Detecting Data Races In Binaries Using Software Static Analysis

by

Maxime Laurin

Thesis submitted to the University of Ottawa
In partial fulfillment of the requirements for the
Master of Computer Science degree

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

© Maxime Laurin, Ottawa, Canada, 2024

Abstract

In our modern computer systems, more and more programs are using concurrency to improve performance. These programs are vulnerable to unique concurrency issues such as when multiple threads fail to synchronize access. Because these mistakes depend on specific thread timings and are therefore difficult to identify, we present in this thesis a novel static analyzer to detect data races in binary programs.

To help in the identification of data races, we choose to build a program analysis tool. Our analyzer uses a lockset-based analysis inspired by the static analysis RacerX by Engler and Ashcraft. The Racerx analyzer requires source code and most of the tools currently used to detect data races require dynamic instrumentation. We propose in this thesis a new static data race detector with new binary analysis techniques that can successfully identify possible data races in programs without the need for the source code.

Our thesis describes the design of our binary analyzer built on the Ghidra reverse engineering framework. We also tested an implementation of our analyzer on a set of real-world data races previously found in the Linux kernel. Finally, we explore areas where this research can be expanded such as in automated security testing or coupled with surgical dynamic instrumentation.

Acknowledgements

I would like to thank my family and colleagues for their encouragement and support during my Masters. I am also most grateful to Dr. Carlisle Adams for his feedback and supervision which have been indispensable.

Table of Contents

1 Introduction **1**

1.1 Application security 2

1.2 Thesis Goals 4

1.3 Thesis Contributions 5

1.4 Thesis Outline 6

2 Background **8**

2.1 Concurrency 8

2.1.1 Multi-thread programs 8

2.1.2 Race conditions and Data Races 9

2.1.3 Memory Consistency Model 9

2.1.4 Execution model 12

2.1.5 Techniques to find race bugs 16

2.2 Program analysis 17

2.2.1 Static Analysis 18

2.2.2 Dynamic Analysis 19

2.2.3 Why static analysis? 19

2.2.4 Binary Analysis and Source Code Analysis 20

2.2.5 Binary Analysis 20

2.2.6 Source Code Analysis 21

2.2.7 Intermediate representation 21
2.3 Binary Analysis Platform 22
2.3.1 Characteristics of Binary Analysis Platforms 22
2.3.2 Comparing different Binary Analysis Platforms 25
2.3.3 Scripting an analyzer with Ghidra 30
2.4 Related Work 32
2.4.1 RacerX 32
2.4.2 DataCollider 36
2.4.3 Eraser 37
2.4.4 RaceTrack 37
2.4.5 Locksmith 38
2.4.6 ThreadSanitizer 39
2.5 Summary 40

3 Design **42**
3.1 Data Race Detection design 43
3.1.1 Detecting locking operations 45
3.1.2 Detecting memory accesses 45
3.1.3 Gathering thread information 47
3.1.4 Detecting race 47
3.2 Walkthrough 48
3.2.1 Using the tool 49
3.3 Conclusion 54

4 Implementation **55**
4.1 Components 55
4.1.1 Executable Parsing, Code Disassembly and Control-Flow Graph Reconstruction 56

4.1.2 Decompilation 61
4.1.3 Control-flow graph exploration 64
4.1.4 Data Race Detector 70

5 Evaluation **72**
5.1 Methodology 73
5.2 Testing set 74
5.2.1 Program A and B 74
5.2.2 Program C and D 75
5.2.3 CVE-2021-32606 76
5.2.4 CVE-2022-45886 80
5.3 Evaluation Setup 85
5.4 Results 86
5.5 Limitations 89

6 Conclusion **90**
6.1 Conclusion 90
6.2 Future Work 91

Bibliography **93**

Chapter 1

Introduction

Application security is fundamental to the integrity of a computer system. Application security refers to the security of the software being executed by a computer system, such as a web browser or an operating system kernel. These applications have certain privileges and may implement different security policies. For example, a browsing application executes JavaScript code locally but specifically enforces that the JavaScript code loaded from a website cannot interact with data related to a different website (same-origin policy). On an operating system, some files may be accessible only by an administrator user. The security of the application enforcing these policies is therefore extremely important to ensure the security of the system.

Programming mistakes, which we will refer to as *bugs*, in an application can create security vulnerabilities, causing a weakness that can possibly be exploited or used by attackers to violate the security policy of the system. We use the CIA triad (*confidentiality*, *integrity* and *availability*) to model the impact of such vulnerability: vulnerabilities can impact confidentiality (e.g.: a bypass of the authentication mechanism), integrity (e.g.: memory corruption modifying data) or availability (e.g.: make the application crash). Protecting the integrity of an application is critical to the security of the system. Depending on the program implementation, memory corruption can lead

to changes in the application's logic. In the worse cases, this can lead to arbitrary code execution by an attacker. More often than not applications are given a lot of permission and have wide access to the rest of the system or the network. Modern operating systems such as Android implement a more elaborate platform security policy with application sandboxing and SELinux rules to implement mandatory access control (MAC)[1]. While these mobile applications no longer have the great access given to applications on desktop computers or servers, they still inherit powerful permission and have access to sensitive user data. Compromising the integrity of such a program can lead to the attacker gaining control of the application and to take advantage of the permission granted to the application by its user.

1.1 Application security

As the number of critical systems hosting our personal data increases, ensuring its integrity becomes an even more important problem. Multiple approaches are taken to improve the state of application security, such as improved operating system mitigation like control-flow integrity, stack canaries to prevent stack buffer overflows or the use of memory safe languages such as Rust. However, security mitigation does not prevent or detect all possible vulnerabilities and a lot of critical software are still using memory unsafe languages due to their age, their need for low-level memory access or because of inertia. To prevent software vulnerabilities being used by attackers, the cybersecurity industry has created multiple programs and initiatives such as bug-bounty and security disclosure programs with the aim of disclosing found vulnerabilities to the software vendors for security patching while preventing their usage by malicious actors.

Detecting security-relevant bugs and fixing them also improves security by removing a vulnerability possibly used by an attacker. A research report on the life of zero-day vulnerabilities by RAND concluded that exploits have a median survival time of 5.07

years[2]. They also found that the median collision rate, the chance of somebody else finding the same vulnerability, is around five percent per year. Finally, they quickly mentioned the swarm behaviour of security researchers: researchers tend to focus their research on areas known or suspected to contain more bugs. This behaviour leads to an increase in the bug collision rate once an area of a specific program becomes known as an interesting area for researchers. This makes discovering and reporting new security vulnerabilities an activity that can actively improve the security of the software users since there is a reasonable chance that the vulnerability found by a security researcher might've already been in use by a malicious attacker.

The standard way to keep track of security relevant bugs by the cybersecurity industry is the CVE (Common Vulnerabilities and Exposures) system. CVE are unique identifiers for publicly known cybersecurity vulnerabilities affecting software that are publicly released. [3] Companies or individuals who find and report vulnerabilities can register them and have it assigned a CVE identifier by a CVE Numbering Authority.

A pattern quickly emerged from the security vulnerabilities detected and fixed over time in popular software. At the 2019 BlueHat security conference, Matt Miller, a member of the Microsoft Security Response Center gave a presentation on the trends & challenges in the software vulnerability landscape.[4]. Their analysis showed that for the last 15 years, approximately 70% of all CVEs patched by Microsoft were caused by memory safety issues. Over the same period, the number of stack corruption vulnerabilities went dramatically down while heap out-of-bounds and non-initialized memory went up.

Interestingly, a similar result with around 60 or 70% of CVEs being caused by memory safety issues, has been echoed by other companies such as Google [5], [6] or Mozilla [7].

As shown by the CVE data collected, a lot of security relevant bugs are still caused by errors in the handling of the program's memory. A lot of effort has been done

to prevent such issues with improvement to the languages (such as C++ unique_ptr), to the addition of advanced code optimization in compilers and by the use of more thorough and complete testing such as with software fuzzing.

However, while these improvements help to analyze code, software is increasingly running on new platforms that are difficult to access for analysis, such as mobile devices or embedded devices. While open-source code is popular, a considerable amount of software is closed source and can only be found as compiled code.

1.2 Thesis Goals

This thesis focuses on the topic of detecting security relevant data races in software.We choose to detect data races because these are considered to be undefined behaviours in the C/C++ memory model and because data races are the cause of bugs that are very difficult to detect. For performance reasons, more and more codebases use multithreading, introducing the risk of concurrency bugs in the application. Such races often lead to undefined behaviour, memory corruption and therefore, to security issues.

We will detect data race issues in programs by developing a new static program analyzer. The novelty of this analyzer will be its great portability, ease of use and its ability to analyze binary code compared to other tools that requires the program source code or to instrument the system while executing the binary. We will evaluate our analyzer on multiple binaries of different complexity and size. We will also evaluate our analyzer based on its performance, its ability to find previously found bugs in the targeted programs and by hopefully finding and reporting novel vulnerabilities.

By concentrating our efforts on an analysis of bugs that are hard to find and against binary code, we hope to improve the number of vulnerabilities found in areas that were overlooked by other researchers. Program analysis tools and other projects with the objective of detecting data races often have multiple requirements preventing them from

being used against a lot of programs. Most of these tools either require source code or require the program to be running on an instrumented system. We are hoping to be able to create a tool that is closer to being usable on any program of any architecture with relatively little involvement from the user. This would enable our analyzer to be used as part of a continuous integration development cycle or as part of a security vulnerability scanner. We hope that finding vulnerabilities and fixing them should make software more secure by reducing the lifetime of vulnerabilities used by attackers and by reducing the number of exploitable vulnerabilities in the analyzed codebase.

1.3 Thesis Contributions

We make the following contributions in this thesis:

1. **We designed and built a static analyzer to find data races in arbitrary binaries.** We built an easy-to-use data race analyzer on top of the software reverse engineering tool called Ghidra. Using Ghidra allows us to get a platform agnostic analysis by analyzing the intermediate representation generated by Ghidra from the binary. We also have a mature user interface within Ghidra, enabling the user to specify parameters for the analysis. The reverse engineering framework also gives the user the ability to easily customize and control the analysis by adding annotations or renaming variables inside the program. Our data race analyzer will be an easy-to-use tool to detect data races against any software supported by Ghidra.

2. **We improve upon previously built data race detection tools** Our analysis is built on a lockset analysis technique, as used by previous tools such as RacerX [8], a static data race detector. We improve upon it by making our analyzer work against binaries and by implementing new techniques and heuristics to improve

lockset and thread detection while executing the analysis algorithm on binary code.

3. **We conduct a security analysis of data races CVE in the Linux kernel and conduct security research against the Linux kernel** We identify data race bugs found recently within the Linux kernel and we analyze how they happened and how they were patched by the kernel community. The Linux kernel was used for this as it is a major open-source software program, which gives us visibility into previously fixed data race vulnerabilities. We use these case studies to build an evaluation set for our analyzer. This is useful to test the accuracy and the performance of our analyzer and for other researchers to test their program analysis techniques against a set of known data races.

1.4 Thesis Outline

The rest of this thesis will be structured as follows:

1. Chapter 2 will introduce the reader to the data races and program analysis concepts required to understand the remainder of this thesis. Data races and the methods used to detect them will be described alongside an overview of the state of the art in this domain.

2. Chapter 3 introduces the design of our analyzer. This chapter will explain the decisions we've made concerning our analyzer's architecture and our approach to detect data races. We start by describing our choices in binary analysis platform and finish with a detailed description of the algorithms and techniques we used to detect data races.

3. Chapter 4 will present the details of our implementation. It will describe how each component of our analyzer works and is implemented, discuss how to configure

and run our analyzer and finish with an explanation of how the results are triaged at the end of our analysis to give a useful report to the user.

4. In chapter 5 we will introduce our testing methodology which includes a testing suite made of known data races in a set of testing binaries. We then use this testing suite to evaluate the performance of our analyzer. We also describe the result of our security research done using our analyzer to find new data races. We finish this chapter by discussing our results and our analyzer performance.

5. Finally, in chapter 6 we summarize our work and discuss further areas of research and improvements

Chapter 2

Background

In this chapter we introduce the concepts of a race condition and of static analysis, which are key concepts used in this thesis. In section 2.1, we will explain the idea of concurrency in computer programs and of their security implications. We then follow this in section 2.2 by an overview of the program analysis notions required to understand our design. Next, we discuss related work and previous data race analysis tools and their different approaches and challenges in section 2.4.

2.1 Concurrency

2.1.1 Multi-thread programs

A lot of the advances in computer performance have been the result of the increasing number of cores in modern processors, including in the processors used by personal computers and mobile devices. According to an online survey, over 75% of users have a processor with at least 4 cores [9]. For example, current flagship smartphones such as the iPhone 14 contains multiple processors with multiple computing cores. [10] The iPhone 14 has a 6-core CPU and a 5-core GPU[10]. Therefore, software developers now must effectively write concurrent code, which presents its own set of challenges,

in order to take advantage of the new hardware platforms.

Multiple cores allow different processes or threads to run concurrently on a system. We'll use as a definition for *threads* the one used by most modern operating systems: a sequence of execution within a specific process. This means that multiple threads within the same process will share resources, including the same process identifier, the same memory space and the same permissions. The other most common way that code is run concurrently is with multiple processes running in parallel on the operating system. In this case, each process has its own memory space, but may share resources by using inter-process communication (IPC) mechanisms such as the file system or with a specifically designed shared memory space.

2.1.2 Race conditions and Data Races

A **race condition** (also known as a *general race*) is defined in parallel programs as an execution where the order of an access to a common resource depends on the timing of two or more threads (i.e., the threads *race* against each other for access to a resource)[11]. This race condition can change the behaviour of the program if it ends up moving the execution into an invalid or unintended state by the programmer. According to Padua [12], when the unintended program state causes unwanted program behaviour, then it should be considered a programming mistake. Of note, according to this definition, not all race condition are mistakes as many of them do not lead to different program behaviours. Since they introduce non-determinism and are hard to reproduce, race conditions are known to be difficult for developers to diagnose and fix.

2.1.3 Memory Consistency Model

As previously discussed, race conditions can lead to unintended behaviour in a program. In order to precisely define the causes and consequences of race conditions on the state of a program's memory, we need a memory consistency model: a formal specification

of memory semantics. Such an interface has to be defined between the programmer and the system, we will focus on the level between the high-level software programmer and the hardware that executes his code and explain how this influences the writing of parallel programs. The paper on memory models by Saraswat et al. provides a very good summary of the goal of memory models: "Memory models address a central question of imperative concurrency: When can a write done by one thread be read by another?"[13].

The main expectation of programmers regarding the hardware memory consistency model is *sequentiality*. We can easily imagine a simple memory model for single threaded program. The programmer expects that operations are executed sequentially and that the memory is consistent based on the specified program order: the value read from a memory location A should be equal to the value *written* to the memory location A *in a previous instruction in the program order*. This property in a system was defined by Leslie Lamport as *sequential consistency* and is defined as follows [14].

Definition 2.1.1 (Sequential consistency)**.** [A system is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order and the operations of each individual processor appear in this sequence in the order specified by its program.

However, while this model enforces determinism and sequentiality, such a strict memory consistency model is unnecessarily limiting. Compiler optimizations and the processor reordering of the instructions allow for performance optimization while maintaining the semantics of the program. Implementing a stricter memory consistency model becomes a choice between performance and consistency for modern high-level programming languages. As explained in the "Foundations of the C++ Concurrency Memory Model" paper by Boehm and Adve [15], C++ chooses to keep data races as *undefined behaviours*, as it would otherwise incur a lot of added complexity for the compiler and would increase the cost of C++ constructs by requiring the addition of

unavoidable checks.

Some other languages take a different approach to their memory consistency model, such as the Rust language, which uses its ownership model to enforce the absence of data races and of other bugs at compilation time. In contrast, memory unsafe languages such as C still define data races as undefined behaviour. This means a data race will break the language's memory consistency model: the programmer cannot assume the value of a shared memory variable is consistent if a data race happens in a program.

To prevent race conditions, programmers add explicit synchronization to create **critical sections**. A critical section is a block of code to be executed as if it were **atomic**. An atomic execution means that it is independent of the operations of other threads, and the final value of a variable is dependent only on its initial value followed by the operations performed on it. The final state should be dependent only on the initial state. This can be guaranteed if no shared memory used by the atomic section is modified by any other thread (formally stated as *Bernstein's conditions*[16])

Definition 2.1.2 (Bernstein's conditions)**.** With *R(u)* representing the set of addresses read by operation u and with *W(u)* representing the set of addresses written by the operation u, two operations a and b can be reordered without changing the outcome if they respect the following conditions:

$$(W(a) \cap W(b)) \cup (W(a) \cap R(b)) \cup (R(a) \cap W(b)) = \varnothing \tag{2.1}$$

A violation of the *Bernstein's condition* means that an operation cannot be reordered without potentially changing its results. Violations of these conditions are called **data races** and we can consider data races to be a subset of race conditions.

A study on the characteristics of concurrency bugs found in the open-source applications MySQL, Apache, Mozilla and OpenOffice offers a good overview of the types of errors and practices leading to race conditions bugs [17]. It was found that almost all of the non-deadlock concurrency bugs are caused by an unexpected violation of

either atomicity or order in a code region. If we assume *S1*,*S2* and *S3* to be different statements, programmers assume that their code executes sequentially (S1 followed by S2) and do not expect a third another statement (S3) to affect concurrently the resources used by S1 or S2. They also often make incorrect assumptions about the order of execution between threads, which are not enforced. For example, a programmer might assume that the statements in T2, a thread created by T1, are executed after the statements within T1. In reality, the operating system could possibly order T2 to run before T1.

Their study also found that almost all of the examined concurrency bugs involve only two threads and a third of them involve only one variable being accessed. Furthermore, they showed that over 90% of the examined bugs can be triggered with four or fewer specifically ordered memory accesses. This study gives a comprehensive overview of what common race conditions bugs look like. Including one thread accessing a variable before its initialization or one thread accessing one variable while assuming it is executing in an atomic context (i.e., that the variable value cannot be changed by another thread).

2.1.4 Execution model

To help define the problem while facing different types of data race bugs, different types of synchronization techniques (locks, signals, etc.) and different memory models, we need an execution model to represent the execution of our concurrent program. A 2014 survey on race bug detection techniques by Hong et al. gives a general execution model that we will describe here and use to characterize data races. Different race bug detection techniques use different methods and different execution modes. We will base our design on the execution model described in the cited survey, aiming to detect the most useful race bugs using static analysis. [18]

Terminology

We will use the following execution model terminology, taken from the Hong et al. survey. [18]

Critical section: Also known as "Operation block" in the survey. This represents a sequence of operations in a thread intended to be executed with no interference from other threads.

Data Association: In a program, often variables are correlated to each other and need to be accessed or modified together to give a consistent view. An easy example is an array with a correlated variable representing its length. Both values are associated, modifying the array variables (e.g. by reducing its length) involves modifying the associated length variable.

The same survey categorizes race bugs into four classes:

1. Data race bug

   A data race bug is when at least two threads interact with the same memory location with no synchronization and at least one of the operations is a write.

2. Critical section bug

   A critical section bug is when another thread interferes with the intended data flow in one block. For example, if during a critical section we write to a shared variable and then read the value back later in the same critical section. A critical section bug is when another thread comes in and modify the variable, interfering with the intended value of this variable. This could be considered a bug in the implementation of the critical section.

3. Multi-data race bug

   Same as a data race bug but multiple memory locations are modified, in this case the possibility of data association bug also appears. An example is two threads

modifying two different variables in the same structure.

4. Multi-data block race bug

   Same as a critical section bug where multiple memory locations are modified. Once again, this reveals the possibility of data association bugs.

In this execution model, an execution is represented as *operations* and *synchronized order relations* of the operations. An operation is an atomic action such as a read or write to a memory location. A synchronized order relation represents an order on the operations enforced by a synchronization primitive.

Here are informally the pieces composing an execution model and their symbol:

1. A set of operations (Σ)

   Is either a write or a read, made by a specific thread towards a specific memory location. There are few attributes for an operation:

   (a) thread(p): Indicates the thread executing p

   (b) optr(p): Indicates the type of operation (read or write)

   (c) operand(p): Indicates the memory location used by the operation.

2. A set of threads (T)

3. A set of memory location (M)

4. A synchronized order relation on operations ($\rightarrow s$)

   In other words, a way to generate the order relation between two operations, describing when an operation ends before the start of another operation.

   If p and q are operations, p -¿ q iff p and q are from the same thread and p is executed before q or synchronization mechanisms causes q to start once p has ended. The generation of this order is a source of the differences between most bug detection techniques.

5. A conflict relation (C: $\Sigma \times \Sigma$)

   A relation between order of execution and computational result. If the order of execution of p or q can interfere with the other, then they are in conflict. This definition changes depending on the technique.

6. A set of critical sections ($SC : \Sigma \times \Sigma$)

7. A relation of data associations ($S_D A : M \times M$)

   Specifies which data (memory regions) has data associations. If two memory regions are associated, they should be modified together.

The survey mentions that 34 out of the 43 techniques analyzed uses dynamic techniques to help build this execution model by instrumenting the target program.

To obtain the information statically, the survey lists the following techniques used:

1. Alias analysis

2. Data flow analysis

3. Dependency analysis

4. Type systems analysis

The author also noted that such techniques often have bad accuracy (many false positives). This is related to the limitations of static analysis, where a lot of the problems are undecidable or NP-hard. The current solutions must give up optimality to be tractable. Detecting race conditions in a concurrent program implementing mutual exclusions is also NP-hard according to Netzer and Miller [11].

2.1.5 Techniques to find race bugs

Data race

As defined before, a data race can be defined as at least two threads who access the same memory location with no synchronization and one of the operations is a write.

In an execution model, this can be represented by the following conditions: $p, q \in \Sigma$

1. $thread(p) \neq thread(q)$
2. $operand(p) = operand(q)$
3. conflict(p,q)
4. $p \nrightarrow sq \wedge q \nrightarrow sp$

The last condition means that there is no synchronization to have p execute before q (and none to have q executes before p)

Here are the different ways used to find these conditions in a program according to the survey:

1. Condition #1: Obtaining thread information

   The survey notes two techniques: dynamically using a thread identifier during runtime or statically. When doing it statically, there has to be an approximation such as trying to identify and track functions that create new threads.

2. Condition #2: Detecting access location

   There are two challenges in detecting accesses to a specific memory location:

   (a) Extracting control flow information When exploring possible code paths in a program containing conditionals and loops, we will quickly face an exponential number of paths, a problem known as path explosion. Heuristics

have to be made to be able to explore the memory accesses made in the program with their context.

(b) Pointer aliasing An alias occurs when two pointers point to the same memory location. Determining if two pointers points to the same location at a particular program point was found to be undecidable by William Landi.[19] During dynamic analysis, memory accesses can be recorded at runtime. For static analysis, approximations have to be made and some aliases found will be missed or will be wrong.

3. Condition #3: Conflict definition

   If more than one access is detected at a specific memory location, a conflict is defined when at least one of the accesses is a read and at least one of the accesses is a write.

4. Condition #4: Finding the synchronized order relation

   Every technique used the lock operation as a way to build critical sections and create an order relation. Some techniques also looked at the message send/receive and thread create/join operations. In the end some sort of synchronization mechanism; either explicit (locks) or implicit (happens-before relationship) has to be used and identified.

2.2 Program analysis

In this thesis we are interested in detecting security relevant data race bugs in software with binary analysis. As explained in the previous section, they are one of the hardest types of bugs to detect as they are non-deterministic and the race may very rarely resolve in a way leading to undefined behaviour. For performance reason, all modern applications are typically multithreaded and have to share variables, leading to possible

data races. We will focus on intra-process data races between multiple threads of the same process.

In this chapter we will compare two approaches to software analysis and describe their usage and limitation when trying to detect data races. Binary analysis will also be evaluated compared to the more common approach of source code analysis. This section will conclude with an observation of the current state-of-the-art techniques and tools to identify data races and race conditions.

2.2.1 Static Analysis

Static program analysis is performed without executing the program and can target either the source code or the binary. The main benefit of static analyzers is their ability to reason over all possible contexts and input by abstracting them. However, the problems tackled by static analysis quickly become undecidable. A classic theorem in the theory of computation, the *Rice Theorem* states that any nontrivial semantic property of a program is undecidable.[20] Even more research was done on specific problems faced by static analysis, such as determining if two pointers are aliases (i.e., points to the same memory location) in an inter-procedural analysis, and were determined to be undecidable. [19]

This forces static analysis to make approximations, which leads to the well-known problem of either too many false positives or false negatives. Performance is also an issue with code path explosion and the exponentially increasing number of states.

Different models can be built from static analysis to gather different information about the program. For example, a control-flow analysis can be done to recover a control flow graph (CFG), which can be used to find code paths that will be executed or to determine relationships between functions. Alternatively, data-flow analysis can use a CFG to track the values of variables throughout the program. More advanced techniques such as abstract interpretation can also be used to model the program in

a more tractable way, such as abstracting integer variables to their signs (positive or negative).

2.2.2 Dynamic Analysis

A dynamic analysis is done while executing the target program. This allows the analysis to have access to a lot of useful information from the program execution such as values of variables in memory, thread numbers, memory accesses executed and so on. While this greatly enhances the analysis; this is information is only obtainable on the current program context and require the ability to instrument the program. For completeness of analysis, dynamic analysis faces the main problem of trying to explore all possible contexts of the target, often with the help of fuzzing or of runtime instrumentation (for example to measure path coverage).

2.2.3 Why static analysis?

Code analysis is often done to find security issues such as bugs or vulnerabilities or determine the properties of the program. Two types of program analysis are possible:

1. Static analysis: Analysis of a program without executing it.
2. Dynamic analysis: Analysis of the state of a program when running it, sometimes also modifies the original program with the goal of having better monitoring.

Program analysis aims to find the semantic properties of a program, while program verification seeks to ensure that it is correct. In this research, we are interested in finding data races by examining the semantics of a program, since most software does not have a formal specification.

The problems with dynamic analysis

The goal of our analysis is to find data races and other concurrency bugs in software. Here are the reasons we choose static analysis over dynamic analysis:

1. While dynamic analysis seems convenient for acquiring information about concurrency of the application, race conditions are known to be extremely hard to trigger and hard to diagnose because they depend on timing.

2. Dynamic requires the need to run the target program, which is not always possible or easy. The analysis might also add overhead, which again might impact the timing required to trigger a race.

3. A lot of research was done on detecting race conditions dynamically, by using instrumentation to monitor memory accesses or lock taking. We believe better coverage can be done by using static analysis to detect bugs who are not detected with dynamic instrumentation. A survey shows over 17 data race detection tools, of which 4 are using static only techniques. [18]

4. When analyzing lower-level code, dynamic analysis becomes harder, as the code being analyzed or instrumented is often also required by the analysis tool.

We believe that a novel static analysis tool can be used to detect concurrency issues in software that are otherwise hard to analyze dynamically or when the race happens in very rare occurrences.

2.2.4 Binary Analysis and Source Code Analysis

2.2.5 Binary Analysis

A huge problem in practice is that the source code of the application is not always available. A lot of modern analysis tools depend on having access to source code,

either for static analysis or to add instrumentation to improve dynamic analysis.

Finally, the biggest reason to analyze binaries is that binaries are what is executed on the processors, not the source code. This is explained in more detail in the WYSINWYX: What You See Is not What You eXecute paper.[21]. If we are interested in bugs and their security impact, then those can be introduced or affected by the decisions made by the compiler. Any code deemed "undefined behaviour" by the language can lead to very different choices and binary for different compilers.

This paper also mentions the basic steps required to do binary analysis. The first step is to represent the binary in a standard way, to extract the meaningful information and to have our analysis work on this standard model. This is commonly known as the intermediate representation (IR), a concept also used in compilers.

2.2.6 Source Code Analysis

Binary analysis is an analysis done on binary code; architecture-specific code ready to be executed by a processor. Another popular type of analysis done is on source code, the code written in a programming language by the programmer.

2.2.7 Intermediate representation

To create a good IR, a lot of work must be done such as binary parsing (to find code section, data section), code disassembling and control flow graph construction. A big difficult in translating binary code to an IR is that the side effects of instructions (for example, setting a status flag in the processor) have to be represented in the IR. Different IR can have different properties, for example, REIL [22] is an IR made with the goal of facilitating reverse engineering. As such, it has a very small instruction set and it is designed to be easier to read than other IR.

2.3 Binary Analysis Platform

A binary analysis platform can operate on binaries of different architectures and transform the binary characteristics into a representation appropriate for an analysis program. To enable our analysis against as many programs as possible, we propose that our analysis run on top of an IR constructed by a binary analysis platform. The choice or construction of a good analysis platform is essential for the success of our analyzer, as it is the source of the data ingested for our analysis.

In the next section, we describe the characteristics researched in a binary analysis platform. We discuss the characteristics of different existing platforms as well as how they work. Following this decision, we evaluate the various features offered by different platforms and compare them with each other. According to our design requirements, we select a binary analysis platform and explain our decision. We design our analyzer on top of this binary platform and describe the flow of information between an arbitrary binary and the IR used by our analyzer. Finally, we emphasize the impact of our design and its influence on our analyzer's accuracy and portability.

2.3.1 Characteristics of Binary Analysis Platforms

One of our primary goal is to develop a tool capable of analyzing as many programs as possible. Since a significant number of programs do not have source code available, binary analysis support is a requirement in our design. Handling binaries is a well-known challenge due to need to handle the different processor architecture, processor platform, executable format and because of the common idiosyncratic operating system behaviours or features.

To solve these problems, binary analysis frameworks are built to translate a diverse set of binaries into a uniform representation that is ready for analysis. These frameworks create an IR from the binary, just as a compiler creates a different representation

from the source code before translating it into assembly for the targeted architecture. They also handle steps such as the parsing of the executable format, identifying code sections, disassembling code, constructing a control flow graph and other features that will be explained below. Figure 2.1 provides an overview of the various steps from a binary to its final representation

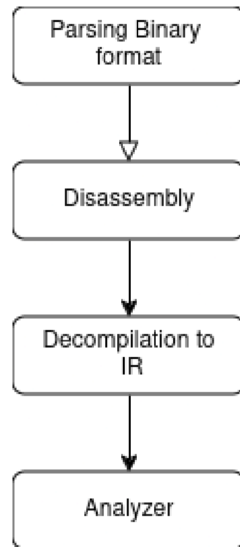

Figure 2.1: The different steps from binary to analysis

We are interested in building a platform-agnostic analysis by writing it on top of an existing IR. There are already many tools meant to lift binaries up to different IR. A large variety of these tools originate from academia and are used to improve either reverse engineering or static analysis techniques. Some of these IR, such as REIL, are new contributions to the field of program analysis research while other IR are created to help solve a specific research project [22]. As such, a lot of those tools either have a lack of portability, support or are scoped for a specific problem.

The most popular IR formats are used by compilers. Many tools are built on top of these IR to add new features or provide support during the compilation process, such as supporting a new computer architecture. The most well-known of these is the LLVM IR, which is part of the LLVM compiler infrastructure. To benefit from this extensive ecosystem, some frameworks choose to convert binary directly into the LLVM IR. The main advantage of using such an IR is that it is the representation used by compilers, meaning our analysis can also be implemented as a step during compilation. There are many binary lifters to LLVM, the McSema project by Trail of Bits contains a comparison table evaluating different LLVM lifters. [23]. There are eleven lifters on this table, each with different levels of support and completeness. Some of those tools are lacking in portability or adequate platform support. Furthermore, the LLVM IR is a very complex representation and was designed for compiler development rather than for analysis.

Finally, multiple reverse engineering and binary analysis tools have their own IR. These tools are designed to analyze binaries and ultimately need to build their own IR or representation of the binary. They usually provide an API to use to create program analysis scripts. Examples include IDA Pro, Binary Ninja and Ghidra. Since these are professional tools, they require a paying monthly subscription to use (aside from Ghidra which is open source).

In the next section, we provide a short overview of Ghidra and discuss its design. We

also compare its performance with other binary analysis platforms, noting its particular strengths and weaknesses. Since our analysis will be built on top of Ghidra, it is especially important to consider Ghidra's performance in the different aspects of reverse engineering that interests us, such as the correct detection of functions and the recovery of the program's control-flow

2.3.2 Comparing different Binary Analysis Platforms

State of the art in Binary Analysis

The main purpose of a binary analysis platform is to handle the binary disassembly required to provide us with a model for building our analysis. Binary analysis is a complex problem due to the use of certain advanced techniques in binaries, including indirect function calls and with the loss of a lot of information from the compilation process. Program symbols, representing the name and location of functions in a binary, are usually stripped from binaries to reduce its file size.

The systematization of knowledge paper by Pang et al.[24] gives an overview of the current state of binary disassembly. For vulnerability research, four pieces of information were deemed required:

1. Instruction disassembly
2. Control flow information
3. Function entry identification
4. Symbolization (determining cross-references)

We will explore each of these points, understanding how each of the surveyed platform approaches it, as well as their limitations and influence in the context of a binary analysis.

Instruction Disassembly against Linux binaries				
	Average		Minimum	
	Precision	Recall	Precision	Recall
Ghidra	99.99	99.65	99.96	64.51
ObjDump	99.98	99.99	88.80	99.70
McSema	99.99	99.99	99.96	99.80
Angr	99.99	99.97	99.95	97.64
IDA	99.99	99.99	99.61	98.15
BinaryNinja	99.99	99.81	99.72	89.85

Table 2.1: Successful instruction disassembly between the most popular disassemblers. Data summarized from Pang et al.[24]

.

Disassembly and function identification

There are two main strategies for disassembly: linear sweep, which tries to disassemble each memory location in succession in a given range and recursive descent, which begins disassembling at a specific address and proceeds to disassemble by following the control flow. Linear sweep disassembly has higher recall but lower precision, as it aims to disassemble every memory location and may try to disassemble data. For the recursive descend method, following the control flow is a major challenge. Since the perfect recovery of indirect control flow is an undecidable problem, the disassembly inevitably fail to follow the control flow perfectly and will therefor miss some of the code. The evaluation of Pang et al. [24] shows that, on average, recursive descent misses 49.35% of the code. To improve code coverage, the implementation of heuristics, such as searching the binary for specific byte patterns, are necessary.

While comparing the different binary analysis platforms on instruction disassembly,

Function entry identification against Linux binaries				
	Average		Minimum	
	Precision	Recall	Precision	Recall
Ghidra	99.94	99.66	86.82	55.11
Angr	91.44	98.29	22.13	80.00
IDA	99.47	92.64	87.34	36.28
BinaryNinja	97.24	99.78	38.75	85.88

Table 2.2: Successful function entry identification between the most popular binary analysis tools. Data summarized from Pang et al.[24]

.

we notice that against the average binary all of them obtain high precision and recall values. As shown in the table 2.1, against the average binary. It is important to keep in mind the difficulty in recovery all the instructions in a binary. Our analysis can only be as good as the underlying data and while some disassemblers such as McSema are better at instruction recovery against the more complex binaries, we are assuming for our analyzer that we want to optimize for the analysis of an average binary rather than for specific complex binaries.

For function identification, the main techniques used are based on patterns and control flow. For example, when a direct function call occurs, the platform assumes that the call destination is a function entry point. It also searches for common function prologues and epilogues to identify additional functions. As seen in table 2.2, these heuristics work well in standard binary executables. Ghidra effectively finds the most function entry on average. However, against the harder samples, the binary analysis platform surveyed struggle either on precision or recall. These results suggest that while function identification is accurate in general, this is one area of concern for our

Control flow graph reconstruction against Linux binary				
	call graph		Jump table	
	Precision	Recall	Precision	Recall
Ghidra	99.99	96.66	98.49	75.58
Angr	99.99	99.93	99.80	87.34
IDA	99.99	99.60	99.95	99.99
BinaryNinja	99.99	99.75	99.76	99.30

Table 2.3: Successful control flow graph recovery of the most popular binary analysis tools. Data summarized from Pang et al.[24]

.

analysis as we iterate through identified functions. Special care should be taken if analysis is to be done against complex binaries to avoid missing the detection of too many functions.

Control flow

The main challenge in recovering and constructing a control-flow graph is identifying the target of indirect jumps, indirect calls and other constructs that have effects on the control flow, such as non-returning functions.

For indirect jumps and calls, binary analyzers try to determine the base of the jump table. If this fails, it will search for constants to determine the location of the function call. To detect non-returning functions, a set of known non-returning library functions and system calls are known to the platforms and can be used to identify the majority of these cases. For the other non-returning functions, some heuristics such as whether the code falls outside of standard locations (for example, in a data location or in a location that contains another function) are used.

According to the data from Pang et al. seen in table 2.3, the different platforms have an excellent precision and recall for detecting both direct calls and for determining the targets of calls made by jump tables. The original paper also describes the performance on other control flow recovery tasks such as on the detection of tail calls, non-returning functions and of intra-procedural flow. While Ghidra has a relatively lower recall for detecting non-returning functions than the other platforms (with a recall of 75.58, compared to other tools which have around 80), this is not expected to impact its performance against most binary targets. A failure to detect a non-returning function could lead to incorrect control flow recovery, but such a situation can be detected and corrected during the course of our analysis.

Symbolization and cross-references

The goal of symbolization is to recover symbols in the binary, as well as to recover them when they are referenced in functions. A symbol is the name associated with a function entry point or with a memory variable used in the program.

Symbolization involves many heuristics, including the assumption that symbols referring to code only point to function entry points or that pointers have a specific size or alignment.

Symbolization algorithms usually try to identify address tables (location in memory of consecutive pointers) and attempt to infer the data types of operands to determine whether any of these operands are pointers and if so, what data types they point to.

In this case, aside from Ghidra, open-source tools tend to outperform closed-source ones. This is caused by the heavy use of heuristics and their influence in the performance of these tools. In the end, Ghidra does not perform well against the more difficult cases, it obtains a lower precision of between 60 and 80% compared to the other platforms and a lower recall of around 40% compared to Angr and McSema. Of note by the authors is the fact that programs with more data tend to cause more

Symbolization against Linux binaries				
	Average		Minimum	
	Precision	Recall	Precision	Recall
Ghidra	99.90	95.70	61.99	45.52
McSema	99.99	99.77	99.04	94.05
Angr	99.86	99.52	81.74	95.38
IDA	99.98	95.96	97.17	36.96
BinaryNinja	99.99	79.89	99.81	17.29

Table 2.4: Successful symbolization recovery of the most popular binary analysis tools. Data summarized from Pang et al.[24]

.

symbolization mistakes.[24]

2.3.3 Scripting an analyzer with Ghidra

For our research, we chose Ghidra, an open-source reverse engineering framework developed by the NSA. Its main advantages are its maturity (Ghidra can parse binaries on the most popular architectures) and it is an open source project with very good documentation that can be modified if necessary. Ghidra projects and scripts are also easy to share.From the survey of the different binary analysis platforms by Pang et al. [24], Ghidra performs more than adequately compared to the other binary analysis platforms while analyzing the average binary. We decided not to use a binary lifter to LLVM, since those seemed more complex to use and sometimes required additional tools to handle specific platform architecture. For example, McSema needs another system such as IDA Pro to create a control-flow graph. Because a mistake in an earlier part of the analysis can propagate and be hard to find, we prefer to use fewer systems.

In our case, the analysis will consist of creating a control-flow graph with Ghidra and transforming the binary into p-code (Ghidra's internal IR). Finally, Ghidra also offers an extensive scripting API making it easy for researchers to access the results of these analyses as well as the corresponding p-code representation.

An interface is present in Ghidra to display the results of its binary analysis to the reverse engineer. P-code is a language used by Ghidra as an IR to model the behaviour of a binary. It aims to provide a common representation that allows processors from different architectures to be analyzed.

Each original instruction is translated into a sequence of p-code operations. A p-code operation accepts one or more *varnode* as input, and it can produce zero or one *varnode* as an output. It has no side effect on any varnode other than the output varnode.

A *varnode* is a generalized representation of a variable, represented by its address space, its offset and its size.

An address space is represented by a name, a size and an endianess. They can be used to represent the process's memory, location of registers and to store constants used by the binary.

Ghidra uses many different algorithms to analyze the p-code that it obtains, simplifying the results and making them more suitable for use in the next decompilation step. The resulting p-code provides a model that is well suited for analyses that need to use the *varnodes* and address space to keep track of the data flow.

Another advantage of using Ghidra's p-code is that we can still take advantage of Ghidra's reverse engineering framework features in our scripts. For example, if for some reason Ghidra's fails to detect a function correctly, we can use built-in tool in the reverse engineering framework to manually correct the situation and make sure the function is being detected and that the control flow to this function is correct. Any changes made to the program within Ghidra will be reflected in our analysis scripts.

We can also set the names of functions and variables in the program. These names will be used in our analyzer, making the output of the analysis easier to understand. Our analysis scripts have been designed for ease of use, but they are complemented by the possibility of performing manual binary analysis by advanced users.

Our analyzer explicitly uses this approach while attempting to solve the pointer aliasing challenge. If two varnodes names are exactly the same, we assume that they refer to the same memory location. This is helpful for allowing the user to quickly annotate such a situation, despite our analyzer's difficulty in finding that specific alias relationship between the two varnodes. Otherwise, as described in greater detail in section 3.1.2, the higher-level features of the p-code *varnode* are used to determine its origin and whether aliasing occur.

2.4 Related Work

2.4.1 RacerX

RacerX [8] is a static tool using flow-sensitive, interprocedural analysis to detect race conditions and deadlocks. It was made by Dawson Engler, well known for making KLEE and other bug finding tools at Standford, and Ken Ashcraft one of his students.

It uses flow-sensitive, interprocedural analysis to detect race conditions and deadlocks.

Overview

There are 5 phases in RacerX:

1. Retargeting to locking functions (manual process)
2. Extracting control flow graph (CFG)
3. Deadlock and race checker on the flow graph

4. Ranking results
5. Human inspection

In RacerX, the operator must configure it for a new system by supplying it functions used to acquire and release locks (and those enabling and disabling interruptions). The operator also must give some details on the locks, such as if they are spinlocks, semaphores, read-only locks, etc. They can also give more details such as specifying some functions as single-threaded.

The extraction phase built a CFG for each file, it contains all function calls and any concurrency operation (locks) is annotated, including their names, types, etc.

The analysis takes all the CFG files and then tries to construct a whole system linked CFG. It then traverses it to check for deadlocks and races. There is the possibility of loops in the CFG and of having multiple analysis starting points called *roots*. For example, a kernel has multiple roots for its different system calls.

For each root, RacerX does a depth-first inter-procedural search, tracking the locks acquired by the program. By looking at the current program statement, it decides on if there is a possible error with the lock-set (lock-set analysis will be described in the next section). The analysis result is then cached to save time when reanalyzing the same parts of the CFG to determine if data races occur.

Finally, the results are ranked. The author notes that this was harder than the analysis part.

Lock-set Analysis

A lock-set is the set of all locks taken during a memory access. When doing lock-set analysis, the goal is to monitor all access to a memory location and the lock-set of these accesses. To detect if it is possible to access this memory location with no protection (no lock), we can verify that at least one lock is taken for all the access (i.e., the intersection of all the lock-set is not empty).

A common issue of lock-set analysis is that it detects violation of a specific locking discipline and not all data races. It may also miss violations of other types of synchronization, such as semaphores.[25]

The lock-set analysis starts at every root of the CFG and does a depth first search (DFS). It can be summarized in three parts:

1. traverse_cfg Iterate over all roots in the CFG and calls traverse_fn
2. traverse_fn Analyze a function f with given lock-set l. If the result was cached previously, return that value. Else, call traverse_stmts on the entry of the function.
3. traverse_stmts Does a DFS traversal of all statements in fn. Compute the set of lock sets on each path (given l the initial lock-set given).

As usual, a lot of details and heuristics have to be made to avoid undesirable situations. For example, recursion is broken after a certain number of iterations and a limit is made on the number of times a function can be analyzed. (For example, some kernel functions are called very often, which leads to a reanalysis whenever the callee has a different lock-set).

Ranking

Result ranking is based on three criteria

1. Locality of involved locks Preference for global
2. Depth of the call chain, shorter is better
3. Number of threads involved; fewer is better.

Challenges with race condition detection

The authors noted three challenges to detect a race condition:

1. Lock-set accuracy. Analyzing memory access with an invalid locket will lead to a lot of false hits.

2. Detecting concurrency. Is the code doing an unprotected access running concurrently? And with whom?

3. Is the race an issue? Some unprotected access to shared memory is harmless.

Race false positives happen if not all valid locks are in the lock-set. The authors used techniques to handle mistakes in the lock-set. Since they do not do path-sensitive analysis (which they called undecidable, there was the possibility of going through a false path in a function.

To rectify the wrong lock-set, they prune it when on a path that contains locking errors (e.g., double acquisition of lock). They also try to reduce it by only propagating the most plausible lock-set from the function summaries. For example, the one which is in majority (most likely the right lock-set).

To find if a function is multithreaded, RacerX uses belief analysis If the programmer uses functions or locks, then the programmer believes the function is multithreaded (and it probably is!) The other way RacerX determines if code is multithreaded is programmer annotation.

Finally, to determine if a race is harmless, they ignore variables that are never written, they require no concurrency control. When a variable is almost always read, then they deem it a counter used for statistics and ignore it. They also favour write accessed and rank errors based on how many unprotected accesses a thread encounters. They also favour errors on variables which were more often accessed within a critical section since the programmer believes those must be protected.

2.4.2 DataCollider

DataCollider is a lightweight data race detection technique for kernels. It is a dynamic analysis tool, instrumenting a sample of the memory accesses being executed by the kernel. This instrumentation is done using modern hardware features, leading to almost no runtime overhead for the non-sampled memory accesses. [25]

Overview

DataCollider works by placing breakpoints on a subset of the memory location accessed by the program based on its sampling algorithm. When one of the breakpoints is triggered, DataCollider will suspend the thread triggering the first breakpoint and put back another breakpoint on the same memory location. It then waits a little bit to catch any other thread trying to access this location. If the breakpoint triggers again, a race is detected and both culprit threads are known.

One of the major design considerations of a dynamic tool like DataCollider is the sampling algorithm. Sampling every memory access would be too costly, as a program can execute thousands of memory accesses. While other tools may decide to sample memory accesses during a fixed time interval or based on how often the memory accesses are executed (adaptive sampling), DataCollider uses data breakpoint which lets it use a low sampling rate.

DataCollider uses *static* sampling: the analyzer starts with static analysis of the program to identify memory locations to instrument using the data breakpoint. This is performed uniformly through all of the program locations which allows DataCollider to identify even rarely occuring data races.

The researchers of DataCollider implemented their analyzer against the Windows operating system and were able to find 38 data races, of which 25 were confirmed as bugs.

They found that around half of the benign data races involved were updates to

counter values or the update of a flag bit in memory. Out of the 38 data races found by DataCollider, only one crashed the kernel.

2.4.3 Eraser

Eraser is a dynamic data race detector tool for multithreaded programs published in 1997. Eraser defines the concept of *locking discipline* and the tool verifies that all memory accesses follow this programming policy. [26]

Overview

Eraser uses a lockset algorithm to enforce the policy that all shared variable access needs to be protected by a lock. This is done by Eraser by monitoring the lockset of each memory access and by checking the intersection of the different lockets when a shared variable is accessed.

Eraser takes a program binary as input and adds instrumentation to produce a new instrumented binary implementing the lockset algorithm. Each load and store instruction in the program are instrumented as well as each call to acquire or release a lock.

As performance was not a focus of Eraser while implementing their solution, the instrumented binary was slowed down by a factor of 10 to 30. Eraser also produces false alarms, mostly caused by benign races or because Eraser missed locks being acquired.

2.4.4 RaceTrack

RaceTrack is a dynamic race detection tool using a hybrid detection algorithm to direct efforts to areas that are more suspicious [27]

Overview

RaceTrack is a data race detection tool for object-oriented programs on the Microsoft .NET platform and runs its analysis on the Common Language Runtime (CLR). .NET applications are translated in this intermediate language and are instrumented at the virtual machine level.

To stay practical while analyzing large programs, RaceTrack uses a novel adaptive approach to dynamically adjusts which memory accesses are monitored based on happens-before analysis.

RaceTrack also implements *Adaptive Granularity* by monitoring accesses on an object-level, before automatically refining the detection down to field-level when a potential race is detected. This is done by running the detection algorithm twice, once tracking object accesses and once again focusing on the fields.

RaceTrack generates warnings and by post-processing them. Warnings are ranked based on their granularity (field-level warnings are higher than object-level warning) and warnings generating similar stack traces are ranked higher. RaceTrack also supports

Adaptive tracking results in a slowdown ratio of between 2 and 3 depending on how much intensively the program uses memory. RaceTrack was able to analyze the CLR regression test suite with only a small number of false positives.

2.4.5 Locksmith

Locksmith is a static analysis tool to detect data races in C programs. Locksmith uses an interprocedural dataflow analysis [28]

Overview

Locksmith works by first parsing the C source code into a sub-language. This is first used for labeling by traversing the control flow graph and identifying the run-time memory locations that contain locks or data. They take care to be field-sensitive and to model each field of a C structure separately.

The next phase of Locksmith tries to detect which memory location could potentially be accessed simultaneously by two or more threads. To determine these locations, Locksmith calculates which memory location can be accessed by each thread and calculates their intersections.

Locksmith then computes the state of each lock at every program point by generating a lock set. To determine if a data race occurs, the intersection of these locksets is calculated. To evaluate Locksmith, a benchmark made of POSIX programs and of Linux device drivers were used. Over 200 000 lines of code were analyzed in 30 minutes with 400 warning generated and 67 races found.

2.4.6 ThreadSanitizer

ThreadSanitizer is a dynamic detector of data races implemented as a Valgrind tool. It observes the program execution as a sequence of events, including memory access events and synchronization events. This tool also introduces dynamic annotations: a way for users to inform the detector about synchronization in the program using C macros. [29]

Overview

ThreadSanitizer consists of a global state and of a state for each detected memory location. Each memory location state keeps tracks of the reading and writing operations received and checks if they cause a data race based on the presence of locks.

The hybrid algorithm used by ThreadSanitizer combines happens-before and lockset algorithms. The happens-before relationship is established through synchronization events like Signal and Wait, creating a partial order of events. Locksets are used to track locks held by each thread, and race conditions are detected when two threads access the same memory location with at least one write access and without proper synchronization

Using C macros, the user can specify details about synchronization in their program, such as defining happens-before relationship. This interface is useful to make ThreadSanitizer ignore false positives and improve the overall accuracy of the tool.

Typically, in Google's unit tests and applications, the slowdown experienced is 20 to 50 times the original execution speed. Most of the analysis time is spent on intercepting and analyzing memory accesses. Caching and sampling of the memory accesses have been done, but ultimately memory access calls are being replaced by a much more expensive series of operations.

ThreadSanitizer has been successfully used to find bugs in the Chromium browser and is now part of Google's integration test for the Chromium browser

2.5 Summary

This chapter describes the main challenges and techniques used in both binary analysis and for detecting data races. We define what is meant by data race and concurrency bug in section 2.1 and then explain what program analysis is in section 2.2. We then did an overview of related program analysis work done to identify data races in modern software in section 2.4.

The prior work in this area requires either instrumentation of the system coupled with sampling of the instrumented memory accesses or required the source code to analyze. This is a gap for the analysis of software running on embedded devices or

mobile devices, where instrumentation can be hard to implement and where source code is not always available. There seems to be consensus on some of the techniques used in the different data race analyzers such as the construction and use of locksets.

However the main challenges remains either making the instrumentation efficient for dynamic analyzers or the precise tracking of memory accesses required for static analyzers Based on the difficulties previously mentioned about dynamic analysis (requiring instrumentation of the operating system or of the target program, difficulty to identify hard to trigger data race, etc.) we have chosen to write a static analyzer. We also decided to make our analyzer work against binaries since it would enable analysis of software even without access to the source code and because related works tend to be either dynamic analysis or require source code. We hope that this allows our analyzer to run easily on more software that could not easily be analyzed by other tools and to make it easy to use for the programmer. The next chapter will introduce the design of our analyzer and explain the choices we have made for our solution.

Chapter 3

Design

The state of the art in data race detection is described in chapter 2. In this chapter, we consider the challenges and techniques previously used to detect data races and we propose a design for performing detection of data races against as many target programs as possible. Our design emphasizes usability across all target programs, while also being effective. The analyzer should be easy to use and allow for the identification of possible data races against any computer program in a reasonable amount of time.

As seen in the previous chapter, most analyzers used to detect data race are dynamic tools. They are based on using operating system features to detect data races when they occur during the program execution. They track memory accesses and modify the environment to create more favourable conditions to provoke and identify data races. The following characteristics of dynamic analysis make it inconvenient for our goal of easy-to-use, portable and accurate analysis:

1. Non-determinism A dynamic analysis coverage depends on the code path that the program execution takes. For concurrency bugs such as data races, it also depends on the random nature of concurrency scheduling.

2. Operating system dependence To run such analysis on a targeted program, we

need to be able to run the specific program inside a specific environment which may not always be possible.

3. Bug analysis When a bug is detected by a dynamic tool, it may be hard to identify the cause of the faulty memory access. The cause of the data race can be located far from where the code accessing the memory is located.

In the next section, we will describe our analyzer's design and describe how each of it's component works. This is followed by an explanation of our choices for data race detection algorithms and by a discussion on the tradeoffs and the challenges we encountered while engineering the analyzer these Finally, we summarize our design, detailing the different steps of our analysis.

3.1 Data Race Detection design

Our data race detection design is based on building a lockset for each memory access detected in the program, similar to the RacerX analysis engine [8]. We chose the lockset approach because it allows us to use caching to avoid reanalyzing previously explored functions. A lockset-based analysis requires precise detection of the following aspects of the target binary:

1. Locking instructions
2. Memory accesses
3. Threading information

As we are analyzing binaries rather than source code, our program will contain heuristics to navigate the binary more effectively while keeping a reasonable amount of precision. We will describe in the following sections how our design solves these three challenges respectively. We chose to perform a top-down, flow-sensitive interprocedural

analysis. The analysis begins with a set of entry points specified by the user, followed by a depth-first search (DFS). As we explore the control flow graph made by Ghidra, we will keep track of the lockset values between different functions (interprocedural), and we will handle the effects of basic control flow operations (such as if/else conditionals) correctly (flow-sensitive). Because handling these situations correctly in very complex programs can be very expensive computationally and algorithmically, we will use a set of heuristics. One example of the heuristics we plan to have is to limit the depth of the DFS search to avoid it going down in the wrong path, we are assuming that the code of interest is near to the entry point defined by the user. The other main way we plan to avoid code exploration issues is to leverage the flexibility of Ghidra: we will let the user define multiple entry points manually. If complex code prevents our analyzer from exploring the program properly, we want to enable the use of Ghidra's reverse engineering features so that the user can help the analyzer.

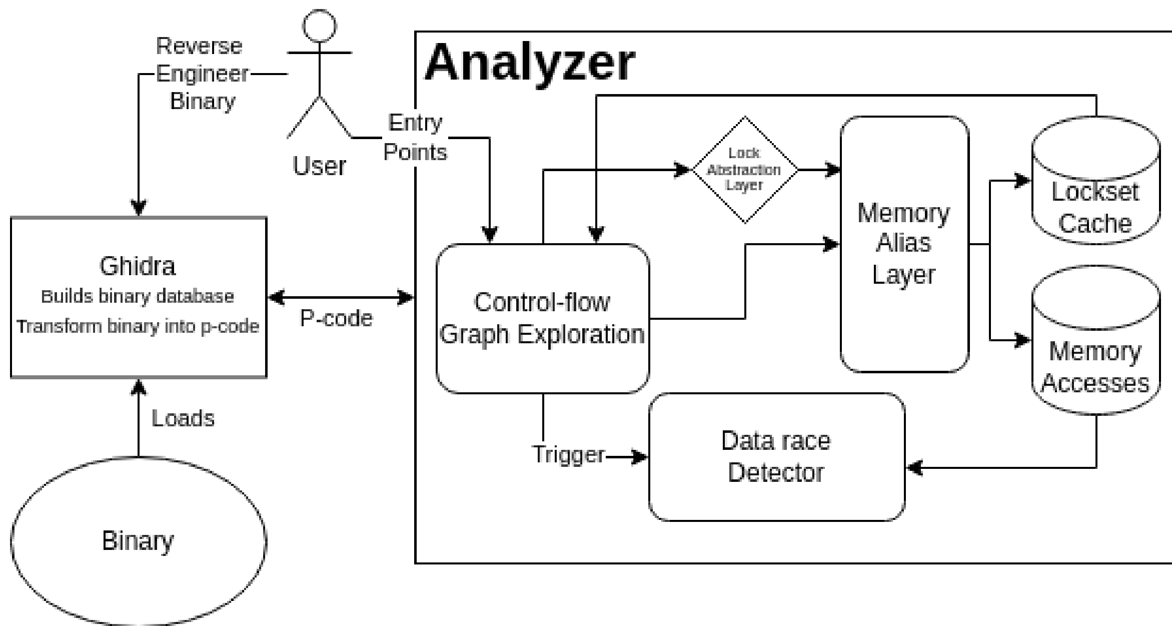

Figure 3.1: Analyzer system design

3.1.1 Detecting locking operations

Since our objective is to build a lockset, we must identify the different locking operations possible and keep track of the current lockset while exploring the program's control flow graph. Since programmers use synchronization mechanisms various way, such as by the use of different locking libraries, we will have an abstraction layer that detect when a lock is acquired. This abstraction layer will detect what type of program is analyzed (can also specified by the user) and will identify locks being acquired based on the locking mechanism used in that type of program. For example, we expect this layer to understand the *pthread* function *pthread_spin_lock* used to acquire locks in Unix programs. The only thing that needs to be changed when using a different system is adding new abstraction layers. These helps avoid having to analyze low-level code from third-party libraries, thereby simplifying the analyzer. Part of the job of this layer is also to normalize how the lock is represented. Our analyzer expects a single variable representing the lock, such as by parsing the parameters passed to the locking function call.

Once we have identified the *varnode* representing the lock, we must address the pointer aliasing problem. Two locking operations may act on different varnodes that alias to the same memory location in the program. To handle this, we use the *Memory Alias Layer* which attempts to detect whether two *varnode* represents the same memory location. This is explained in the next section on memory accesses.

3.1.2 Detecting memory accesses

While RacerX starts by building a lockset cache for each function visited in a first pass, we decided to create both the function lockset cache and the memory access cache during the same pass. We choose this approach to limit our analysis to only one pass for performance reason. We plan to register each time a memory location

(denoted by *varnodes* in Ghidra) is accessed, identifying it as an input (read) or an output (write) of an operation. The memory location and the details of the operation are also saved in a list. Whenever a new memory access is registered, we run the race detection algorithm which tries to generate alerts when two memory access may create a data race. We designed this data structure so that it can quickly detect data races: we will only have to compare a memory access with other memory accesses that are from different threads, point to what we suspect are the same memory locations and make the comparison only when at least one of the accesses is a write. This allows the race detection algorithm to scale well, despite the large number of expected memory accesses occurring in a standard program.

Once again, one of the most difficult issues that we will face while detecting memory accesses is the pointer aliasing issue. While the use of some memory locations in a program, such as global variables, can often be determined easily by looking at the program headers, the use of other dynamic memory locations can be more difficult to identify. Ghidra's p-code handles the creation of *varnode* variables representing a unique memory location, but Ghidra's decompilation is intra-procedural. We designed our analyzer with a *Memory Alias Layer* to detect whether a particular memory access is an alias to a previously observed memory access. This layer analyzes each memory access detected using a set of heuristics and determines if a previously seen memory access is an alias to this new memory access and tag the new memory access with this information if this is the case. These heuristics may perform extra-procedural analysis, such as basic control flow analysis to match a function's parameters to a previously seen function call. This layer also uses information specified by the user, such as variable names and defined structures to determine when two memory accesses are aliases.

3.1.3 Gathering thread information

Our analyzer needs to identify which memory accesses are executed on the same thread and which memory accesses could be concurrently executed by different execution thread. To achieve this goal, we need to keep track of when new threads are created. Since threads are based on specific operating systems system calls, our design simply monitors for the invocation of these pre-defined system calls or library functions that are known to initiate the creation of a new thread. We also let the user specify entry points for starting the analysis, each entry point is expected to be running concurrently. While tracking the creation of new threads, our analyzer also tracks the "happens-before" relationship between the memory accesses identified. A memory access A "happens-before" another access B and cannot be part of a data race, if the access A is executed before the creation of the new thread which executes the memory access B. We can be certain that two memory accesses having an "happens-before" relationship will never lead to a race condition, as the A is executed before B is created.

3.1.4 Detecting race

Since our design only uses a single control-flow pass, when a new memory access is registered, we verify if this access is a potential data race. This is done in the *Data race detector* module, which verifies whether the new memory access conflicts with any of the previously recorded accesses. This is done by quickly filtering the previous accesses: only accesses from other threads, without a *happens-before* relationship, where at least one of the accesses is a write operation and where both accesses are alias causes the generation of a data race alert. This is considered one of the most computationally expensive parts of our analysis and the race detector module is designed to reduce the amount of computation done as much as possible by filtering out as many memory accesses as early in the process as it can.

3.2 Walkthrough

In this section, we will describe the final algorithm used by our analyzer and walk through each step individually.

Final algorithm

Algorithm 1 Exploration algorithm

Require: entry functions

```
for entry functions do traverse_function(f)
    if (cachedLocksets = functionCache.get(f)) != null then return cachedLocksets
    else if detectRecursion() then return ∅
    else traverse_blocks(f.getFirstCodeBlock())
    end if
end for
```

1. The binary is loaded into Ghidra, and basic analysis is done. These analyses will disassemble the code sections and identify the functions in the binary.
2. (Optional) The user can take time to reverse engineer the binary and rename desired functions or variables within Ghidra. Some of these changes, such as naming different variables with the same name when they are aliases to the same memory location, can potentially help with the analysis. Furthermore, issues in the disassembly, such as missing function definitions, can also be manually fixed here.
3. The user specifies the different entry points for the analysis. All of these entry points will be independently explored and are considered to be executing concurrently.
    (a) The initial function is decompiled and traversed, codeblock by codeblock.

When there are multiple code paths, each is followed independently. Each codeblock is composed of a series of decompiled *pcode* operations.

i. If the p-code operation is a locking operation, change the current lockset
ii. If the p-code operation creates a new thread, start another control flow graph exploration from the new thread's entry point
iii. If the p-code operation is a new function call, traverse the next function
iv. When the p-code operation is a memory access, register it. When the memory access is registered, possible data races are detected and alerts are created

(b) When the last codeblock is explored, add the current lockset to the function lockset cache to avoid unnecessary reanalysis of the function.

4. When all entry points have been explored up to a user-specified depth, print all data race alerts and registered memory accesses

3.2.1 Using the tool

As the analysis is built as a Ghidra script, the first step is to load the binary into Ghidra and execute the auto-analysis of Ghidra to populate the database. This auto-analysis determines information about the binary required by our analyzer, such as entry points of the different functions of our program. As our analyzer starts by looking up the functions specified as entry points by the user, the only required analysis by Ghidra are the ones that find and define the functions found within the binary. Other analysis such as searching for defined strings or referenced data are not required for our analysis.

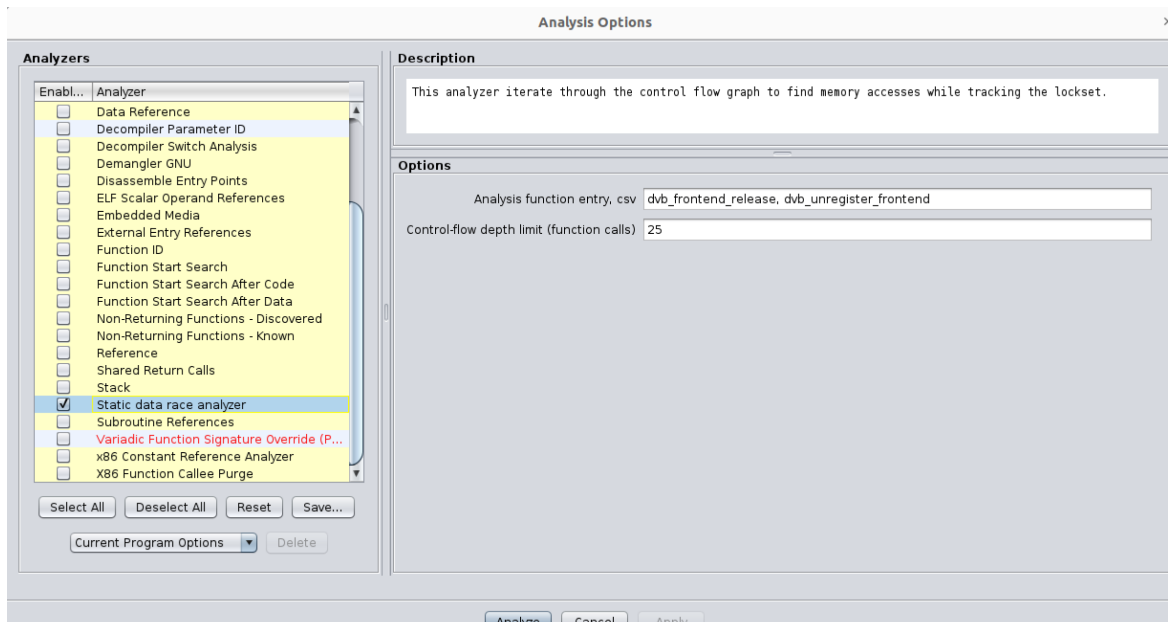

Figure 3.2: Launching menu for the analysis

As seen in the menu to launch our analyzer in figure 3.2, the user needs to specify the name of the functions to be used as the entry points for the analysis. Our analyzer will start a control-flow exploration starting from each of these specified entry-points and assume that these functions are executed in different threads. This lets the user be precise on which functions to analyze if they already know these functions are executing concurrently. Otherwise, they can specify an entry point earlier in the execution flow of the binary (such as the program entry point) to run an analysis thorough the whole program.

Another significant factor on the accuracy of our analyzer is the quality of the function definitions recovery by Ghidra. In this situation, the user can modify these definitions manually and rename function parameters to inform our analyzer that different parameters point to the same memory location. More precision function parameters and function definition will lead to better results, such as fewer false negatives as the user could specify a relation between two parameters that our analyzer would miss.

At the end of the analysis, our analyzer will return alerts specifying the address of the p-code instruction interacting with the shared memory location and a list of all other conflicting memory accesses found in other threads. The alert also includes the name of the function containing the instruction and the specific p-code operation leading to the data race. As Ghidra transforms the binary disassembly into p-code, the best way to find the assembly instruction is to browse to the specified address in the binary and Ghidra should display side by side both the p-code instruction and the assembly instruction.

For example, the results of the analysis printed in the console can be seen in listings 3.1 and 3.2. In this case, a memory access in the *updater* function, which runs on its own thread, could be conflicting with the memory accesses executed by the *reader* thread. In this specific example, the varnode *(ram, 0x104020, 4)* represents a global memory variable being accessed by different threads without synchronization and an alert is generated only once for this set of memory accesses. An analyst could start their analysis at the instruction found at address 0x104020 in the disassembly view of Ghidra to see the assembly instruction causing this memory access.

Listing 3.1: analysis logs

```
[Found possible data race]: Memory Access faulty = MemoryAccess:{updater,(ram, 0x104020, 4) INDIRECT
    (unique, 0x10000097, 4) , (const, 0xbf, 4),00101480,(ram, 0x104020, 4),true,[]} HappensBefore:
Thread:[Thread:updater,001015d9]

Source:null

Conflicts with:
MemoryAccess:{reader,(register, 0x10, 4) INT_ADD (ram, 0x104020, 4) , (register, 0x0, 4),001012f8,(
    ram, 0x104020, 4),false,[]} HappensBefore:
Thread:[Thread:reader,001015bc]

Source:null

MemoryAccess:{reader,(unique, 0x10000074, 4) INT_MULT (ram, 0x104020, 4) , (const, 0xffffffff, 4)
    ,00101310,(ram, 0x104020, 4),false,[]} HappensBefore:
Thread:[Thread:reader,001015bc]

Source:null

MemoryAccess:{reader,(register, 0x0, 8) INT_SEXT (ram, 0x104020, 4),00101360,(ram, 0x104020, 4),
```

```
    false ,[]} HappensBefore:
Thread:[Thread:reader ,001015bc]

Source:null

MemoryAccess:{reader ,(register , 0x30, 8) INT_ZEXT (ram, 0x104020, 4),001013bb,(ram, 0x104020, 4),
    false ,[]} HappensBefore:
Thread:[Thread:reader ,001015bc]

Source:null
```

Listing 3.2: statistics logs

```
Number of Memory Accesses:300
Time(seconds) spent=14.743 (start=1711579582718 end=1711579597461)
Stats: INT_SLESS:14
BRANCH:6
INT_ZEXT:1
CALL:82
INT_SREM:14
LOAD:14
COPY:37
INT_SEXT:9
RETURN:30
INT_MULT:2
STORE:5
INT_ADD:56
INT_EQUAL:8
INDIRECT:8
TotalWrite percent = 0.24333333333333335 (73/300)
Used memory is bytes: 113496712
Used memory is megabytes: 108
INFO  ------------------------------------------------------------
    Static data race analyzer                  14.748 secs
------------------------------------------------------------
     Total Time   14 secs
------------------------------------------------------------
\label{lst:stat_logs}
```

Finally, to confirm whether the alert is a false or a true positive, the user can conduct further code analysis to confirm that the memory access is not synchronized with the other threads and that the accessed memory is shared. The difficulty of this analysis depends on the program, however the alert can point the analysis towards an area of the code where the analyst could notice a lack of explicit synchronization primitives. Another approach would be to instrument the binary or the system if possible and to try to trigger the race manually by introducing delay to specific threads or by adding breakpoints to the accessed location. Nevertheless, identifying if the alert is a false positive is easier with a specific alert pointing to a specific instruction.

3.3 Conclusion

Our data race analyzer design is built on top of the Ghidra software reverse engineering framework. This allows our analyzer to have access to the disassembly and decompilation work done by Ghidra and to use a single IR (p-code) for our analysis. This lets us analyze all binary architectures that are supported by Ghidra, offering flexibility to the user by allowing them to modify the binary database in order to get more details or to correct errors. The proposed design is a single pass algorithm that explores the specified entry points once while performing detection of locking operations and memory accesses in the binary. Our design uses caching to avoid the multiple analyses of the same function or codeblock. While exploring the binary, a list of data race alerts is generated and saved. The resulting design is an analyzer that should support all commonly used binary formats and systems, while offering maximum flexibility to the user.

Chapter 4

Implementation

This chapter describes how we implemented the design described in chapter 3. We chose the Ghidra framework to convert the target binary into a more suitable IR for our analysis. Ghidra provides a scripting API to interact with the binary and with models representing its characteristics that were built by Ghidra. The convenience of this interface and the support for multiple computer architectures were the key factors in our decision to use Ghidra in our implementation. At the time of writing, the latest public release of Ghidra (version 11.0.1) has been selected for our implementation.

In this section, we will provide an overview of the different components of our implemented architecture. For each component, we will describe its implementation and the associated trade-offs made. We will then use this opportunity to discuss how the various stages of our analysis interface with each other, which is an especially important part of our analyzer.

4.1 Components

Our implementation consists of the following five components:

1. Executable parsing,disassembly and control-flow graph reconstruction

2. Decompilation
3. Control-flow graph exploration
4. Lockset construction
5. Memory access identification
6. Data race detector

Some of these components have been implemented by using Ghidra's features. Thanks to our design decisions, most of these components are architecture agnostic. However, since binaries are architecture-specific, some of these components are specifically designed to abstract away these details from the rest of the analysis.

4.1.1 Executable Parsing, Code Disassembly and Control-Flow Graph Reconstruction

The first part of our analysis involves ingesting a binary program and identifying its code segment for analysis. Although analyzing common executable formats such as ELF and PE files may seem relatively straightforward, supporting a wide variety of executable formats can be challenging. These executable formats usually contain markers to inform the program loader where each section of the program resides in the binary file. Determining the location of the data and code sections is a prerequisite for analyzing the program, since it allows us to identify the program entry point and the location of global variables. Recovering the location of these sections can be challenging due to the wide variety of compilers used and due to the use of obfuscation in certain executables.

One of the core objectives of Ghidra, as a binary analysis platform, is the safe and efficient handling of a diverse set of binaries. Ghidra supports over 35 architectures

and implements multiple heuristics to try and determine the architecture of the given binary automatically. A command-line interface is also available to ingest binaries in a programmatic way.

Once a binary has been successfully loaded, several analysis options become available. These allow us to start dissecting and augmenting the binary and to add supplemental information to the analysis. This allows the analyst to modify parameters and for example, be more aggressive in disassembly of possible data or in function identification.

A very important component of our analyzer is the creation of a control flow graph using Ghidra. This process identifies and creates entries for each function in the binary. It also builds a call graph between each function, which allows our analyzer to easily follow the different code paths in the binary during its execution.

To construct the control flow graph of our binary, we import it into Ghidra and use it to detect its target architecture. We then execute multiple analyzers provided by the Ghidra default library to identify the functions contained in the binary. Finally, we disassemble the instructions to recover the control flow of the program. This information let us determine how the execution threads progresses through the program, such as jumping into different functions or taking different conditional jumps.

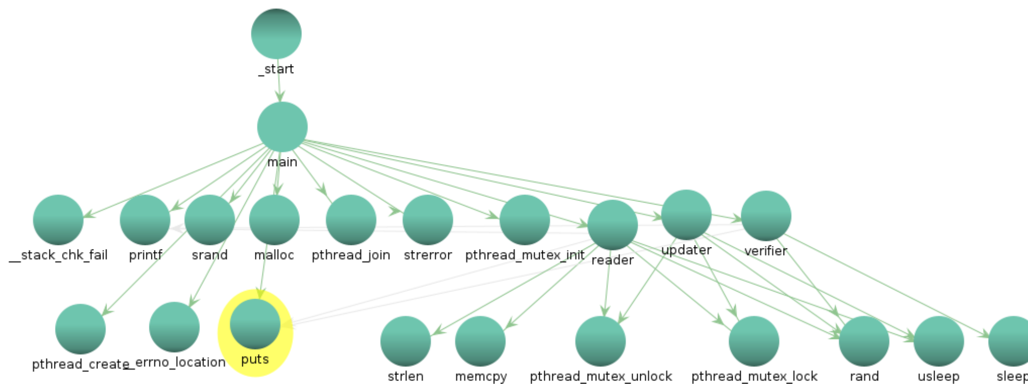

Figure 4.1: A function call graph created by Ghidra, which shows what function can be called from another function

The resulting function call information collected by Ghidra can be represented as a function call graph, showing which function can be called from another function, as depicted in figure 4.1. In this example, the function reader, updater and verifier can all be called from the main function. We can also notice the locking-related functions (pthread_mutex_lock and pthread_mutex_unlock) are also called from these functions.

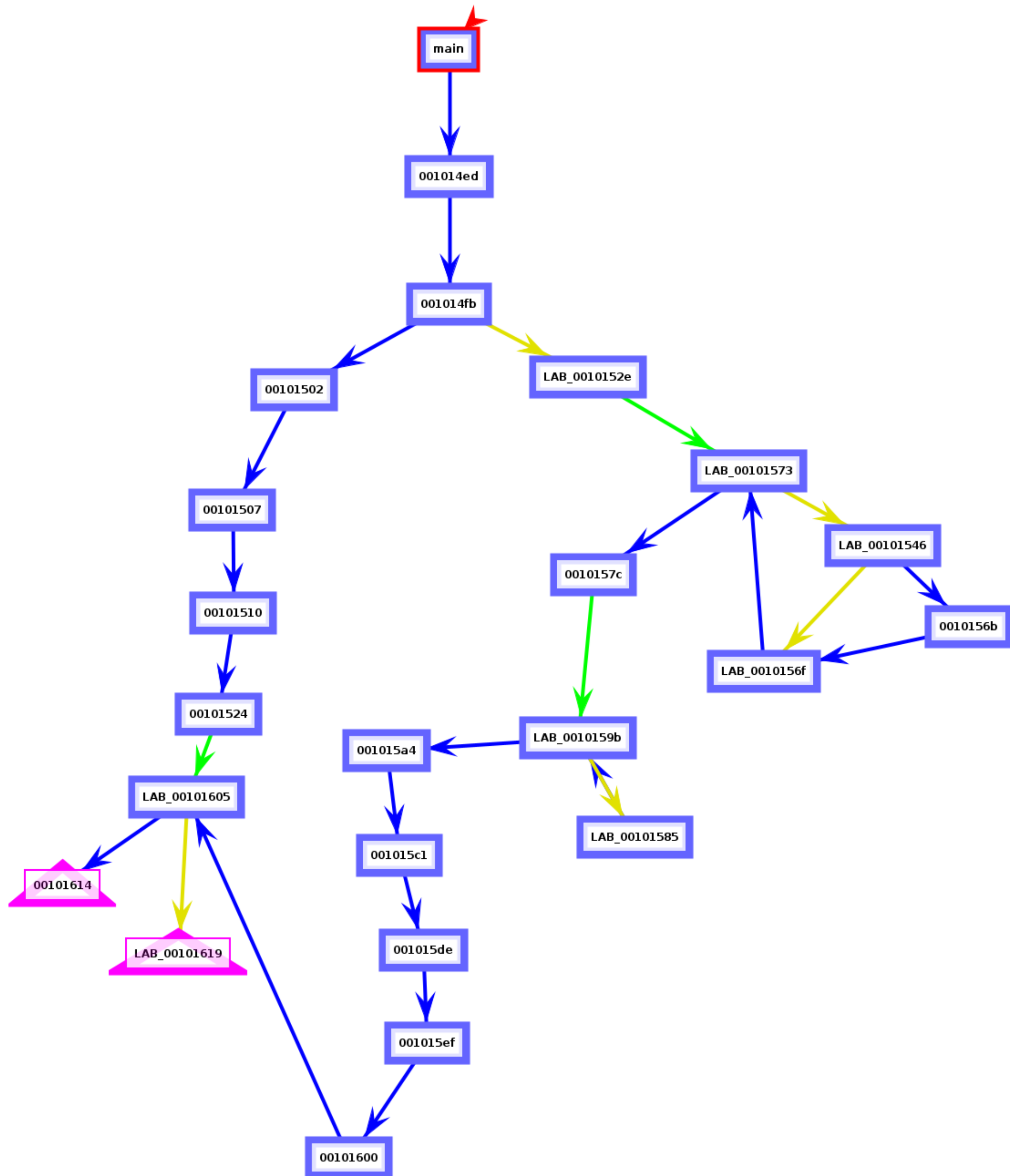

Figure 4.2: A codeblock graph created by Ghidra. The graph shows the flow of different codeblocks in a function

With the recovered control flow, Ghidra can also build a code block graph to describe the inner control flow inside a specific function. In figure 4.2, we can see the code block graph of the main function. Each code block represents a group of instructions that are continuous to each other without any changes in control flow. Arrows leaving a code block represents when an instruction has an impact on the control flow. In this example, we can see the branching created by an if/else statement in the main function.

This component of our analyzer has very solid results. As shown in chapter 3, Ghidra has a very good performance at detecting code and disassembling it. Performance-wise, larger binaries take a long time in this component, as Ghidra must explore many possible code paths in order to identify functions and construct a control-flow graph. When analyzing obfuscated or complex binaries, the analyst must take care that this component returns adequate results in function identification and code disassembly so that the rest of the data race analyzer can traverse a sufficient amount of the available code path in the binary.

4.1.2 Decompilation

A very important ability of Ghidra is its ability to decompile binary back into a higher-level representation. To do so, Ghidra converts the disassembly into p-code instructions as described in chapter 3. As decompilation is computationally intensive, it is normally only performed when a section of the code is being analyzed by the user. To enable a consistent analysis across different computer architectures (32-bit or 64-bit) and against different CPU instruction sets (x86, ARM and others), we use Ghidra to convert the binary representation into the p-code IR described in the previous section. We use the official Ghidra Java API to interact with the decompiler and initiate decompilation of the desired functions.

When using the scripting API, our analysis must first initialize the *DecompInterface*

with the desired simplification style. The decompiler's simplification style specifies the degree of simplification performed by the decompiler when translating the assembly code into p-code. In our analysis, we use the *normalize* simplification style, which omits some of the final decompilation steps but performs basic optimizations to simplify and normalize the resulting p-code. This allows our implementation to have a normalized p-code syntax tree for easier analysis, while keeping our p-code representation closer to the original machine code instructions.

When interacting with a function in our analyzer, we always use the *HighFunction* object. This object represents a higher-level abstraction function based on the information produced by the decompiler. To obtain this object, we pass the function to the *DecompInterface* and extract the *HighFunction* from the decompiler's results. Since this interface requires a timeout value, our current implementation uses a relatively large timeout value of 50 seconds. Any decompilation that takes longer than this fails, and our analyzer assumes that the function cannot be analyzed. The decompiler is very robust, and only external function stubs have failed to decompile successfully, which is of no consequence to our analysis.

When decompiling machine code instructions into p-code instructions with our previously specified decompilation configuration, the resulting IR is not based exclusively from the instructions of the original binary. Some instructions, such as the *MULTIEQUAL* and *INDIRECT* p-code operators, are added by the decompilation process to provide additional information about specific varnodes.

For example, the instruction *MULTIEQUAL(input0, input1, output)* indicates that the value of *output* can originate from either *input0* or *input1*. This is equivalent to a *phi-node* from the *Static Single Assignment* form commonly seen in compiler theory. The *MULTIEQUAL(input0, input1, output)* is used to specify that the value *input0* may be altered in an indirect way by the operation referred by *input1*. An example of Ghidra's decompilation into p-code operations can be seen in figure 4.3 below. The

variable *lock*, which is in this example a global variable, is represented in the p-code instructions as a specific address *0x104040:8*. The effects of the x86 *CALL* instructions are converted into multiple equivalent p-code instructions: *INT_SUB* on the stack register to grow the stack downward for the function call, followed by the *STORE* to prepare the following *CALL* instruction used to call the function. The function is specified as *EXTERNAL*, because *pthread_mutex_lock* is a call to a library loaded dynamically by the program.

```
0010167e 48 8d 3d       LEA      RDI,[lock]
         bb 29 00 00
                                          RDI = COPY 0x104040:8
00101685 e8 a6 fb       CALL     <EXTERNAL>::pthread_mutex_lock
         ff ff
                                          RSP = INT_SUB RSP, 8:8
                                          STORE ram(RSP), 0x10168a:8
                                          CALL *[ram]0x101230:8
```

Figure 4.3: An example of Ghidra's disassembly on the left coupled with Ghidra's p-code on the right.

The main output of this decompilation process is the creation of the *HighFunction* objects to be used in subsequent phases of our analysis. A *HighFunction* provides all the information related to a function that is normally of interest to our analyzer. We can obtain the original *Function* object and its basic information collected by Ghidra in the previous step, such as the function name, either generated or found using symbols, and location in the binary. The higher-level of abstraction provided by the *HighFunction* object lets us obtain an iterator over the function's p-code abstract syntax tree. These p-code instructions are all related to a particular *CodeBlock*. Using these iterators and these *CodeBlocks* are the main way we explore the code in our next section about control-flow graph exploration.

4.1.3 Control-flow graph exploration

The first important phase of our data race analyzer is to develop an exploration algorithm to explore as many code paths that could happen during a normal execution of the binary. This exploration is important to build the different locksets used in the following phases of our analyzer. Additionally, having good coverage is necessary to track the creation of a new thread and the location of its entry point, since our exploration algorithm will use these entry points to continue our analysis. Because of the exponential growth in possible paths, we chose and implemented some heuristics to have our analysis end in a respectable amount of time while exploring most of interesting code paths in the target program.

To begin our graph exploration, we need an entry point to start our analysis. While the binary entry point specified in binary format is an interesting default option, we added the ability to specify a different entry point for our analysis. To do this, we added our analyzer to Ghidra's analysis menu, allowing the analyst to start the analysis from Ghidra's user interface. Additionally, they can provide parameters for the chosen analyzer. This is how we let the user provide a list of memory addresses that are used as different entry points for our program. If no entry points are specified, our analyzer will determine and use the default entry point from the binary database built by Ghidra in the previous steps. Usually, the *_start* function is identified and represents the entry point of the program. After some program initialization is done, the *_start* function will call the more well-known *main* function. Our approach allows flexibility, allowing the analyst to focus on specific areas of interest if desired while also being easy to use by anyone. Using the analyzer interface in Ghidra also allows us to use other features such as giving the analyst feedback on the progress of the analysis and to send dynamic messages using the graphical user interface.

When following the control flow of a function, we use the code block graph built by Ghidra. To explore this graph, we use a DFS algorithm. We chose this algorithm

because it allows us to easily keep track of which locks are currently held by storing them on a stack. A few issues we encountered when trying to explore a control-flow graph were the presence of loops and the *"branching off"* problem. The *"branching off"* problem occurs when the control flow of the program branches off into a commonly called location, such as in a commonly used system library. A lot of analysis time can be wasted by branching off into a commonly called logging function, which itself branches off into more and more irrelevant functions. To deal with both of these issues, one of the heuristic we use is to limit the depth of our DFS. The maximum depth value is manually chosen depending on the required performance and complexity of the targeted program. A higher limit makes the analysis slower but more precise, however a low limit might make the analysis miss interesting code paths.

Our search algorithms also detect when new functions are called. When exploring a function, we aim to gather information about its effect on the lockset and to find occurrences when shared memory is accessed. To avoid re-exploring functions when we know their effects on the current lockset, we use a caching technique similar to the technique described in the RacerX research paper[8]. This technique involves storing in a cache the effects of a function call on the lockset. When a similar function call is encountered, we can avoid re-analysis by looking up the changes to the lockset in the cache. We implemented this cache with the use of a HashMap, the Java implementation of a hash table data structure, mapping each function previously explored to another HashMap. This inner HashMap maps each entry lockset onto an output lockset. When a value is found in the hash tables, the function is not explored and the returning lockset value is used. If a value is not found, the function is explored and changes to the lockset are registered and are saved into the cache when the function returns. We extended this technique by also caching code blocks, since the instructions are divided into a set of unique code block and because the same code block can be analyzed multiple time while in a loop. This let us speed up our analysis considerably.

While searching our graph, we face branching instructions such as an if/else code block. Since analysis is path sensitive, a branching block will lead to our analysis to also branch off into two paths. To implement this easily, we use recursion to start two new graph searches starting from the two output nodes from our branching node. This introduces the possibility of an exponential branching of analysis, but was not an issue thanks to the use of caching. Because our analysis is not interprocedural, we assume both branching statements could be executed by a function call and the effects on the lockset of both branching analysis are added to the function cache.

While search thorough the program with our DFS algorithm, we keep track of system calls or functions used to create new threads. Managing threading is the responsibility of the operating system and is typically managed by a set of system calls. Because it is impossible for our tool to determine the effect of a system call or of a function, the system calls or functions to be tracked have to be manually chosen. In our current analysis of Linux ELF binaries, our analyzers recognize the basic functions defined by the pthread library. These functions are used to create threads or terminate threads. They also contain the functions used to create new locks and to acquire or release them. We use this knowledge of the pthread library to track which locks are currently acquired by the program. Function calls to *pthread_create* are analyzed to find the thread entry points. Whenever a possible creation of a new thread is found, we start over our DFS from the entry point of this new thread. To avoid unnecessary computation and to avoid infinite loops, our analysis only does a graph search once at a specific entry point.

Analyzing a code block

When we analyze a codeblock made of this normalized p-code, our analysis iterates through the p-code instructions to find a *CALL* operation represented by the class *p-codeOp*. From this *p-codeOp*, we can obtain the *Varnodes* used as input parameters.

As explained in chapter 3, a *Varnode* is a generalized representation of a variable. In our analysis, we look at the *Varnode* associated with the destination of the *CALL* operation and obtain from the *Varnode* its address in the program's memory space. From this address, we can easily verify if the destination is a known function with Ghidra's *FunctionManager*, an API used to query Ghidra's knowledge about analyzed functions.

If the destination of the *CALL* is to a function of interest, such as a function creating a thread, we have to analyze the call to find the entry point of the new thread. This depends on the function's implementation and the function parameter of interest has to be specified. In our analysis against Linux binaries, to determine the starting routine of the new thread the arguments of *pthread_create* were analyzed. In this case, the third parameter of *pthread_create* represents the address of the initial function for the new thread. Its address can be obtained in a similar way by obtaining the *Varnode* representing the third input to the *CALL* operation and by obtaining its address in the program address space. This interface is also used to know the destination of a CALL instruction or the parameters of other instructions. As we are using p-code instructions, these values should never change for the p-code instructions themselves. However, when analyzing platform-specific functions, such as functions from locking libraries (e.g. pthread_mutex_lock), we have to handle platform-specific behaviours.

When analyzing some call functions in our exploration, we will need to handle different platform-specific behaviours. For example, the commonly used concurrency libraries used to create threads are not the same on different platforms. As such, we developed an abstraction layer to handle the detection of threading logical operations. We classify known functions into a special ThreadOperation object, a ThreadOperation contains a logical operation enum, an optional targeted lock and an optional targeted address. We have enumerated the following logical operations:

1. CREATE_THREAD This logical operation represents a function that ends up

creating a new thread. The resulting ThreadOperation contains the entry point of the newly created thread.

2. ACQUIRE_LOCK This represents the acquisition of one lock, the ThreadOperation contains the targeted lock.

3. RELEASE_LOCK This represents the release of one lock, the ThreadOperation contains the targeted lock.

4. EXIT_THREAD This logical operation represents a function who ends up exiting the thread. The resulting ThreadOperation contains only the logical operation enum.

5. NOP This logical operation represents a function who has no effect on threading.

Using this layer of abstraction to classify functions into a ThreadOperation, we can keep platform-specific code to handle the identification of thread-related functions inside their own modules. This allows our graph exploration code to be target agnostic. In the same level of abstraction, we keep a list of blacklisted functions to avoid exploring if encountered during the graph search. These functions are known to be benign and to lead to a lot of wasted searches. Examples of such functions are logging functions or memory management functions. The use of a blacklist is an easy way to avoid known bad functions on specific platforms.

With our code exploration algorithm, a thorough exploration of the specified entry points is done and of the newly created threads found during the exploration. We have implemented multiple heuristics to improve the performance and to abstract away platform-specific behaviours. As the construction and exploration of a control flow graph are complex, there is always the chance that the graph exploration takes a long time for the analyzer. As such, it is important in our implementation to have the heuristics parameters available to the analyst for modification. Our implementation

has the advantage of being a path sensitive interprocedural analysis, leading to very good code coverage. We, however, risk having our analyzer spend too much time unrelated code path, as our heuristics will not always detect cases when we need to abort the unneeded search.

Lockset Construction

The main objective of following the code flow of our program is to build a lockset representing the locks or other synchronization primitives being held at each shared memory access done by the program. This is part of our data race finding algorithm where two shared memory access to the same region without any lock shared in their respective lockset is detected as a possible data race. A lockset is a set of currently acquired locks, where a lock is represented by a *Varnode*.

While we explore the control flow graph, we keep track of various locksets by keeping them inside a FlowState object. This object's goal is to keep track of the current state of our exploration, such as the current lockset state, but also information such as the current thread and the initial lockset from when we started the search.

A lockset is defined as a Java HashSet containing objects of the type Lock. The use of this Java class simplifies our implementation and guarantees the absence of duplicates in members of the HashSet. This is guaranteed because in the Java HashSet implementation, the addition of an object to the set returns a false value if this object was already in that set. This is accomplished by comparing the hashcode value of the objects in the set and of the object to be added. Our Lock class was created to augment the Varnode class with the addition of a valid *hashCode*, *toString* and of a *equals* methods. This allows our Lock class to be compatible with other Java data structures and to work with the Java HashSet implementation. This prevents the presence of duplicate Lock object representing the same Varnode.

Memory Access Identification

Our algorithm to detect data race depends on detecting concurrent access to the same memory location. As described in section *Background* the third major difficulty of program analysis is trying to do pointer aliasing to determine when two pointers point to the same location.

To find memory accesses during our control flow graph exploration we inspect every p-code operation related to data manipulation. Because the p-code IR is much simpler than the ISA (instruction set architecture) commonly seen in modern binaries we only have to look for a small set of p-code instructions. The output and input of these memories accessing instructions are once again *Varnodes*, the generalization of registers and memory for Ghidra's decompiler.

4.1.4 Data Race Detector

The goal of the data race detector is to generate alerts when a memory access is suspected of potentially causing a data race. An alert contains a reference to the specific memory access and a reference to a list of all other memory accesses that potentially overlap with it.

Hashmap are used to separate every memory access based on their execution thread. This allows us to quickly obtain a list of all memory accesses executed by each thread, and compare them with each other. As mentioned in our Design section, our algorithm quickly filters out most memory access by verifying that at least one of them must be a write operation. Furthermore, we also verify the *"happens before"* relationship between the memory access: when a new thread is created, all previously seen memory accesses are tagged as *"happening before"* this new thread. A data race cannot happen if the memory access is known to only occur before the creation of the thread. Our algorithm also verifies that the varnodes do not intersect according to our decompilation process.

Finally, our detector verifies if both memory accesses share a lock by calculating the intersection of their respective locksets. If both memory accesses intersect and do not share a lock in their lockset, we register this access as a possible data race.

In this part of our detection algorithm, we also use the annotated information provided by the analyst. We verify the name assigned to different variables and function parameters by the analyst. If different varnodes have the same name, our analyzer assumes that these varnodes refer to the same memory location and continues the analysis accordingly.

Finally, for each detected memory access conflict our detector tries to generate a new alert. Our objective is to gather as much information as possible about possible data races, but to only generate alerts that are useful for the user (typically by triaging the list of memory conflicts). Before generating a new alert, our detector verifies whether a previously generated alert also covers the same memory accesses. We assume that a lack of synchronization will lead to multiple data race alerts around the same memory location, therefor we only need one alert to guide the analyst towards the possible data race bug.

The data race detector is also responsible for collecting statistics about the analysis: number of memory accesses tracked, which operands were used and the number of execution threads that were detected. As we slowly build up the list of memory accesses accumulated while analyzing the program, this allows us to provide feedback to the analyst over time about the progress of the analysis.

We were able to keep this component relatively simple by implementing a very thorough memory access system. All of the information needed to compute the possibility of a data race is contained in the data collected about the memory accesses that occur throughout the program.

Chapter 5

Evaluation

In this chapter, we evaluate the performance of our implementation against a testing set of previously identified vulnerabilities. Our goal is to analyze the execution time and the ability of our analyzer to identify previous data races in the test set. We also conduct this evaluation to gain a deeper understanding of how our analyzer performs against real data races. When performing this evaluation, we will focus on understanding each step of our analysis against the target binary. This section will also be used to explain and describe how to configure and use the analyzer with new binary.

In section 5.1, we describe the methodology that we used in our evaluation. We then present the targeted binaries and the chosen data races that we will be trying to rediscover with our tool in section 5.2. This section describes in detail the testing cases that we have chosen for our evaluation, explain their complexity and how they work. In the next section, we will also describe how the analyzer is configured to successfully find these data races. We conclude our evaluation with section 5.4, which explores the results of our analysis on the testing set, including performance statistics and a detailed explanation of the more interesting analysis results made by our analyzer.

5.1 Methodology

The evaluation focuses on two major aspects: the performance of the analyzer and its ability to generate relevant alerts for the user. To investigate the performance of the analyzer, we will evaluate each stage of the analysis using a Java profiler. With the help of the profiler, we can determine how much time was spent in each phase of the analysis. We will also collect statistics about our analysis, such as the number of memory accesses tracked and code blocks explored. To evaluate our analyzer's results in detecting data races, we will look at the alerts generated and verify whether the detected memory accesses are related to the known data race. If so, we consider the alert a true positive. Otherwise, we consider it a false negative.

To begin our analysis, some configuration of the binary in Ghidra is required. The binary needs to be loaded into Ghidra by importing it into a Ghidra project. After a binary has been imported; we need to run Ghidra's auto-analysis on the binary in order to extract the information we need to run our analyzer, such as the disassembly and identification of functions found in the binary. We do not consider the time spent loading the binary in Ghidra as part of our analysis. We tried to minimize the size of the analyzed binary while possible, such as by configuring the Linux kernel to minimize its size while keeping the relevant code.

Because our analyzer is integrated inside Ghidra and since Ghidra is a mature reverse engineering framework, we can also take the opportunity to do manual reverse engineering. For example, we can rename functions, specify the type of a variable, identify new functions missed by Ghidra, or even modify the disassembly done by Ghidra. All of this information is available to our analyzer and can be very helpful in making the analysis more precise or informative. For this evaluation, we took the time to rename the analyzed function and to specify the type and name of parameters passed to the entry point functions.

5.2 Testing set

To conduct our evaluation and to develop our analyzer, we wrote four basic C programs. The first program, called *program A*, is a basic C program using threads and contains a basic data race where multiple threads access and modify the same global memory buffer. The *program B* binary is the same as *program A* but uses locks to prevent the data race from occurring. The third program, *program C*, is an example of a more complex data race. In this program, multiple threads are created and some threads are initialized by other threads. A data race occurs when multiple threads update the same memory object protected by locking operations containing a mistake. The last program is *program D*, similar to *program C*, but the locking operations have been fixed to prevent the data race. We have included programs that have fixed data races to test our analyzer and measure the number of false positives it produces.

For our testing set of real data races found in modern software, we chose two CVEs found in the Linux kernel. We chose to focus on analyzing the Linux kernel, as it is an open-source project where we can easily find detailed information about the fixed data races and how they were patched. While our analyzer enables analysis of closed source programs by targeting binaries, the open-source nature of Linux helps us evaluate our analyzer since we can better understand how the code works and why the data races happen.

5.2.1 Program A and B

The goal of this program is to have an easy example of a data race to analyse and to test our analyzer. This program consists of three threads: *main, reader*, *updater.*

The *main* thread will initialize a buffer in global memory with an index value and a length. The thread *reading* will start by generating a random number determining the read size. It will then verify if the number of characters to read will overflow the

buffer based on the current index. The read operation should only happen if it does not go outside the bounds of the buffer. After the operation, the result of the read operation is verified to detect an out of bound read.

Meanwhile, the *updater* thread will simulate another reader by advancing the index forward by a random value and reinitializing it when the end of the buffer is reached. Since there is no synchronization between the *reader* and *writer* threads, a data race occurs when the thread *reader* checks if the read operation would overflow the buffer, followed by the index being advanced by the *updater* thread. The following read operation would now overflow the buffer since the index value has been modified.

5.2.2 Program C and D

This program also contains a data race but is a more complex example than *Program A*. This program starts with the main function initializing an object containing pointers to other objects allocated in memory. These objects will all contain a reference to a string buffer and a read index. Following the object's initialization, the program will be creating two *creator* threads. These threads will then themselves create multiple *writer* and *reader* threads. These threads will behave similarly as the threads in *program A*, they will randomly pick one of the previously initialized objects by derefencing the global variable pointing to these objects. They will then either read or write to the buffer randomly while respecting the buffer size limit. While some of these operations are protected, some read and write operations lack locks. When the data race happens between a *writer* and *reader* thread, we expect them to access the buffer out of bounds because of a race between verifying that the read operation fits within the buffer and another thread modifying the buffer index.

To successfully analyze this program, our analyzer is able to determine that the operations from the different threads may refer to the same underlying memory access and our analyzer is be able to follow threads through multiple and different thread

creation calls. Our analyzer also tracks which operations happens before the new threads are created.

5.2.3 CVE-2021-32606

CVE-2021-32606 is a race condition in the isotp_setsockopt() function of the Linux kernel. There is a race which allows the socket structure to be changed after being bound, which can lead to a use-after-free (UAF). UAF are bugs where a pointer is accessed after the underlying memory has been deallocated and possibly re-used. This leads to undefined behaviour because the underlying data could be modified by another thread allocating new dynamic memory.

This code is related to the ISO-TP standard, used to send packets over a CAN-Bus (Controller Area Network Bus). The kernel defines a series of operations for a protocol, such as a *binding* operation or a *connect* operation. These operations are linked to specific kernel functions which will be called by the kernel with user supplied values when the operation is executed.

Listing 5.1: proto_ops definition

```
static const struct proto_ops isotp_ops = {
  .family = PF_CAN,
  .release = isotp_release,
  .bind = isotp_bind,
  .connect = sock_no_connect,
  .socketpair = sock_no_socketpair,
  .accept = sock_no_accept,
  .getname = isotp_getname,
  .poll = datagram_poll,
  .ioctl = isotp_sock_no_ioctlcmd,
  .gettstamp = sock_gettstamp,
  .listen = sock_no_listen,
  .shutdown = sock_no_shutdown,
  .setsockopt = isotp_setsockopt,
  .getsockopt = isotp_getsockopt,
  .sendmsg = isotp_sendmsg,
  .recvmsg = isotp_recvmsg,
  .mmap = sock_no_mmap,
  .sendpage = sock_no_sendpage,
};
```

These functions can be called simultaneously by different threads and can be executed in parallel by the Linux kernel. In this CVE, the data race is between the isotp_bind and the isotp_setsockopt functions.

Here is shortened explanation of how isotp_bind works as inlined comments starting with "EX"

Listing 5.2: isotp_bind

```
struct sockaddr_can *addr = (struct sockaddr_can *)uaddr;
  struct sock *sk = sock->sk;
  struct isotp_sock *so = isotp_sk(sk);
  struct net *net = sock_net(sk);
  int ifindex;
  struct net_device *dev;
  int err = 0;
  int notify_enetdown = 0;
  int do_rx_reg = 1;

  if (len < ISOTP_MIN_NAMELEN)
    return -EINVAL;

  /* do not register frame reception for functional addressing */
  if (so->opt.flags & CAN_ISOTP_SF_BROADCAST) //EX: This is a check to see if the socket needs to
       register a CAN receiver
    do_rx_reg = 0;

  /* do not validate rx address for functional addressing */
  if (do_rx_reg) {
    if (addr->can_addr.tp.rx_id == addr->can_addr.tp.tx_id)
      return -EADDRNOTAVAIL;

    if (addr->can_addr.tp.rx_id & (CAN_ERR_FLAG | CAN_RTR_FLAG))
      return -EADDRNOTAVAIL;
  }

  ....

  lock_sock(sk);

  ....

  if (do_rx_reg) //EX: If we we do not have the CAN_ISOTP_SF_BROADCAST flag set, we allocate and
       register a CAN receiver in can_rx_register
    can_rx_register(net, dev, addr->can_addr.tp.rx_id,
        SINGLE_MASK(addr->can_addr.tp.rx_id),
        isotp_rcv, sk, "isotp", sk);

  ....
  so->bound = 1; //EX: Here the socket is bound

out:
```

```
  release_sock(sk);

  ....

  return err;
}
}
```

Here is the explanation for the isotp_setsockopt function:

Listing 5.3: isotp_setsockopt

```
static int isotp_setsockopt(struct socket *sock, int level, int optname,
          sockptr_t optval, unsigned int optlen)
{
  struct sock *sk = sock->sk;
  struct isotp_sock *so = isotp_sk(sk);
  int ret = 0;

  if (level != SOL_CAN_ISOTP)
    return -EINVAL;

  if (so->bound) //EX: Here we check if the socket is not bound and return if it is.
    return -EISCONN;

  switch (optname) {
  ....
  case CAN_ISOTP_OPTS:
    if (optlen != sizeof(struct can_isotp_options))
      return -EINVAL;

    if (copy_from_sockptr(&so->opt, optval, optlen)) %EX: Here we copy into the socket so the opt
        values in optval
      return -EFAULT;

    /* no separate rx_ext_address is given => use ext_address */
    if (!(so->opt.flags & CAN_ISOTP_RX_EXT_ADDR))
      so->opt.rx_ext_address = so->opt.ext_address;
    break;

  ....

  default:
    ret = -ENOPROTOOPT;
  }

  return ret;
}
```

If isotp_bind and isotp_setsockopt are both run simultaneously, there is a data race between the socket flag check in isotp_bind and the values of these flag being changed by isotp_setsockopt. This can result in a socket where the flag CAN_ISOTP_SF_BROADCAST

is set while the socket structure has been initiated as if the flag was not set (with a CAN receiver allocated).

This is problematic when isotp_release() is called:

Listing 5.4: isotp_release

```
static int isotp_release(struct socket *sock)
{
  struct sock *sk = sock->sk;
  struct isotp_sock *so;
  struct net *net;

  if (!sk)
    return 0;

  so = isotp_sk(sk);
  net = sock_net(sk);

  ....

  lock_sock(sk);

  ....

  /* remove current filters & unregister */
  if (so->bound && (!(so->opt.flags & CAN_ISOTP_SF_BROADCAST))) { //EX: Free the registed CAN
      receiver only if the flag CAN_ISOTP_SF_BROADCAST is not set
    if (so->ifindex) {
      struct net_device *dev;

      dev = dev_get_by_index(net, so->ifindex);
      if (dev) {
        can_rx_unregister(net, dev, so->rxid,
              SINGLE_MASK(so->rxid),
              isotp_rcv, sk);
        dev_put(dev);
      }
    }
  }

  so->ifindex = 0;
  so->bound = 0;

  sock_orphan(sk);
  sock->sk = NULL;

  release_sock(sk);
  sock_put(sk);

  return 0;
}
```

In isotp_release, a check is made to verify if the CAN receiver should be unregistered.

This is only done on sockets that do not have the flag CAN_ISOTP_SF_BROADCAST set. However, because of the data race it is possible to construct a socket where the CAN_ISOTP_SF_BROADCAST flag has been set and where we also have a registered a CAN receiver. When the socket is unregistered, this leads to a dangling CAN receiver structure which when used will lead to a UAF.

The patch to this vulnerability locks the socket structure before changing the socket options in isotp_setsockopt and moves the acquisition of the lock in isotp_bind to before the check for the CAN_ISOTP_SF_BROADCAST flag. This ensures that the socket structure cannot be modified in the middle of the isotp_bind function.

To find this data race, our analyzer will be run on the isotp_bind and isotp_setsockopt functions. We are expecting to find a conflict between these two threads when accessing the socket structure flags: it is possible for isotp_bind to read the socket flag before or after isotp_setsockopt changes them.

5.2.4 CVE-2022-45886

CVE-2022-45886 is a race condition in the Linux kernel dvb (digital video broadcasting) device driver. The vulnerability and patch for this vulnerability was written and posted by Hyunwoo Kim in November 2022 to the Linux kernel maintainer for the media subsystem. [30].

Sadly, it seems like the patch introduced to fix these race conditions introduced a regression in the linux kernel and were reverted in June 2023 [31]. The locking changes are suspected to have introduced another race condition leading to a potential deadlock while loading a new dvb device.

Once again, specific kernel functions are called on a dvb_frontend object by mapping file operations to the specific functions to be called.

Listing 5.5: dvb-frontend

```
static const struct file_operations dvb_frontend_fops = {
```

```
        .owner          = THIS_MODULE,
        .unlocked_ioctl = dvb_frontend_ioctl,
#ifdef CONFIG_COMPAT
        .compat_ioctl   = dvb_frontend_compat_ioctl,
#endif
        .poll           = dvb_frontend_poll,
        .open           = dvb_frontend_open,
        .release        = dvb_frontend_release,
        .llseek         = noop_llseek,
};
```

dvb frontends are created and register by devices, such as a usb device. The author of the report used for example the Abilis AS102 DVB driver. The following code leads to the allocation of a dvb_frontend structure:

Listing 5.6: dvb-frontend-allocation

```
static int as102_usb_probe(struct usb_interface *intf,   //EX: the probe is called by the kernel to
     initialize the device
                           const struct usb_device_id *id)
{
....
        /* register dvb layer */
        ret = as102_dvb_register(as102_dev);
        if (ret != 0)
                goto failed_dvb;
....
int as102_dvb_register(struct as102_dev_t *as102_dev)
{
....
 ret =  dvb_register_frontend(&as102_dev->dvb_adap, as102_dev->dvb_fe);
        if (ret < 0) {
                dev_err(dev, "%s: as102_dvb_register_frontend() failed: %d",
                    __func__, ret);
                goto efereg;
        }
....
int dvb_register_frontend(struct dvb_adapter *dvb,
        struct dvb_frontend *fe)
{
  struct dvb_frontend_private *fepriv;
  const struct dvb_device dvbdev_template = {
    .users = ~0,
    .writers = 1,
    .readers = (~0) - 1,
    .fops = &dvb_frontend_fops,
#if defined(CONFIG_MEDIA_CONTROLLER_DVB)
    .name = fe->ops.info.name,
#endif
  };
  int ret;

  dev_dbg(dvb->device, "%s:\n", __func__);
```

```
  if (mutex_lock_interruptible(&frontend_mutex))
    return -ERESTARTSYS;

  fe->frontend_priv = kzalloc(sizeof(struct dvb_frontend_private), GFP_KERNEL);
  if (!fe->frontend_priv) {
    mutex_unlock(&frontend_mutex);
    return -ENOMEM;
  }
  fepriv = fe->frontend_priv;

  kref_init(&fe->refcount);
  mutex_init(&fe->remove_mutex);

  /*
   * After initialization, there need to be two references: one
   * for dvb_unregister_frontend(), and another one for
   * dvb_frontend_detach().
   */
  dvb_frontend_get(fe); %EX: kref == 2 now
....
}
```

The probe function is called by the kernel whenever the device is plugged in. The probe function leads to the creation of a dvb_frontend. When open() is called on the device, the following functions are executing, leading to an increase in the number of references to the dvb_frontend object.

Listing 5.7: dvb-frontend-open

```
static int dvb_frontend_open(struct inode *inode, struct file *file)
{
  struct dvb_device *dvbdev = file->private_data;
  struct dvb_frontend *fe = dvbdev->priv;
  struct dvb_frontend_private *fepriv = fe->frontend_priv;
  struct dvb_adapter *adapter = fe->dvb;
  int ret;

  .....

  dvb_frontend_get(fe);

  .....
}
static void dvb_frontend_get(struct dvb_frontend *fe)
{
  kref_get(&fe->refcount); %EX: kref == 3 now
}

.....
```

With the open call done on the device, the reference counter of the dvb_frontend object is now at 3.

If, after this open call is finished, the device is closed and disconnected at the same time, a race condition happens leading to a user after free.

When the device is disconnected, both dvb_unregister_frontend() and dvb_frontend_detach() are called. As explained in the comment written in the dvb_register_frontend() function, each of these functions will remove one of the kernel reference to the dvb_frontend object.

Listing 5.8: dvb-unregister-frontend

```
int dvb_unregister_frontend(struct dvb_frontend *fe)
{
        struct dvb_frontend_private *fepriv = fe->frontend_priv;

        dev_dbg(fe->dvb->device, "%s:\n", __func__);

        mutex_lock(&frontend_mutex);
        dvb_frontend_stop(fe);
        dvb_remove_device(fepriv->dvbdev);

        /* fe is invalid now */
        mutex_unlock(&frontend_mutex);
        dvb_frontend_put(fe); %EX: kref == 2 now
        return 0;
}
EXPORT_SYMBOL(dvb_unregister_frontend);
.....
```

Listing 5.9: dvb-frontend-detach

```
void dvb_frontend_detach(struct dvb_frontend *fe)
{
        dvb_frontend_invoke_release(fe, fe->ops.release_sec);
        dvb_frontend_invoke_release(fe, fe->ops.tuner_ops.release);
        dvb_frontend_invoke_release(fe, fe->ops.analog_ops.release);
        dvb_frontend_put(fe); %EX: kref == 1 now
}
EXPORT_SYMBOL(dvb_frontend_detach);

.....
```

If after dvb_frontend_detach() is called, the close() is called, a use-after-free happens. When close() is called, dvb_frontend_release() is called followed by dvb_free_device().

Listing 5.10: dvb-frontend-detach commit:ae11c0efaec3

```
static int dvb_frontend_release(struct inode *inode, struct file *file)
{
        struct dvb_device *dvbdev = file->private_data;
        struct dvb_frontend *fe = dvbdev->priv;
        struct dvb_frontend_private *fepriv = fe->frontend_priv;
        int ret;

        dev_dbg(fe->dvb->device, "%s:\n", __func__);

        if ((file->f_flags & O_ACCMODE) != O_RDONLY) {
                fepriv->release_jiffies = jiffies;
                mb();
        }

        ret = dvb_generic_release(inode, file);

        if (dvbdev->users == -1) {
                wake_up(&fepriv->wait_queue);
#ifdef CONFIG_MEDIA_CONTROLLER_DVB
                mutex_lock(&fe->dvb->mdev_lock);
                if (fe->dvb->mdev) {
                        mutex_lock(&fe->dvb->mdev->graph_mutex);
                        if (fe->dvb->mdev->disable_source)
                                fe->dvb->mdev->disable_source(dvbdev->entity);
                        mutex_unlock(&fe->dvb->mdev->graph_mutex);
                }
                mutex_unlock(&fe->dvb->mdev_lock);
#endif
                if (fe->exit != DVB_FE_NO_EXIT)
                        wake_up(&dvbdev->wait_queue);
                if (fe->ops.ts_bus_ctrl)
                        fe->ops.ts_bus_ctrl(fe, 0);
        }

        dvb_frontend_put(fe); %EX: kref == 0 now, object is freed

        return ret;
}
.....
```

Listing 5.11: dvb-free-device commit:ae11c0efaec3

```
static void dvb_free_device(struct kref *ref)
{
        struct dvb_device *dvbdev = container_of(ref, struct dvb_device, ref);

        kfree (dvbdev->fops);
        kfree (dvbdev);
}
.....
```

At this point, the object used has already been freed and a UAF occurs. The cause

of this bug is the lack of synchronization between the threads executing dvb_unregister_frontend() and the dvb_frontend_release() functions. In this driver, waitqueues are used to make threads wait until a wake-up signal is sent. The patch for this CVE adds a call to wait_event() in dvb_frontend_stop() and a mutex, which will prevent the thread from continuing executing and from decrementing the kernel reference in dvb_unregister_frontend()

To evaluate how our analyzer finds this bug, we ran our analyzer specifying the entry points as being the dvb_frontend_release() and the dvb_unregister_frontend() functions. While our analyzer does not count the number of references to the dvb_frontend object, we will try to detect the improper freeing of the dvb_frontend object by detecting the case when both threads modify the number of references without synchronization.

5.3 Evaluation Setup

Our setup for the evaluation consists of a single x86 system, as represented in the following table:

Evaluation Setup	
Component	Model
CPU	13th Gen Intel® Core™ ientoverview 7-13700K × 24
Memory	64 GB DDR4 Synchronous 2400 MHz
OS	Ubuntu 22.04.2 LTS
Kernel	5.19.0-43-generic
Java	openjdk 19.0.2
Ghidra	11.0.2

As our analysis is written in Java and is running on Ghidra which is also programmed in Java, there is less concern about the environment our analysis is running in. Our analyzer should work on any system supported by Ghidra, which includes all major desktop platforms (Windows 7 or 10, Linux or macOS 10.8.3+). Ghidra's minimum

requirements are 4GB of RAM and Java 11 64-bit JDK. A more performant system can expect a faster analysis and the ability to support bigger binaries.

5.4 Results

The result of our evaluation against our testing set can be seen in table 5.2.

Table 5.1: Data race detection results of our analyzer against the testing set.

	True Positive	**False Positive**	**False Negative**	**Memory Accesses**
Program A	2	0	0	300
Program B	0	0	0	322
Program C	6	0	0	429
Program D	0	0	0	449
Linux Kernel CVE-2021-32606	31	12	0	37228
Linux Kernel CVE-2021-45886	42	25	0	35228

Table 5.2: Performance results of our analyzer against the testing set.

	Analysis Time (seconds) Analysis Depth Limit 25 (function calls)	**Analysis Time Analysis Depth limit 75 (function calls)**
Program A	0.1	0.1
Program B	0.1	0.1
Program C	0.2	0.2
Program D	0.2	0.2
Linux Kernel CVE-2021-32606	17.27	300.46
Linux Kernel CVE-2021-45886	24.17	338.14

We conducted our evaluation by analyzing each program in our testing set once as there is no randomization done in our analyzer. However, the initial importation of the program inside Ghidra also took time for initial analysis. For this evaluation, we have kept the analysis to depth limit of 25, meaning that the analyzer only explores up to 25 functions deep from the entry points. As expected, increasing this value to 75 dramatically increased the analysis time. According to our profiler, more than 70% of the analysis time is spent traversing new functions and decompiling them, and we can expect more in depth or complex analysis to exponentially increase the analysis time because of growing number of code paths to analyze. However, creating new memory access objects and running our data race detection algorithm on them accounted for the remaining 30% of the computation. Even against the bigger binaries of our testing set, less than 1GB of memory is being used. Our analyzer demonstrates that it is possible to analyze binaries with reasonable performance with the use of caching and by minimizing the amount of code exploration when we can.

While conducting our profiling, the majority of the time spent in our analysis is from the decompilation of the different functions by Ghidra. Our use of caching helps reduce the amount of code exploration required to determine the lockset and memory accesses in the targeted program. If possible, a faster decompilation process would definitely be one of the best ways to improve the analyzer's performance. The other solution would be to have a more precise analysis and to explore less code path.

Whenever a new memory access is detected, our algorithm checks if it might be a data race with an already detected memory access. If so, we either create a new alert or match the memory access to an existing one. While evaluating our analyzer against the test programs we created, our analyzer was able to find the memory accesses linked to the data race and create alerts. One of the challenges we face is trying to group together these memory accesses into one single alert. As seen in table 5.2, we often create too many alerts. Whenever our analyzer fails to detect that multiple memory

accesses are related to the same underlying data race (accesses coming from the same thread to the same underlying object) or whenever our analyzer makes a mistake while determining pointer aliasing, we can expect false positives in our analysis. Since we were looking for specific data races and because when in doubt our analyzer prefers to generate a false positive alert rather than miss a data race, we have had zero false negative in our evaluation.

5.5 Limitations

Our binary analyzer is currently able to disassemble and start an analysis against any of the binary architecture supported by Ghidra. There are however currently limitations on the effectiveness of our lock detection component against non-Linux binaries. This is caused by each platform being able to handle synchronization primitives differently. This would need an expansion of the lock abstraction layer to handle these new synchronization primitives.

The other major limitation of our analyzer is its dependence on the accurate disassembly and on the accuracy of the memory alias layer. The false positives detected in our evaluation are caused by mistakes in correctly resolving pointer aliases or by missing synchronization happening between two memory access to the same variable. For example, our analysis might mistake two accesses to the same memory location to be a data race while complex control flow conditions make it impossible for both accesses to occur.

Finally, a major challenge of race detection tools is the management of benign races and of the different generated alerts. While our analyzer implements basic techniques to avoid duplicate alert and to filter out benign races, we focused our efforts on the ability to efficiently analyze binaries. Improvement to the generated alerts could be made to help the user pinpoint quicker the interesting data races found.

Chapter 6

Conclusion

6.1 Conclusion

Concurrent programming has been the main mechanism used to gain more performance out of modern computer processors. As described in chapter 2, the memory model of commonly used systems such as of the C/C++ programming language do not define the expected behaviour when a data race happens. When this is coupled with the difficulty of reproducing data races due to their randomness, this makes security issues especially hard to identify and fix. This makes developing static analysis techniques that are independent from the timing of the underlying system an important area of research to find these hard to identify bugs in software. Moreover, with more software running on closed devices that are difficult to instrument (such as mobile devices) and without access to the software's source code, there is a need for more techniques enabling analysis of this software.

We present in this thesis an approach to detect data races using static analysis on binaries. Previous work in the area of data race detection focused on live system instrumentation and on source code analysis. To enable a more target independent analysis and to allow analysis of closed-source programs, we contribute a binary analysis

for data races built on top of the Ghidra reverse engineering framework. Our analyzer is implemented as a Ghidra script on top of Ghidra's p-code IR and uses the annotations specified by the user. This makes our tool very easy to use but also expandable: Ghidra is open-source and new architecture can be added to lift binary code to p-code. The analysis also takes into account the user's annotations in the binary if there are any, making it easy for the user to guide the analysis and correct possible issues and to make the analysis more accurate.

We performed an evaluation of our approach on a set of testing binaries and on a set of previously found data races in the Linux kernel. Our evaluation demonstrated that our analyzer is able to perform successfully the task of exploring the binary control flow, building a map of the memory accesses in the program, tracking the lockset through the program and determining when two memory access intersect. During our evaluation, our analyzer was able to output alerts to identify the previously reported data races in two different Linux kernel subsystems. Finally, we demonstrate that the chosen approach of using a lockset based static analysis solution is viable and can be used to analyze binaries for security-related data races in an effective and flexible way.

6.2 Future Work

Further research in effective techniques and heuristics to conduct pointer aliasing in modern binaries would help in increasing the accuracy of binary analysis. While pointer aliasing analysis is an undecidable problem, as explained in our background section, specific binary-focused techniques to find an approximate and precise solution to this problem will allow the implementation of a more accurate analysis that can identify harder to find data races in binary programs. Such work would easily fit in our current modular implementation as an extension of our data race detector module that tries to determine when two memory access intersect.

The creation of an evaluation dataset for data races would be a very useful addition. While there are some available testing sets of different types of security bugs, there is no well detailed and annotated set of programs with different types of data race bugs to use for evaluating research done against data races. We built ourselves a basic set of programs to test our analyzer, but a standard testing suite would help evaluate and compare our tool with the other state-of-the-art solutions.

With the increasing support for multiple platforms (mobile and desktop) and of continuous testing, future work could be done to enable the use of this static analysis approach for automatic analysis of the produced binaries. While there would be challenges to solve to ensure that the analysis is precise and performant enough to work in this application, our current approach has the benefit of being flexible and of supporting a large set of binary architecture. While bugs introduced by compilers are rare, the code implementing locking tends to be lower-level code that is usually different for each system architecture. This work would help advance the state of automated security testing of binaries.

Finally, future research could be done on evaluating if such static analysis techniques to detect data races could be coupled with the usually used dynamic analysis techniques. For example, this static analysis technique could be used on running programs in a system to identify memory accesses suspected of causing data races followed by a dynamic instrumentation of such memory accesses to confirm the existence of a data race. The performance and reliability problems of dynamic analysis techniques could be alleviated by having a more precise area of analysis and by having more information about the suspected cause of the data race thanks to our static analysis.

Bibliography

[1] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. "The Android Platform Security Model". In: *ACM Transactions on Privacy and Security* 24.3 (Aug. 2021), pp. 1–35. DOI: `10.1145/3448609`. URL: `https://doi.org/10.1145/3448609`.

[2] Lillian Ablon and Andy Bogart. *Zero Days, Thousands of Nights: The Life and Times of Zero-Day Vulnerabilities and Their Exploits*. RAND Corporation, 2017. DOI: `10.7249/rr1751`. URL: `https://doi.org/10.7249/rr1751`.

[3] MITRE. *About CVE Entries*. 2020. URL: `https://cve.mitre.org/cve/identifiers/index.html`.

[4] Matt Miller. *Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape*. 2019. URL: `https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf`.

[5] Evgenii Stepanov. *Detecting Memory Corruption Bugs With HWASan*. 2020. URL: `https://android-developers.googleblog.com/2020/02/detecting-memory-corruption-bugs-with-hwasan.html`.

[6] The Chromium Projects. *Memory Safety*. 2021. URL: `https://www.chromium.org/Home/chromium-security/memory-safety`.

[7] Diane Hosfelt. *Implications of Rewriting a Browser Component in Rust*. 2019. URL: https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/.

[8] Dawson Engler and Ken Ashcraft. "RacerX". In: *ACM SIGOPS Operating Systems Review* 37.5 (Dec. 2003), pp. 237–252. DOI: 10.1145/1165389.945468. URL: https://doi.org/10.1145/1165389.945468.

[9] PassMark® Software. *Hardware Survey - CPU Cores*. 2022. URL: https://www.pcbenchmarks.net/number-of-cpu-cores.html.

[10] Apple. *iPhone 14*. 2022. URL: https://www.apple.com/ca/iphone-14/specs/.

[11] Robert H. B. Netzer and Barton P. Miller. "What are race conditions?" In: *ACM Letters on Programming Languages and Systems* 1.1 (Mar. 1992), pp. 74–88. DOI: 10.1145/130616.130623. URL: https://doi.org/10.1145/130616.130623.

[12] "Encyclopedia of parallel computing". en. In: ed. by David Padua. 2011th ed. Encyclopedia of Parallel Computing. New York, NY: Springer, Jan. 2012, p. 1692.

[13] Vijay A. Saraswat, Radha Jagadeesan, Maged Michael, and Christoph Von Praun. "A theory of memory models". en. In: *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. San Jose California USA: ACM, Mar. 2007, pp. 161–172. ISBN: 978-1-59593-602-8. DOI: 10.1145/1229428.1229469. URL: https://dl.acm.org/doi/10.1145/1229428.1229469 (visited on 10/10/2023).

[14] Leslie Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". In: *IEEE Transactions on Computers C-28* 9 (Sept. 1979), pp. 690–691. URL: https://www.microsoft.com/en-us/research/publication/make-multiprocessor-computer-correctly-executes-multiprocess-programs/.

[15] Hans-J Boehm and Sarita V Adve. "Foundations of the C++ Concurrency Memory Model". en. In: (), p. 11.

[16] A. J. Bernstein. "Analysis of Programs for Parallel Processing". In: *IEEE Transactions on Electronic Computers* EC-15.5 (Oct. 1966), pp. 757–763. DOI: 10.1109/pgec.1966.264565. URL: https://doi.org/10.1109/pgec.1966.264565.

[17] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. "Learning from mistakes". In: *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems - ASPLOS XIII*. ACM Press, 2008. DOI: 10.1145/1346281.1346323. URL: https://doi.org/10.1145/1346281.1346323.

[18] Shin Hong and Moonzoo Kim. "A survey of race bug detection techniques for multithreaded programmes". In: *Software Testing, Verification and Reliability* 25.3 (Dec. 2014), pp. 191–217. DOI: 10.1002/stvr.1564. URL: https://doi.org/10.1002/stvr.1564.

[19] William Landi. "Undecidability of static analysis". In: *ACM Letters on Programming Languages and Systems* 1.4 (Dec. 1992), pp. 323–337. DOI: 10.1145/161494.161501. URL: https://doi.org/10.1145/161494.161501.

[20] Bernhard Reus. *Limits of computation*. en. 1st ed. Undergraduate Topics in Computer Science. Basel, Switzerland: Springer International Publishing, Apr. 2016.

[21] Gogul Balakrishnan and Thomas Reps. "WYSINWYX". In: *ACM Transactions on Programming Languages and Systems* 32.6 (Aug. 2010), pp. 1–84. DOI: 10.1145/1749608.1749612. URL: https://doi.org/10.1145/1749608.1749612.

[22] Thomas Dullien and Sebastian Porst. "REIL: A platform-independent intermediate representation of disassembled code for static code analysis". In: 2009.

[23] Trail of bits. *McSema*. 2020. URL: https://github.com/lifting-bits/mcsema.

[24] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. *SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly But Were Afraid to Ask*. 2020. DOI: `10.48550/ARXIV.2007.14266`. URL: `https://arxiv.org/abs/2007.14266`.

[25] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. "Effective Data-Race Detection for the Kernel". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. USENIX Association, 2010, 151–162.

[26] Stefan Savage. "Eraser: A Dynamic Data Race Detector for Multithreaded Programs". en. In: *ACM Transactions on Computer Systems* 15.4 (), p. 21.

[27] Yuan Yu, Tom Rodeheffer, and Wei Chen. "RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking". en. In: (), p. 14.

[28] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. "LOCKSMITH: Practical static race detection for C". en. In: *ACM Transactions on Programming Languages and Systems* 33.1 (Jan. 2011), pp. 1–55. ISSN: 0164-0925, 1558-4593. DOI: `10.1145/1889997.1890000`. URL: `https://dl.acm.org/doi/10.1145/1889997.1890000` (visited on 10/06/2020).

[29] Konstantin Serebryany and Timur Iskhodzhanov. "ThreadSanitizer: data race detection in practice". en. In: *Proceedings of the Workshop on Binary Instrumentation and Applications - WBIA '09*. New York, New York: ACM Press, 2009, p. 62. ISBN: 978-1-60558-793-6. DOI: `10.1145/1791194.1791203`. URL: `http://portal.acm.org/citation.cfm?doid=1791194.1791203` (visited on 06/07/2020).

[30] Kim Hyunwoo. *[PATCH 0/4] Fix multiple race condition vulnerabilities in dvb-core and device driver*. 2022. URL: `https://lore.kernel.org/linux-media/20221115131822.6640-1-imv4bel@gmail.com/`.

[31] Mauro Carvalho Chehab. *Re: Sometimes DVB broken with commit 6769a0b7ee0c3b*. 2023. URL: https://lore.kernel.org/all/20230614232024.171130f8@sal.lan/.