52
53
54  def hook_pthread_exit(proj):
55      cfg = proj.analyses.CFGFast()
56      for func in cfg.functions:
57          f = cfg.kb.functions.function(func)
58          for callsite in f.get_call_sites():
59              target = f.get_call_target(callsite)
60              target_func = cfg.kb.functions.function(target)
61              if "pthread_exit" == target_func.name:
```

```
62                  bb = proj.factory.block(callsite)
63                  proj.hook(bb.instruction_addrs[-1], ...
                        hook=pthread_exit_hook)
64                  print("pthread_exit call @ 0x%X" % ...
                        (bb.instruction_addrs[-1]))
```

Listing A.4: binrelay/__init__.py

```
1 from .tep import *
2 from .utils import *
3 from .race_analysis import *
```

Appendix B

Sample Binary Source Code

Listing B.1: Samples Makefile

```
SAMPLES := $(basename $(wildcard *.c))

all: $(SAMPLES)

%: %.c
    gcc $@.c -o $@ -pthread

clean:
    rm -f $(SAMPLES)
```

Listing B.2: Sample 1

```
#include <stdio.h>

#include <pthread.h>

static int x; // expecting 1 race on this variable\address

void *t1_body()
{
    x = 5; // Write
}

void *t2_body()
{
    x = 7; // Write
}

int main(int argc, char **argv)
{
    pthread_t t1;
```

```
20      pthread_t t2;
21
22      x = 3; // Write
23
24      pthread_create(&t1, NULL, t1_body, NULL);
25      pthread_create(&t2, NULL, t2_body, NULL);
26
27      pthread_join(t1, NULL);
28      pthread_join(t2, NULL);
29
30      printf("x=%d\n", x);
31
32      return 0;
33  }
```

Listing B.3: Sample 2

```
1   #include <stdio.h>
2
3   #include <pthread.h>
4
5   static int x; // Expecting 2 races on this address\variable
6
7   void *t1_body()
8   {
9       x = 5; // Write
10  }
11
12  void *t2_body()
13  {
14      x = 7; // Write
15  }
16
17  int main(int argc, char **argv)
18  {
19      pthread_t t1;
20      pthread_t t2;
21
22      pthread_create(&t1, NULL, t1_body, NULL);
23      x = 3; // Write
24      pthread_create(&t2, NULL, t2_body, NULL);
25
26      pthread_join(t1, NULL);
27      pthread_join(t2, NULL);
28
29      printf("x=%d\n", x);
30
31      return 0;
32  }
```

Listing B.4: Sample 3

```c
#include <stdio.h>

#include <pthread.h>

static int x; // Expecting 3 races on this address\variable

void *t1_body()
{
    x = 5; // Write
}

void *t2_body()
{
    x = 7; // Write
}

int main(int argc, char **argv)
{
    pthread_t t1;
    pthread_t t2;

    pthread_create(&t1, NULL, t1_body, NULL);
    pthread_create(&t2, NULL, t2_body, NULL);

    x = 3; // Write

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("x=%d\n", x);

    return 0;
}
```

Listing B.5: Sample 4

```c
#include <stdio.h>

#include <pthread.h>

static int x = 0; // Expecting no races thanks to mutex

static pthread_mutex_t mutex;

void *t1_body()
{
    pthread_mutex_lock(&mutex);
    x += 1; // Read and Write
    pthread_mutex_unlock(&mutex);
}

int main(int argc, char **argv)
```

```
17 {
18     pthread_t t1;
19
20     pthread_mutex_init(&mutex, NULL);
21
22     pthread_create(&t1, NULL, t1_body, NULL);
23
24     pthread_mutex_lock(&mutex);
25     x += 1; // Read and Write
26     pthread_mutex_unlock(&mutex);
27
28     pthread_join(t1, NULL);
29
30     printf("x=%d\n", x);
31
32     return 0;
33 }
```

Listing B.6: Sample 5

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include <pthread.h>
5
6  int foo; // Expecting 3 races: 2x Read-Write, 1x Write-Write
7
8  void *thread_body()
9  {
10     foo += 1; // Read and Write
11 }
12
13 int main(int argc, char **argv)
14 {
15     int result = 0;
16     int number = 0;
17
18     pthread_t thread;
19
20     foo = 0;
21
22     if (2 > argc)
23     {
24         printf("usage: %s <number>\n", argv[0]);
25         return result;
26     }
27
28     number = atoi(argv[1]);
29
30     pthread_create(&thread, NULL, thread_body, NULL);
31     foo += number; // Read and write
32     pthread_join(thread, NULL);
33
```

```
34      return result;
35  }
```

Listing B.7: Sample 6

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4
5   #include <pthread.h>
6
7   int foo; // Expecting 3 races: 2x Read-Write, 1x Write-Write
8
9   pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
10
11  void *thread_body()
12  {
13      pthread_mutex_lock(&mutex);
14      foo += 1; // Read and Write
15      pthread_mutex_unlock(&mutex);
16  }
17
18  int main(int argc, char **argv)
19  {
20      int result = 0;
21      const char *password = "barbazbux";
22
23      pthread_t thread;
24
25      foo = 0;
26
27      if (2 > argc)
28      {
29          printf("usage: %s <password>\n", argv[0]);
30          return result;
31      }
32
33      if (0 == strcmp(password, argv[1]))
34      {
35          pthread_create(&thread, NULL, thread_body, NULL);
36          for (int i = 0; i < 100; i++)
37          {
38              foo += 1; // Read and Write
39          }
40          pthread_join(thread, NULL);
41      }
42
43      printf("value=%d\n", foo);
44      return result;
45  }
```