University of Deusto

Analyses on the Evolution of Memory Corruption into the Concurrent Era and Detection Methods

Oscar Llorente Vázquez

Bilbao, June 2023

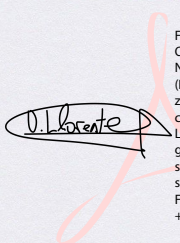
University of Deusto

Analyses on the Evolution of Memory Corruption into the Concurrent Era and Detection Methods

A dissertation submitted by Oscar Llorente Vázquez
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in
Engineering for the Information Society and Sustainable Development

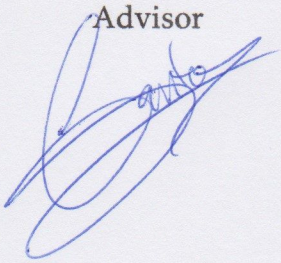
Advised by Dr. Igor Santos Grueiro & Dr. Pablo García Bringas

Author



Firmado digitalmente por OSCAR LLORENTE VAZQUEZ
Nombre de reconocimiento (DN): c=ES, ou=Herritar ziurtagiria - Certificado de ciudadano, cn=OSCAR LLORENTE VAZQUEZ, givenName=OSCAR, sn=LLORENTE VAZQUEZ, serialNumber=16081447P
Fecha: 2023.06.14 14:21:15 +02'00'

Advisor



Advisor

Firmado digitalmente por GARCIA BRINGAS, PABLO (FIRMA)
Nombre de reconocimiento (DN): c=ES, serialNumber=22754794C, sn=GARCIA, givenName=PABLO, cn=GARCIA BRINGAS, PABLO (FIRMA)
Fecha: 2023.06.14 14:55:24 +02'00'

Bilbao, June 2023

To my family and friends for all the support.

Resumen

Las vulnerabilidades de software son la causa de multitud de problemas de seguridad en los sistemas informáticos. Debido a su eficiencia y a su estricto control sobre los recursos del sistema a bajo nivel, los lenguajes de programación C y C++ se utilizan para innumerables propósitos. Sin embargo, una gestión de memoria insuficiente o inadecuada puede resultar en vulnerabilidades de corrupción de memoria. Estas vulnerabilidades son una primitiva fuerte y flexible de explotación, y se utilizan activamente para ataques elaborados que que eluden los métodos de defensa existentes.

Mientras tanto, los programas concurrentes están cada vez más presentes en los sistemas modernos. Estos programas hacen mejor uso de los recursos del procesador, pero introducen un nuevo conjunto de problemas en términos de fiabilidad y seguridad. Los errores de concurrencia provocan comportamientos inesperados y son un área de investigación activa. En términos de seguridad, las vulnerabilidades de concurrencia muestran un comportamiento dañino exclusivamente en ejecuciones concurrentes. Una característica particular de la concurrencia es que no sólo introduce nuevos problemas, sino que también permite que se activen vulnerabilidades existentes de formas específicas a la concurrencia. Concretamente, las vulnerabilidades más peligrosas causan corrupción de memoria, y se conocen como vulnerabilidades de corrupción de memoria de concurrencia.

En el desarrollo de métodos de detección de vulnerabilidades de corrupción de memoria de concurrencia, la instrumentación dinámica de binarios (DBI) es una tecnología fundamental. La DBI es el conjunto de técnicas que permiten instrumentar programas en tiempo de ejecución, haciendo posible monitorizar y modificar la ejecución de binarios compilados o sistemas completos. De este modo, los sistemas de DBI facilitan a los usuarios las herramientas para definir rutinas que se ejecutan cada

vez que se producen determinados eventos. Además, la DBI se utiliza para innumerables aplicaciones y análisis de seguridad, tanto en la industria como en el mundo académico. A lo largo de los años, se han propuesto soluciones basadas en diferentes tecnologías e implementando diversas técnicas. Cada solución intenta superar ciertas limitaciones, pero pero a veces presentan otras deficiencias.

Esta tesis doctoral profundiza en tres elementos interrelacionados: (i) errores de memoria, (ii) vulnerabilidades de concurrencia (centrándose en la corrupción de memoria de concurrencia), y (iii) técnicas de instrumentación dinámica de binarios (DBI). Primero, analizamos las características de los distintos tipos de errores de memoria y cómo son explotados por los atacantes, así como los principales métodos de defensa propuestos. Además, sistematizamos las técnicas más relevantes en materia de detección de corrupción de memoria. Asimismo, desarrollamos una herramienta para analizar la evolución del panorama de la corrupción de memoria en programas reales.

Analizamos de forma sistemática las vulnerabilidades de concurrencia en programas escritos en C y C++, su explotación y su detección, centrándonos en la corrupción de memoria de concurrencia. Organizamos el trabajo previo sobre las características y detección de errores de concurrencia, destacando las diferencias en relación a las vulnerabilidades. Después, ampliamos nuestra herramienta anterior para analizar la presencia y las propiedades de las vulnerabilidades de concurrencia en programas reales, identificando una tendencia en aumento. Además, analizamos y comparamos los enfoques existentes para la detección de vulnerabilidades de corrupción de memoria de concurrencia, muchos de ellos implementados sobre DBI.

En cuanto a la instrumentación dinámica de binarios (DBI), unificamos los enfoques a nivel de proceso y de sistema. Identificamos y definimos sus bloques básicos y analizamos las técnicas de instrumentación subyacentes, comparando su capacidad para instrumentar primitivas y eventos. Después, evaluamos su rendimiento al implementar cada primitiva, y destacamos las observaciones relevantes. Nuestros resultados muestran que ninguna técnica es mejor que el resto en todas las circunstancias.

Abstract

Software vulnerabilities are the root cause for a multitude of security problems in computer systems. Owing to their efficiency and tight control over low-level system resources, the C and C++ programming languages are extensively used for a myriad of purposes, from implementing operating system kernels to user-space applications. However, insufficient or improper memory management frequently leads to invalid memory accesses, eventually resulting in memory corruption vulnerabilities. These vulnerabilities are a strong and flexible primitive for exploitation, and are actively used as a foothold for elaborated attacks that bypass existing defense methods.

Meanwhile, concurrent programs are widespread in modern systems. They make better use of processor resources but inevitably introduce new problems in terms of reliability and security. Concurrency bugs usually lead to program crashes and unexpected behavior. From a security perspective, concurrency vulnerabilities exhibit harmful behavior exclusively in concurrent executions. They take place in a diverse range of environments, such as in operating system kernels, file system operations, or general-purpose multithreaded programs. A particular characteristic of concurrency is that it not only introduces new problems, but also enables traditional vulnerabilities to be triggered in concurrent-specific ways. Those that lead to dangerous security vulnerabilities usually cause memory corruption, and are known as concurrency memory corruption vulnerabilities.

In the development of concurrency memory corruption vulnerability detection frameworks, Dynamic Binary Instrumentation (DBI) is a core technology. DBI is the set of techniques that enable instrumentation of programs at run-time, making it possible to monitor and modify the execution of compiled binaries or entire systems. DBI frameworks facilitate users the tools to

define routines that will be executed whenever certain events occur in the program. DBI is used for countless security applications and analyses, and is extensively used across many fields in both industry and academia. The applications of DBI are countless besides concurrecy vulnerability detection: program analysis, security policy enforcement, software emulation or virtualization, and so on. Over the years, several DBI approaches have been proposed based on different technologies and implementing diverse techniques. Every solution tries to overcome certain limitations, but sometimes bring other shortcomings.

This dissertation delves into three interrelated elements: (i) memory errors, (ii) concurrency vulnerabilities (focusing on concurrency memory corruption), and (iii) Dynamic Binary Instrumentation (DBI) techniques. First, we analyze the characteristics of memory safety violations (i.e., memory errors), and how they are exploited by attackers, as well as the main mitigation methods proposed in the research community. We further systematize the most relevant techniques with regards to memory corruption detection. Moreover, we develop a tool to analyze the memory corruption landscape evolution in real-world programs.

We systematically analyze concurrency vulnerabilities in C and C++ programs, their exploitation and their detection, focusing on concurrency memory corruption vulnerabilities. We organize previous work on concurrency bug characteristics and detection, and highlight the differences in relation to vulnerabilities. Then, we extend our previous tool to analyze the presence and the properties of concurrency vulnerabilities in real-world programs by searching the CVE databases and point out a growing trend. Further, we analyze and compare the few existing approaches towards concurrency memory corruption detection, many of them implemented on top of DBI.

With regard to DBI, we bring together process-level and whole-system approaches. We depict their building blocks and analyze the underlying instrumentation techniques, comparing their ability to instrument different primitives and run-time events. Then, we evaluate their performance when implementing each primitive, and highlight relevant observations. Our results show that no single technique is better than the rest in all circumstances.

Acknowledgements

First and foremost, I would like to thank all the people who have been with me on this journey and who in some way have contributed to help me achieve this goal. Thank you very much.

I would like to thank my advisors Dr. Igor Santos Grueiro and Dr. Pablo García Bringas. Thanks to Igor I discovered and had my first experience in the world of system security research, not even knowing that it would be my dream job. During these years, he has taught me many different things and has inspired me with his passion for science. We have gone through many difficulties and in the end I get more of a friend than an advisor. Pablo was the first person to teach me about computer security when I was his undergraduate student. He always motivated us with all his enthusiasm when talking about the university's security lab. He has helped me as much as possible in this journey and I am very grateful for it.

I would also like to thank all my closest colleagues: Iskander, Xabier Cantero, Irene, Ainhoa, Alberto, and Xabier Ugarte. Working with you has been a real pleasure and I will always carry very good memories with me. I also thank everybody in the University of Deusto and EURECOM who has spent time or collaborated with me in any way.

I also owe a special thanks to all my close friends. Even though most of them did not understand a word when I talked about my dissertation, they have always been curious, listened to me and cared about me. They always have been there and always will, hopefully. I am deeply grateful for having such an amazing second family. Many thanks also to Ari, one of the most wonderful people I have ever met.

Last but not least, I would like to thank my family, especially my mother Rosa, my father Ezequiel, and my brother Ivan. I cannot

express how lucky, grateful and happy I am for having them with me. I would also like to express my deepest gratitude to my uncle Luis, the first person who ever told me about science and passed on his curiosity to me when I was just a kid. Thank you very much.

Contents

List of Figures

List of Tables

Chapter

1

Introduction

Computer systems are widespread in current society. Companies, institutions, and people in general constantly use and rely on desktop computers, laptops, smartphones, servers, and many other forms of devices. The programs running in these systems are prone to software errors that may hinder their proper functioning, or even worse, lead to security vulnerabilities that can compromise these systems and the information they store.

Software vulnerabilities have become a serious risk at many levels and different areas in modern society. Attackers unceasingly discover vulnerabilities in widely used software and are able to launch massive and directed attacks.

Among the known software vulnerabilities, memory corruption is one of the oldest and still one of the most harmful issues in programs to date. With the development of increasingly complex systems and programs, memory corruption has become ubiquitous and has exhibited new behaviors in programs that take advantage of concurrent systems.

The rest of this chapter is organized as follows: Section 1.1 gives an introduction to the area of memory corruption vulnerabilities. Section 1.2 describes how concurrency and memory errors are linked together. Then, Section 1.3 introduces the main concepts of dynamic binary instrumentation and current challenges. Section 1.4 defines the hypothesis and objectives of this dissertation. Section 1.5 describes the methodology for this research. Finally, Section 1.6 details the outline of the manuscript.

1.1 Memory Errors in Programs

Memory and type unsafe languages such as C and C++ do not ensure that memory accesses in programs are valid, which can lead to program bugs that result in memory corruption and eventual exploitation, compromising system security. Memory errors have been present in software for more than three decades now, leading to numerous security issues. Low-level languages like C and C++ are prone to this class of errors and in widespread use, meaning that a large number of current systems are susceptible to attacks that target memory corruption vulnerabilities. Over the years, memory errors have been detected in countless programs of any type and nature [WXL+20, CRD20], and have since established themselves as the most dangerous and exploited vulnerabilities [Com]. Therefore, hardening programs and protecting against this type of attacks is of the utmost importance to build secure systems.

In spite of massive advances in memory safety violations detection and mitigation, memory errors persist in current versions of software, as they are further being exploited to carry out increasingly sophisticated attacks [IAJP18, GKK+18]. Unfortunately, a significant part of the proposed approaches are not adopted in real-world scenarios, either because of performance concerns, compatibility issues, or insufficient mitigation capabilities [SPWS13]. In fact, a previous work [VdVCB+12] analyzed the evolution of the memory corruption ecosystem, and concluded that it is unlikely that memory errors will become less important in the near future, a fact that remains true at present.

More worryingly, memory corruption does not take place during sequential execution. Depending on the interactions between threads and the scheduling, new memory errors can arise in concurrent programs, leading to concurrency memory corruption vulnerabilities that traditional detection tools fail to unmask [LZL+18, CZM+19].

1.2 From Sequential to Concurrency Vulnerabilities

Concurrency is a crucial capability in modern systems. By means of different technologies, it makes it possible to maximize the usage of computing resources and increase overall performance. However, concurrency brings in new problems and challenges as well, especially in the context

of security vulnerabilities. In contrast to sequential programs, concurrent program states are not only affected by program inputs but also by thread scheduling and the different thread interleavings, making it challenging to find, reproduce, and fix errors and vulnerabilities due to non-determinism [LPSZ08].

Concurrency bugs usually arise as a result of concurrent accesses to a shared resource without proper synchronization. The most representative scenarios include shared memory accesses among threads within the same process, and system resource accesses among different processes [FF09, LLM+19]. As a result of improper synchronization, different concurrent accesses are executed in non-deterministic interleavings, which compromises reliability and can lead to incorrect outputs, program crashes, and so on. Different from concurrency bugs (e.g., data races), concurrency vulnerabilities exhibit harmful behavior from a security perspective. Therefore, attackers are able to exploit these vulnerabilities to escalate privileges, execute malicious code, leak sensitive data, and so on. Concurrency vulnerabilities are relatively similar in nature to concurrency errors and have been generally oriented towards the file system (e.g., to read from or write to sensitive files) [CGJ09].

In practice, a concurrency vulnerability can become apparent as a result of a concurrency error, but it is not a mandatory requirement. In fact, making thread accesses to shared memory race-free does not necessarily prevent a concurrency vulnerability from taking place, even if they are correctly protected by the same lock [cvea]. Therefore, a potential concurrency vulnerability may appear if two or more out of a set of events (i.e., memory operations) in a given execution can have their order changed, leading to harmful program behavior.

In a different direction, memory errors are still pervasive in current software (i.e., written in low-level languages such as C and C++) and have been the focus of research for many years. By now, researchers have proposed a myriad of approaches to exploit memory corruption, protect systems from these attacks, and to enforce memory safety to prevent errors [SPWS13]. However, memory errors do not take place exclusively in their traditional form (i.e., sequentially). The role of concurrency makes it possible for memory corruption to be triggered in additional ways in concurrent executions. This gives rise to concurrency memory corruption vulnerabilities that standard detection approaches fail to unmask and to which little attention has been paid. In fact, while examining kernel commits to search for concurrency use-after-free bugs, researchers found that most of reported

concurrency use-after-free errors were identified by manual inspection or runtime testing [BLCH19].

At first, it was assumed that concurrency memory corruption vulnerabilities were only a consequence of concurrency errors such as data races [ZSL10, ZGQ+18], and that they were visible later in the execution flow. However, a concurrency vulnerability is more related to the orders of events (at different granularities) that can be inverted in alternative executions and exhibit harmful behavior, no matter whether the corresponding accesses can form data races [CZM+19]. A major difference between these two concepts is that conflicting accesses are usually expected to take place simultaneously in a data race, but not in concurrency vulnerabilities, as long as the harmful execution order can materialize.

The most naive method to try to find concurrency vulnerabilities would be to explore all possible thread interleavings of a program to identify harmful execution orders of operations. However, the number of possible thread interleavings grows exponentially, leading to interleaving space explosion and making this method unfeasible. Further, using data race detectors to find concurrency vulnerabilities leads to potential vulnerabilities being missed, and requires additional effort, since usually only a small portion of all reported data races are indeed harmful and even some are purposefully introduced by developers. Recently, some approaches have been proposed that focus on detecting concurrency memory corruption vulnerabilities, many of them building upon Dynamic Binary Instrumentation (DBI) systems. Some rely on fuzzing due to its demonstrated effectiveness [LZL+18, CGX+20], whereas others borrow and adapt predictive detection techniques from concurrency bug detection approaches [Hua18, CZM+19].

Although concurrency error characteristics and detection approaches (many of them using DBI as their core) have been extensively studied previously [LPSZ08, PVH10], concurrency vulnerabilities — especially those related to memory corruption — have not received nearly as much attention. More worryingly, an increasing number of concurrency vulnerabilities are being reported each year. Therefore, a comprehensive analysis of concurrency memory corruption vulnerabilities and known detection methods remains an important open task. A systematic study in this field will lead to a better understanding and to the development of new detection approaches.

1.3 Dynamic Binary Instrumentation

Dynamic Binary Instrumentation (DBI) frameworks are an extremely versatile and powerful mechanism with a wide range of applications besides concurrency error and vulnerability detection. For instance, they have been extensively used for performance profiling and analysis [KC12, ACF12] or software fault injection [WTLP14]. Recent work [DCN+19] has highlighted the pervasiveness of DBI in security research, which includes applications such as shepherding to enforce security policies such as Control-Flow Integrity (CFI) [PHY15, VDVGC+16], diverse vulnerability discovery and analysis [JC13, XLCZ17], fuzzing of embedded devices [PWL20, CGS+20], and malware analysis [PCM+17, UPBSB16].

Traditionally, the only frameworks that were considered DBI were those that explicitly insert user-defined code into the context of the target at runtime, executing that code as part of the normal instruction stream. It has also sometimes been ascribed entirely to user-space solutions (i.e., Intel Pin [LCM+05]). However, this interpretation is overly restrictive and excludes several approaches. Therefore, we broaden the perspective of DBI analysis and design so as to include systems that qualify as DBI frameworks by virtue of their similar purpose and characteristics.

Due to the high number of DBI implementations, and the heterogeneity in the designs and techniques, the current state of the art is diffuse, hard to understand and properly evaluate. For example, a new researcher in this field, or any professional who has to develop or use DBI for a particular problem would need to carefully read several research papers, websites and source code repositories, and test multiple tools in order to understand the fundamentals and capabilities of existing solutions. Moreover, it is still difficult to know which techniques and solutions are adequate in which circumstances, and to understand the advantages and disadvantages of each of them.

On the one hand, there has been extensive discussion on DBI transparency and evasive behavior [MANP17, DCN+19, FRF22, DIPQ22], especially in the context of adversarial environments such as malware analysis, but to the best of our knowledge there is no existing work that clearly categorizes DBI components, and analyzes and compares existing instrumentation techniques, their capabilities and relative performance in a comprehensive manner. In fact, they usually focus on a single technique or technology, or exclusively consider process-level or whole-system approaches. For instance, in a previous work [DCN+19], authors focus on DBI evasion

and escape problems in the context of Just-In-Time (JIT) dynamic compilation based user-space DBI. Our goal is different in nature – instead of focusing on the transparency and security of specific solutions, we are interested in analyzing and evaluating all known instrumentation techniques and their capabilities, mapping those instrumentation techniques to the most relevant publicly available implementations.

On the other hand, a fair amount of research papers that propose a new technique or DBI tool do not evaluate its performance, measure different metrics, or only report the relative overhead with respect to native execution using arbitrary test cases. Unfortunately, this is not sufficient to understand which approach might show better performance for a particular situation. In contrast, our interest is to take into consideration all existing techniques, and analyze their performance when instrumenting different types of events under different conditions, under a variety of programs, with different lengths, and a different frequency of observable events.

1.4 Hypothesis and Objectives

The problems described in previous sections represent the current boundary in the study of memory corruption, its concurrent version, and its detection. Therefore, we have formulated the following hypothesis:

> «*Memory errors remain as one of the main threats to current systems, their concurrent variant has largely gone overlooked, and is harder to detect.*»

From the formulated hypothesis, we have defined the general objective of the research for this dissertation as:

> **General objective 1** *Evaluate the evolution of memory errors over time and verify the security risk posed by concurrency in combination with memory corruption, and promote their detection.*

From the general objective, this research defines the following specific objectives:

> **Specific objective 1** *Analyze in-depth memory corruption, offensive and defensive methods, and the evolution over time in real programs.*

Specific objective 2 *Define the properties of concurrency vulnerabilities, the role of memory corruption in them, and analyze their existence in current software and their detection approaches.*

Specific objective 3 *Develop and evaluate a DBI system evaluation framework, and analyze the underlying foundations and techniques.*

1.5 Research Methodology

Our goal in this dissertation is to extend the knowledge in the field of memory errors. To do so, we need to acquire a deep understanding of the status of current memory corruption research, as well as to propose new analyses.

To achieve these goals, the research will be carried out using the following methodology:

1. **Knowledge Acquisition.** First, we must analyze the relevant and current research on memory errors. To do so, we will study in depth the state of the art on this topic to determine which are the lines of research in the community and, therefore, the challenges to be solved.

2. **Research questions.** Then, we will formulate research questions to be solved during the next steps, thus defining the lines to be followed and the challenges to be solved in this dissertation.

3. **Development of new analyses and tools.** We will develop new tools and analysis methods in order to address and solve the identified research questions.

4. **Analysis and evaluation of results.** After developing the necessary tools and methods, we will analyze and evaluate their usage and the produced results to answer the formulated questions.

5. **Gather feedback from the scientific community.** We will present the results of the research at international conferences and journals. In this way, we will get feedback from the scientific community about our work.

6. **Dissemination of results.** Linked to the previous step, we will communicate the final results of the research to the community.

1.6 Dissertation Outline

This dissertation is structured under the chapters listed below:

- **Chapter 1. Introduction.** In this chapter, we give an introduction to the area of memory corruption vulnerabilities, we describe how concurrency and memory errors are linked together, and also introduce the main concepts of dynamic binary instrumentation and current challenges. Then, we detail the hypothesis and objectives of the dissertation and describe the methodology for the research.

- **Chapter 2. The Never-Ending Threat of Memory Errors.** In this chapter, we describe in a comprehensive manner the details of memory corruption, review the state-of-the-art in memory corruption detection techniques, and detail further attacks as well as mitigation approaches. Furthermore, we describe the implementation of the developed tool to analyze the landscape of memory corruption vulnerabilities in real-world programs.

- **Chapter 3. Towards Concurrency Memory Corruption Vulnerabilities.** In this chapter, we describe concurrency bug characteristics and highlight the differences between bugs and vulnerabilities. We expand the previously developed tool and further analyze concurrency memory corruption vulnerabilities, revealing a growing trend. Moreover, we systematize the knowledge in concurrency memory corruption detection for C and C++ programs.

- **Chapter 4. Unveiling Dynamic Binary Instrumentation Techniques.** In this chapter, we propose a systematic study of DBI. We define the fundamental aspects of DBI, and identify, classify, analyze, and evaluate underlying instrumentation techniques. We build a framework to evaluate the performance of the techniques by testing a set of reference implementations and analyze the results to discover several interesting findings.

- **Chapter 5. Conclusions.** In the last chapter, we present the aggregated conclusions of this research and review the initially proposed hypothesis. Moreover, we expand on future work on this line of research.

Chapter

2

The Never-Ending Threat of Memory Errors

Among all the software issues that lead to security vulnerabilities, memory corruption is one of the most dangerous and oldest problems in which the systems security community has focused on, and despite the numerous approaches proposed to mitigate them, they are still found in generalized and specialized software [WXL+20, CRD20]. Memory corruption occupies the top positions in the Common Weakness Enumeration (CWE) most dangerous software weaknesses list [Com], appearing separately as out-of-bounds writes, out-of-bounds reads, Use-after-Frees (UAF), and so on.

Several factors influence the prevalence of memory errors, but one of the main reasons is that a large amount of proposed detection and protection solutions are not deployed in practice due to performance overhead, compatibility problems, lack of completeness in protection, or dependence conflicts [VdVCB+12, SPWS13].

In this chapter, we aim at bringing a broad comprehension into the field of memory corruption. To this end, we thoroughly describe memory safety, review existing memory corruption identification techniques, and expand on further attacks as well as mitigation approaches. Additionally, we analyze the landscape of memory corruption vulnerabilities in real-world programs.

The rest of this chapter is organized as follows. Section 2.1 introduces memory safety and enumerates memory safety violations, organizing them on the basis of their characteristics. Section 2.2 presents the wide variety of exploitation methods around memory errors along with the most relevant mitigation strategies, with an emphasis on sophisticated attacks such as those that reuse code already present in memory, and those that target non-control data. Then, section 2.3 categorizes and details the main methods used to detect memory errors. Section 2.4 describes the developed tool that automatically parses, analyzes, and classifies vulnerabilities from the CVE databases, along with the obtained results. Section 2.5 surveys analogous research works in the literature. Lastly, section 2.6 presents the summary of the chapter.

2.1 Memory Safety

Explicit memory management and pointer arithmetic are convenient features of the C and C++ programming languages that make them the ideal choice for many tasks. Their implementations enable developers to manage and access objects stored in memory manually, providing static, automatic, and dynamic memory allocation (and deallocation) methods. However, the improper use of these mechanisms leads to a range of memory-related errors that may result in unexpected program crashes and security vulnerabilities.

Memory safety is a property of programs that guarantees that memory accesses are always to defined and valid memory. Namely, memory objects are only accessed between their intended bounds, during their lifetime, and given their original or compatible type.

Languages with automatic memory management (e.g., with garbage collection) such as Java, abstract away memory management details and ensure memory safety in the implementation. On the contrary, the programmer and the compiler are responsible for handling this property in C and C++.

Taking into account the above definition, memory safety violations comprise any access to memory not defined by the program (i.e., at the time of dereference), and pointer dereferences to not designated memory regions. There are certain situations that infringe memory safety and lead to memory corruption. They are usually grouped into spatial errors, temporal errors, and type confusion vulnerabilities (which eventually lead to temporal

```
char buffer[10];
char input[15];

strncpy(buffer, input,
        sizeof(input));
```

(a) Example buffer overflow

```
char *p = (char *)malloc(
    5 * sizeof(char));

free(p);
p[0] = 'a';
```

(b) Example use-after-free

```
class Parent { int a; };
class Child: public Parent { int b; };

Parent *p = new Parent();

Child *c = static_cast<Child *>(p);
c->b = 0;
```

(c) Example type confusion

Figure 2.1: Memory errors code examples

or spatial safety violations).

2.1.1 Spatial Errors

Spatial memory safety violations take place when a pointer is used to access a location in memory that is outside the bounds of the intended allocated memory object, resulting in accesses to not designated memory areas (i.e., to that pointer) or undefined memory.

Buffer overflows are the typical representative instance of this class of errors, when the program writes past the buffer boundary. To illustrate, Figure 2.1a shows a simple buffer overflow example. The `strncpy` function is called to copy the characters from `input` to `buffer`. However, the `input` array is longer than `buffer` and will overwrite adjacent memory contents.

2.1.2 Temporal Errors

Temporal memory safety violations occur when the program uses a pointer to access a memory object out of its lifetime. For instance, when dereferencing a dangling pointer. Within this category, we can find use-after-free errors for accesses to deallocated memory objects, double-free errors, accesses to not initialized memory, and Null-pointer dereferences. Other more specific instances exist, such as use-after-return for accesses to local

objects after a function has returned.

Figure 2.1b shows a simple use-after-free example. Pointer p points to a dynamically allocated object in memory that is later freed in the execution. Afterwards, p is dereferenced, which results in undefined behavior and a possible exploitation opportunity.

2.1.3 Type Confusion

In languages that support the object-oriented programming paradigm such as C++, it is essential to be able to convert one object type to another, known as type conversion. Type confusion (or bad-casting) takes place when an object type is interpreted as another due to unsafe type casting, which results in the wrong reinterpretation of the underlying type representation.

Among the provided types of cast by C++, `static_cast` checks whether the conversion is valid at compile-time, and is commonly used in performance critical applications because of its efficiency. However, it does not verify the safety of the conversion at run-time.

Programs often cast an instance of a parent class to a derived class (i.e., downcasting), which is not safe if the parent lacks any of the fields or virtual functions of the derived class. When the program then uses any of these fields, it may use data as a regular field in one context and as a virtual function table pointer in another, enabling memory corruption. Figure 2.1c illustrates this situation.

2.2 Memory Corruption Attacks and Mitigations

Starting from the first discussion in 1972 on how buffer overflows overwrite memory and could be used to inject code, an arms race began in which both attacks and countermeasures have been evolving with memory errors as the central piece [VdVCB+12].

Triggering a memory error is usually the first step to carry out several attacks. By making a pointer go out of bounds or making it become dangling and then dereferencing it, a potential attacker can modify code pointers and variables, for instance [SPWS13]. Attacks are primarily intended to alter the execution of programs to make them behave in unintended ways, and usually take control. To this end, one of the most straightforward methods is to modify the program code that resides in memory through a buffer overflow, known as code corruption [SPWS13]. Never-

theless, this approach is conveniently prevented by setting code pages as read-only, leveraging the support from modern processors.

As a consequence of the increasing number of developed attacks over the years, several countermeasures have been equally proposed. However, many of them are not deployed in practice, or a less strict version is used, since they usually incur high performance overhead.

Control-flow hijacking forms the basis for plenty of attacks, making programs deviate from their intended control-flow [STL+15, GKK+18]. For example, an attacker can exploit a buffer overflow to overwrite the return address stored in the stack for the corresponding stack frame. When the function returns, the execution will continue at the new given address. To execute the code specified by the attacker, there are two strategies: code injection and code reuse. It is worth to mention that stack canaries are a widely used mechanism to prevent the corruption of return addresses in the stack. They insert crafted values around return addresses to later verify if they have been modified. However, stack canaries do not protect against read operations, making them vulnerable to memory disclosure, and attackers can leverage non-continuous overwrites, or other code pointers to carry out attacks [SPWS13].

2.2.1 Code Injection

Within control-flow hijack attacks, code injection relies on injecting malicious code into the memory of the program (e.g., writing to a buffer) and using the previously described control-flow hijacking technique to direct the control-flow to the injected code (i.e., overwriting a return address). This code traditionally spawned a shell.

The root of these attacks is that the x86 architecture does not discern code from data in memory. To protect systems against this type of attacks, the Write XOR Execute protection policy enforces that memory pages can be writable or executable but not both at the same time, effectively removing the executable permission from data pages like the stack. This protection policy is supported by modern CPUs that include the non-executable bit permission that operating systems can leverage.

2.2.2 Code Reuse

In order to circumvent the ubiquitous Write XOR Execute policy that renders code injection attacks ineffective, new attacks reuse the own code of

the program that is loaded in memory, chaining the execution of existing code pieces through code pointers [RBSS12].

The earliest implementation of this class of attacks is return-into-libc, which exploits memory corruption to make the program deviate from its control-flow (e.g., by overwriting a return address) and execute functions from the standard C library (i.e., libc). By means of chaining the execution of libc functions with attacker controlled arguments, attackers are able to effectively take control [Des97]. However, return-into-libc attacks have several limitations such as architecture compatibility, and inherent constraints in relying on the functionality of the standard C library.

Rather than reusing complete libc functions, Return Oriented Programming (ROP) generalizes code reuse attacks by targeting gadgets — interesting code sequences that end in a return instruction [Sha07, vdVAS+17]. These gadgets perform very specific tasks, and when the execution of these gadgets is chained, the program is able to execute high-level operations. On the basis of ROP, other code reuse variants have been developed, such as Jump Oriented Programming (JOP) [BJFL11], which targets gadgets that end in indirect branches instead of return instructions, or Block Oriented Programming (BOP) [IAJP18], that makes use of basic blocks as gadgets. These new techniques also seek to overcome emerging defenses proposed by the research community.

2.2.3 Control-Flow Integrity

To hinder control-flow hijacking, proposed approaches seek to preserve the integrity of the possible paths that programs may follow. The Control-Flow Integrity (CFI) policy relies on the construction of a Control-Flow Graph (CFG) of the program that represents the legitimate execution paths, and prevents any deviations from it during execution [BCN+17]. This is usually achieved through the instrumentation of indirect branches and the validation of all indirect control-flow transfers at run-time.

CFI implementations are often differentiated by the kind of policy they enforce. Forward-edge CFI is focused on jump instructions and indirect calls, whereas backward-edge CFI preserves the integrity of control-flow transfers from return instructions. Further, fine-grained CFI enforces a more strict policy [PBG15], whereas coarse-grained CFI is less restrictive for better performance [VDVGC+16]. In order to improve overall performance and gain adoption, researchers have explored the application of hardware features, successfully enforcing CFI using the Intel Processor

Trace [GCJ17] for example. Unfortunately, more sophisticated code reuse attacks such as Counterfeit Object-Oriented Programming (COOP) are able to bypass many existing implementations of CFI, taking advantage of object semantics in C++ programs [STL+15]. In contrast to other code reuse techniques, COOP uses virtual functions as gadgets.

2.2.4 Randomization-based Defenses

As an additional line of defense, randomization-based approaches make exploitation harder through probabilistic techniques. Since attackers need to know the specific addresses of their targets (i.e., the location of gadgets for code reuse attacks), randomizing the location of regions in memory renders them ineffective. Address Space Layout Randomization (ASLR) is a widely deployed method in modern operating systems that randomizes the process memory layout by randomly placing memory regions such as program code, the stack, heap, and shared libraries. The more fine-grained randomization, the higher performance overhead.

However, attackers successfully identified and exploited weaknesses on these approaches. First, address randomization in 32-bit systems is susceptible to brute-force attacks due to low entropy [SPP+04]. In addition, taking advantage of memory corruption, attackers can leak memory contents to uncover the memory layout of the program and successfully launch attacks [SMD+13]. Since Write XOR Execute and ASLR emerged as two of the most deployed defenses, attackers increased their focus on memory disclosure attacks, making it possible to leak code pointers or the contents of code pages and find the required gadgets to execute a code reuse attack.

In light of these issues, researchers enhanced existing randomization techniques to not be vulnerable to memory disclosure attacks. In that context, newer approaches such as `ASLRGuard` use a shadow stack to protect code pointers [LSL+15]. Alternative approaches rely on systematic re-randomization of the address space to invalidate current code pointers [CBG17], which prevents the attacker from exploiting the obtained knowledge about the program. However, attacks evolve at the same time, successfully bypassing modern randomization techniques by directly reusing the randomized code pointers [GKK+18].

In contrast, other approaches seek to provide probabilistic temporal safety guarantees by implementing safe memory allocators. For example, `DieHard` [BZ06] randomizes the locations of the program allocated objects in the heap.

2.2.5 Non-control Data Attacks and Mitigations

Different from the aforementioned attacks, non-control data attacks (also called data-only attacks) do not seek to hijack the control-flow of the program to carry out malicious operations. Consequently, existing mitigations that are focused on preventing alterations to the intended control-flow of the program are not effective against this class of attacks.

The ultimate goal of data-only attacks is to modify the data-flow of the program by altering security-critical data in memory, such as user input, configuration data, decision-making values, and user identities, for instance. As a result, attackers are able to escalate privileges, circumvent authentication, or leak sensitive information [CXS+05].

`FlowStitch` [HCA+15] is a practical approach that automates the creation of data-only attacks through the exploitation of memory errors to maliciously join existing data-flow paths to leak or alter relevant data in the program. Afterwards, Data Oriented Programming (DOP) [HSA+16] generalized non-control data attacks by systematically building exploits in a Turing-complete fashion using techniques to find gadgets and to chain them together in a conceptually similar vein to ROP, except that it does not modify any code pointers. Similarly, the code reuse technique BOP is also used to automatically mount successful data-only attacks, following the valid Control-Flow Graph (CFG) of the program but not its Data-Flow Graph (DFG).

Equivalent to CFI for control-flow hijacking attacks, Data-Flow Integrity (DFI) seeks to protect against data-only attacks by enforcing that program data only follows legitimate data-flows. To this end, DFI uses reaching definitions data-flow analysis to construct a Data-Flow Graph (DFG). The DFG is formed by the set of load instructions, and for each load, the set of store instructions that can reach it. In other words, it maps the instructions to written values and uses of these values. By instrumenting read and write instructions (usually at compile-time), DFI verifies that data-flows are within the DFG at run-time [CCH06, SLL+16]. Unfortunately, DFI faces similar problems to CFI having high memory and performance overheads.

2.2.6 Additional Mitigations

Several alternatives exist to the previously mentioned mitigation strategies. For instance, pointer integrity policies are an efficient approach to protect program pointers. This method is used to protect code pointers against

control-flow hijacking [KSP+18] and even data pointers for a more comprehensive protection [LNW+19].

Software Fault Isolation (SFI) is another method that isolates untrusted modules in a sandbox in the address space of the host program, in such a way that it cannot perform memory accesses out of the sandbox. To illustrate, `Google NaCl` [SMB+10] loads untrusted code into a specific memory area, and confines its memory and instruction references to the sandbox through instrumentation. Alternatievly, `ARMlock` [ZWCW14] leverages ARM hardware support to set up sandboxes by assigning memory to different domains, and locking the execution to the current domain.

Furthermore, researchers have also brought memory safety to the C unsafe language in a variety of ways. For instance, `CCured` [NMW02] is a type system that leverages programmer annotations and fat pointers to ensure program safety. `Cyclone` [JMG+02] is a safe dialect of the C programming language that also extends the pointer representation with additional information to perform several checks. Unfortunately, these approaches break binary compatibility.

2.3 Memory Corruption Finding Techniques

Memory safety issues are of great relevance to computer systems. This is also evidenced by the large amount and diversity of approaches proposed to find them in the research community [SPWS13, SLR+19]. These approaches seek to identify security vulnerabilities in software before it is finally deployed, and are based on static program analysis, dynamic program analysis, or a combination of both.

Finding bugs in operating system kernels presents unique challenges compared to user-level applications. This is attributed to their extensive and heterogeneous codebases, interaction with devices, privilege levels, and so on. Nevertheless, several approaches have been developed that specifically target kernel errors by building a *soundy* static analysis technique for taint and pointer analysis [MSC+17], or a dynamic analysis framework to hook and fuzz kernel drivers [SHD+19], for instance.

During the rest of this section, we mainly discuss the most representative general-purpose memory safety enforcement and error identification approaches not tied to specific software.

2.3.1 Fuzz Testing

Among the dynamic analysis techniques, fuzzing has been certainly one of the most active research topics in recent years [MHH+19]. From a broad perspective, the core of the iterative process of fuzzing has three different phases. First, it generates an input, or a set of inputs (i.e., test cases), to be fed to the program under test. Then, it executes the program given the generated inputs. Last, the observations produced from the execution are evaluated to determine whether the given input is interesting or not, so that the input generation subsystem will use it for further executions or not. Moreover, if the program crashes, it generates a crash report with relevant information to analyze and reproduce it. Each of the stages can be further divided into more specific tasks, we encourage the reader to refer to the literature for more in-depth information [KRC+18, MHH+19]. Existing fuzzing techniques are also usually classified into black-box, grey-box, and white-box approaches depending on the information they observe and use during the process.

Dynamic analysis approaches are only able to detect vulnerabilities in a concrete execution, meaning that they only explore a small subset of the possible program paths. Considering that fuzzing usually aims at exploring as many paths as possible and uses code-coverage metrics, combining it with sanitizers greatly boosts the ability to find memory errors [DBXP20, ÖRBG20]. For the specific concurrency memory corruption vulnerabilities, current fuzzers have also successfully incorporated instrumentation and feedback mechanisms specific to threads and scheduling [CGX+20].

2.3.2 Static Approaches

Static analysis vulnerability detection refers to the set of program analysis techniques that are devised to analyze program code (i.e., in either high-level language, intermediate representation, or binary form), and reason about different program properties to check memory safety, producing results that are conservatively correct for every possible execution. Although static analyses are able to provide more complete results by taking into account more execution paths than dynamic analysis, they suffer from a higher false alarm rate, and usually need to make tradeoffs among soundness, precision and scalability.

Among the static approaches developed to detect memory corruption (i.e., each taking into account the different semantics of the different spa-

tial and temporal errors), the most notable used techniques are: (i) abstract interpretation [Cla], which approximates the behavior of program execution and enables the computation of invariants; (ii) model checking [KT14], that computes and reasons about all program states; (iii) pointer analysis [YSCX18], which seeks to approximately determine the possible values of a pointer; and (iv) pattern matching [PLHM08], that reasons and finds occurrences of the given patterns.

2.3.3 Symbolic Execution

In a symbolic execution, programs are executed symbolically. That is, input values are represented as symbolic values instead of concrete values and the execution is carried out by a symbolic execution engine. These values are used to generate path conditions, which are logical expressions that represent the program state and the transformations between them. To implement symbolic execution, it is necessary to maintain a symbolic domain, which is composed of a symbolic state that maps variables to symbolic expressions, and a path condition formula for each program path. Conditional statements update the path condition formula, whereas assignments update the symbolic state. Several approaches exist within symbolic execution, such as forward and backward execution, or concolic execution [BCD+18].

Generally, symbolic execution is used to verify properties of programs (e.g., memory safety) and to automatically generate test cases to augment other dynamic analyses, for instance. In conjuntion with vulnerability models, researchers have successfully used symbolic execution to find memory errors such as heap overflows [JZS+17] and to also generate memory corruption exploits [SWS+16].

2.3.4 Sanitizers

In contrast to static analysis, sanitizers are dynamic analysis tools that instrument programs, usually at compile-time or at binary-level by leveraging Dynamic Binary Instrumentation (DBI) frameworks, to insert the so-called inlined reference monitors and perform several checks at run-time. In this way, they analyze an actual program execution and produce accurate results with regards to that specific execution.

Over the years, a great number of sanitizers have been proposed in order to identify and diagnose memory safety violations, each implementing

methods to find spatial memory corruption, temporal memory corruption, or both. Typically, these approaches create and maintain metadata that describes the spatial or temporal properties respectively of a pointer or object.

In relation to spatial memory safety, we can mostly discern two kinds of approaches: those based on whitelisting memory regions, and those based on blacklisting. At the same time, existing approaches either maintain metadata for each pointer, or for each memory object. For instance, whitelisting and per-object bounds tracking approaches [ACCH09] store bounds metadata for each object in a bounds table, and check it on pointer arithmetic. As an alternative, per-pointer bounds tracking (i.e., SoftBound [NZMZ09]) stores bounds metadata for each pointer in a bounds table, propagates that metadata on pointer arithmetic, and checks it on each dereference. In contrast, blacklisting techniques [SBPV12, SN05, BZ11] insert poisoned red-zones around memory objects, track the state of the program using a shadow memory, and check the shadow state on each memory access.

With regards to temporal memory safety, the main used techniques are lock-and-key and memory quarantining. Lock-and-key approaches assign a unique key to memory objects and store that value in a lock location in memory, keeping that metadata for each pointer, and checking on each pointer dereference [NZMZ10]. When objects are deallocated the corresponding key is no longer valid. Instead, quarantine-based approaches put memory regions into quarantine as soon as they are deallocated, inserting red-zones, and delaying their possible allocation [SBPV12, NS07] while checking memory accesses.

In order to detect type confusion, sanitizers add run-time checks to detect type casts that are incompatible. Existing tools either use run-time type information [Und] or track type metadata for each object [HJP+16, JBC+17] to verify `static_cast` operations, or check type compatibility on each pointer dereference [DY18].

We encourage the reader to refer to existing literature to find further information on sanitization techniques and implementations [SLR+19].

2.4 The Landscape of Memory Corruption

In order to observe and study the landscape, evolution, and possible trends of memory corruption vulnerabilities in real-world software, we have developed a series of scripts that automatically download, parse and analyze

all Common Vulnerabilities and Exposures (CVE) entries from 2002 to the end of 2022 from the National Vulnerability Database (NVD) data feeds [nvd].

2.4.1 CVE Parser and Classifier

Taking advantage of the publicly available data feeds from the NVD, our CVE parser is able to download year by year historic and up-to-date CVE data in JSON format. In addition, we also download all the public exploits linked to their corresponding CVEs from The Exploit Database (exploit-db) repository [exp]. After the first import of the complete data set, whenever the CVE parser is invoked, it checks whether any of the files has been updated with any addition or modification of vulnerabilities to get the most recent data.

Once we have all the available data, the CVE parser parses each entry from each of the files. First, it checks whether the entry has a description and a correct published date and discards the entry if not. It also discards entries that contain the string "DO NOT USE THIS CANDIDATE" in the description, since some CVE entries are sometimes duplicates, are not associated with any vulnerabilities, or are withdrawn. Then, it extracts relevant fields and information to enable the following classification of the entry. The retrieved information includes: description text, published date, Common Vulnerability Scoring System (CVSS) information, available exploits, and the list of Common Weakness Enumeration (CWE) identifiers.

We classify vulnerabilities by looking at their assigned Common Weakness Enumeration (CWE) identifiers combined with heuristics that use pattern matching using regular expressions. Table 2.1 shows the complete list of CWE identifiers along with their names and their memory corruption category. We separate memory corruption vulnerabilities into those that take place in the stack, the heap, integer-related problems, pointer usage vulnerabilities, format string vulnerabilities, and generic memory corruption that does not fall into any of the above categories because vulnerability details are not specific enough.

Unfortunately, it is nearly impossible to obtain fully accurate results in terms of classification for several reasons. To begin with, descriptions and vulnerability reports do not follow a particular standard and therefore are occasionally unstructured or contain incomplete information. Furthermore, many CVE entries do not have a CWE assigned. In addition, searching for overly simple keywords such as "stack" or "heap" is not appropri-

Table 2.1: Common Weakness Enumeration (CWE) identifiers for memory corruption.

Category	ID	Name
Stack	CWE-121	Stack-based Buffer Overflow
Heap	CWE-122	Heap-based Buffer Overflow
	CWE-415	Double Free
	CWE-416	Use After Free
Integer	CWE-190	Integer Overflow or Wraparound
	CWE-191	Integer Underflow (Wrap or Wraparound)
	CWE-193	Off-by-one Error
	CWE-680	Integer Overflow to Buffer Overflow
Pointer	CWE-476	NULL Pointer Dereference
	CWE-590	Free of Memory not on the Heap
	CWE-763	Release of Invalid Pointer or Reference
	CWE-822	Untrusted Pointer Dereference
	CWE-823	Use of Out-of-range Pointer Offset
	CWE-824	Access of Uninitialized Pointer
Format String	CWE-134	Use of Externally-controlled Format String
Generic	CWE-119	Improper Operations within the Bounds of a Buffer
	CWE-120	Buffer Copy without Checking Size of Input
	CWE-124	Buffer Underwrite ('Buffer Underflow')
	CWE-125	Out-of-bounds Read
	CWE-126	Buffer Over-read
	CWE-127	Buffer Under-read
	CWE-457	Use of Uninitialized Variable
	CWE-787	Out-of-bounds Write
	CWE-788	Access of Memory Location After End of Buffer
	CWE-805	Buffer Access with Incorrect Length Value

ate because they may wrongly match cases where they are used to describe software functionality or other unrelated execution aspects. Consequently, more sophisticated patterns are needed.

Bear in mind that our goal is not to develop the most accurate and comprehensive possible tool, but to obtain a general understanding of the landscape and the evolution of memory corruption vulnerabilities in the real-world. Therefore, even though it may not yield the finest results and more exhaustive techniques could have been used, it still provides useful and valuable insights.

Table 2.2: Number of reported CVEs for the selected memory corruption categories and the total reported vulnerabilities.

Category	# Vulnerabilities
Stack	4,322
Heap	9,579
Integer	3,533
Pointer	3,350
Format String	814
Unclassified	23,707
Total memory corruption	42,989
Total	203,652

2.4.2 Evolution and Breakdown of Memory Corruption

Using the developed CVE parser, we maintain a map of vulnerabilities to their reported dates along with their class and any other relevant information. We collected statistical information about the reported vulnerabilities ranging from 2002 to the end of 2022. Note that there are many different possible categories and subclasses, as well as many possible classification criteria. For simplicity, we chose to classify memory corruption vulnerabilities into stack, heap, integer, pointer, format string, and generic memory corruption. We believe that these are well-known categories that fairly represent the different schemes of memory corruption vulnerabilities.

Table 2.2 shows the number of reported vulnerabilities in the NVD database for each of the memory corruption categories that we selected. It is worth to note that according to our classification method described above, generic memory corruption, which we refer to as unclassified in Table 2.2, is the largest category incontestably. By manually inspecting CVE entries, we found that this is because often vulnerabilities do not have any CWE identifiers assigned or use some of the generic identifiers that we listed in Table 2.1, and vulnerability descriptions are not detailed enough.

Figure 2.2 depicts the total number of reported memory corruption vulnerabilities compared to the total number of reported vulnerabilities per 3 months from 2002 to 2022. We group reported vulnerabilities together over 3 months instead of on a monthly basis to minimize fluctuations. Figure 2.2 shows that the number of reported memory corruption vulnerabilities has been consistent over the years until 2015, when it experienced a marked

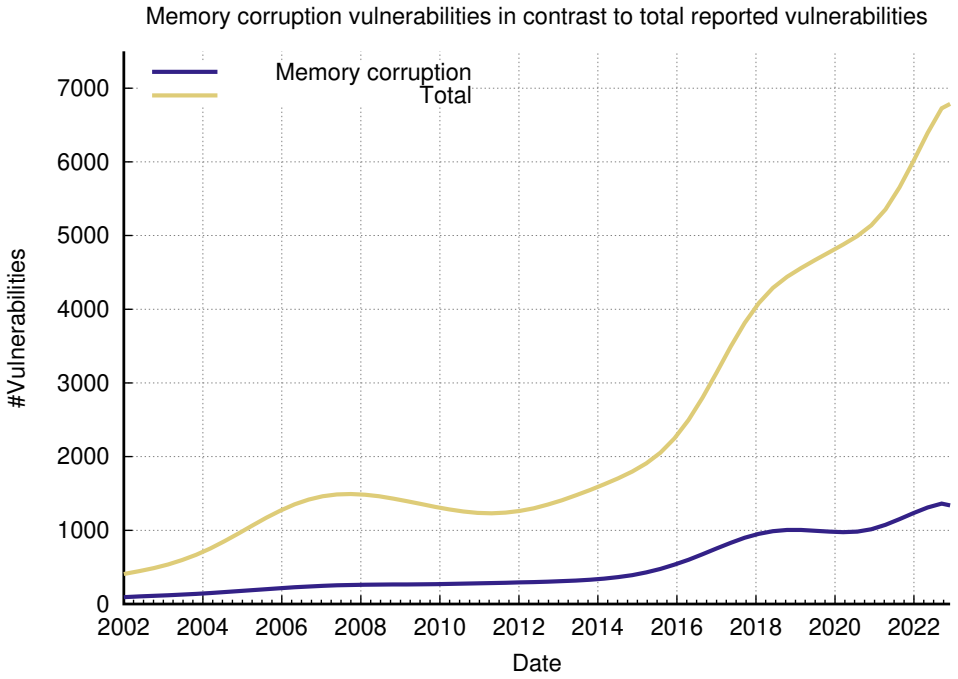


Figure 2.2: Growth of the number of reported memory corruption vulnerabilities and total reported vulnerabilities over time.

growth that resulted in around 1000 vulnerabilities being reported per 3 months. This growth can be attributed to the substantial growth in the number of total reported vulnerabilities.

Figure 2.3 illustrates the percentage of individual categories over the total reported memory corruption vulnerabilities. Unclassified vulnerabilities account for 52.33% of the total as expected, whilst the second largest category are heap-related vulnerabilities with a 21.14% over the total. This may suggest that properly dealing with heap operations and identifying issues during development is harder than with the stack for example, and therefore more vulnerabilities emerge and reach production versions of software. In contrast, format string vulnerabilities are almost non-existent with a 1.80% over the total.

Table 2.3 lists the number of publicly available exploits for each of the selected memory corruption categories. We can observe that the total number of 12,610 available exploits is significantly lower than the total number of reported memory corruption vulnerabilities (i.e., 42,989). Figure 2.4 and Figure 2.5 illustrate the percentage of memory corruption vulnerabilities and exploits over the total reported vulnerabilities and available exploits,

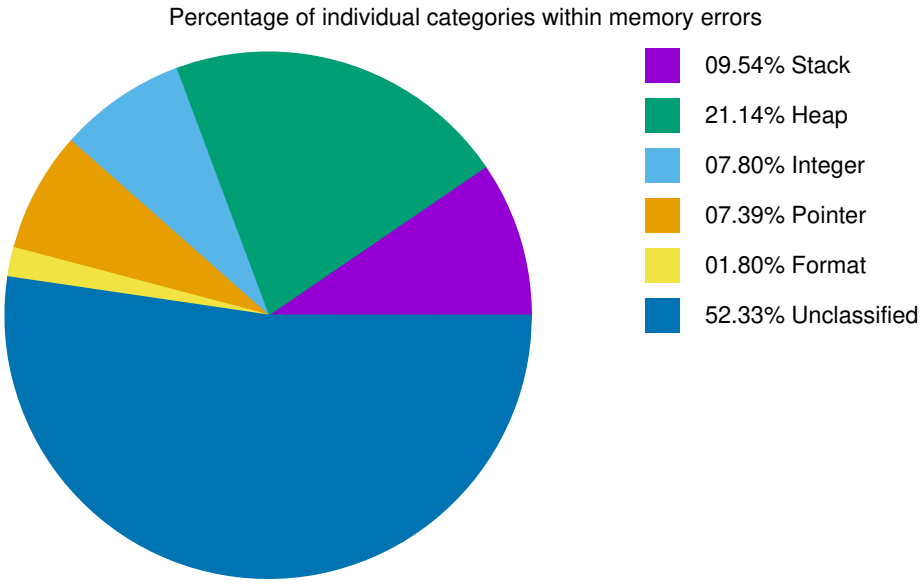


Figure 2.3: Percentage of each memory corruption category over the total reported memory corruption vulnerabilities.

and the percentage of exploits for each individual category over the total memory corruption exploits respectively. The number of memory corruption vulnerabilities and available exploits follow approximately the same trend, experiencing their highest point in 2017. They account for roughly 20% of all reported vulnerabilities and publicly available exploits in a consistent fashion over the years. This clearly shows that memory errors are a major threat to systems since their early detection, and are unlikely to become less of a threat in the near future.

Figure 2.5 breaks down memory corruption exploits by category. It shows similar rates to those of the vulnerabilities as depicted in Figure 2.3. The most remarkable difference is that the percentage of stack exploits is somewhat higher than its corresponding percentage of vulnerabilities. Specifically, we found 2,053 stack-based exploits compared to 4,322 stack-based vulnerabilities. Note that a single vulnerability may have multiple exploits assigned. This indicates that the stack is one of the preferred targets to exploit, possibly due to its expressiveness for complex attacks or to an increased ease of exploitation in relation to other vulnerabilities.

Table 2.4 lists the number of memory corruption vulnerabilities for each

Table 2.3: Number of publicly available exploits for the selected memory corruption categories.

Category	# Exploits
Stack	2,053
Heap	2,946
Integer	1,170
Pointer	1,282
Format String	279
Unclassified	4,880
Total memory corruption exploits	12,610

CVSS score range. The Common Vulnerability Scoring System (CVSS) is a standard measurement system for vulnerability severity calculation. CVSS base scores are calculated using multiple exploitability and impact metrics, and represent the innate characteristics and severity of vulnerabilities. All scores range from 0 to a maximum of 10. Vulnerabilities with a score of 0

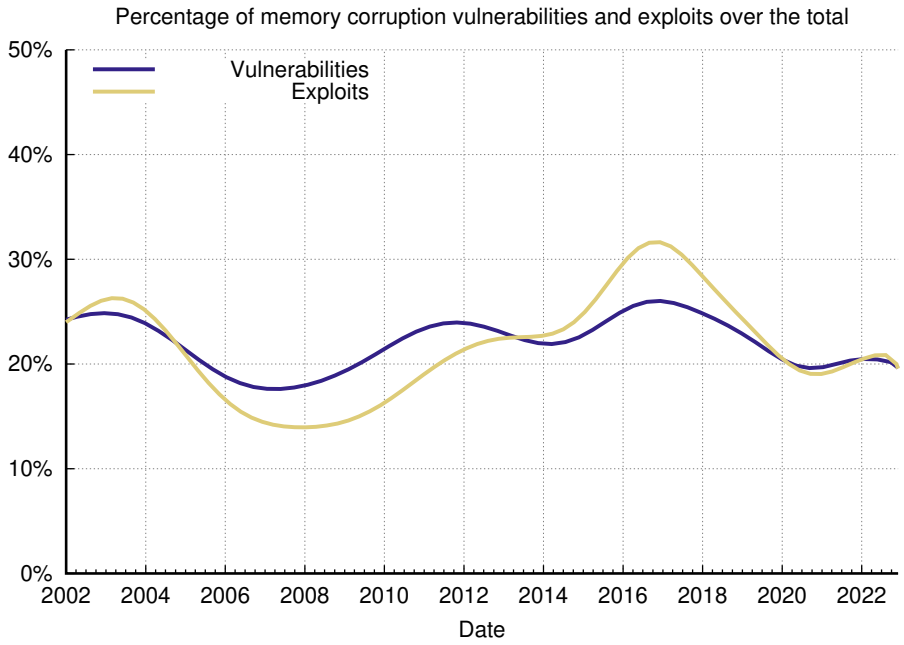


Figure 2.4: Percentage of memory corruption vulnerabilities and exploits over the total reported vulnerabilities and exploits.

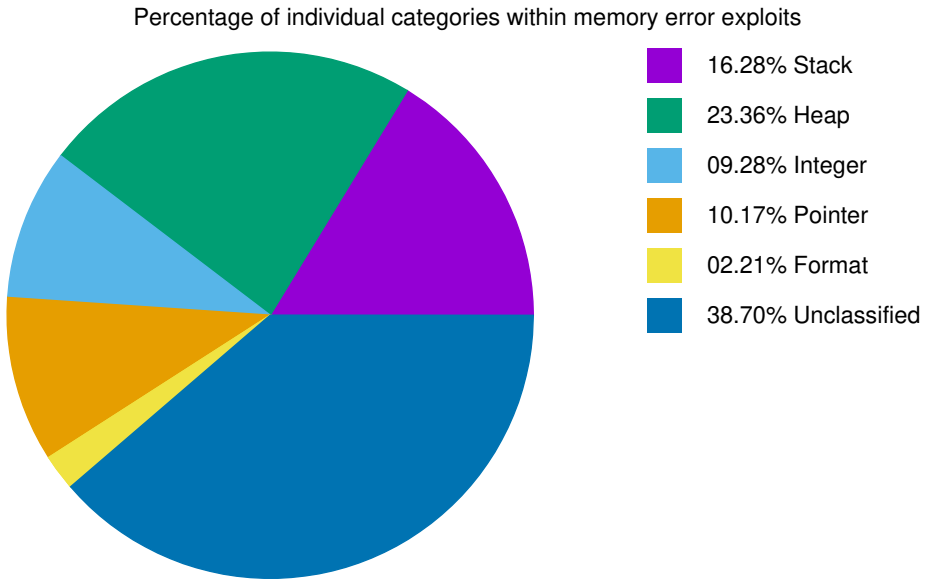


Figure 2.5: Percentage of each memory corruption category over the total reported memory corruption exploits.

are considered as non severe. Vulnerabilities with a score from 0.1 to 3.9 are considered as having low severity. Vulnerabilities with a score from 4 to 6.9 are considered as having medium severity. Vulnerabilities with a score from 7 to 8.9 are considered as having high severity. Finally, vulnerabilities with a score from 9 to 10 are considered critical. Unfortunately, not every CVE entry has a CVSS score specified. For instance, out of the 42,989 memory corruption vulnerabilities identified, only 30,387 had CVSS information available.

Figure 2.6 depicts the overall distribution of memory corruption vulnerabilities over the different severity levels based on their CVSS base score. We can observe that most memory corruption vulnerabilities are considered as high severity vulnerabilities. In particular, out of the total of 30,387 vulnerabilities, 22,659 have a score between 7 and 10, and 5,350 of them are considered critical. These findings further strengthen the concept that memory errors are a mayor threat to computer systems. In addition to accounting for a major part of all reported vulnerabilities over the years, they also have catastrophic consequences for the affected systems.

Table 2.4: Number of memory corruption vulnerabilities for each CVSS score ranked from 0 to 10.

Severity	CVSS Score	# Vulnerabilities
None	0	0
Low	0.1-1	0
	1-2	0
	2-3	24
	3-4	318
Medium	4-5	661
	5-6	3,731
	6-7	2,994
High	7-8	12,410
	8-9	4,899
Critical	9-10	5,350

2.5 Related Work

Memory errors have been a popular topic of research for more than three decades. Therefore, in addition to proposing and evaluating different approaches, researchers have also carried out multiple works that review, survey, or systematize the knowledge on the different areas within this field.

For instance, Szekeres et al. [SPWS13] present a comprehensive analysis of memory corruption in which they develop a general model for attacks and protection policies. For programs in their binary form, angr [SWS+16] is an extensive framework that implements numerous techniques to identify and exploit memory safety violations. From a different perspective, Van der Veen et al. [VdVCB+12] elaborate on the historical evolution of memory corruption. They point out that memory errors are unlikely to lose significance in the near future, and that despite all the proposed mitigations, attackers still manage to exploit memory errors.

In the context of detection techniques, Song et al. [SLR+19] bring together the knowledge about sanitizing methods and implementations. They present a comprehensive taxonomy of existing tools, highlighting several fundamental characteristics. Baldoni et al. [BCD+18] survey the essentials of symbolic execution and the most representative techniques for testing and security applications, whereas Klees et al. [KRC+18] thoroughly eval-

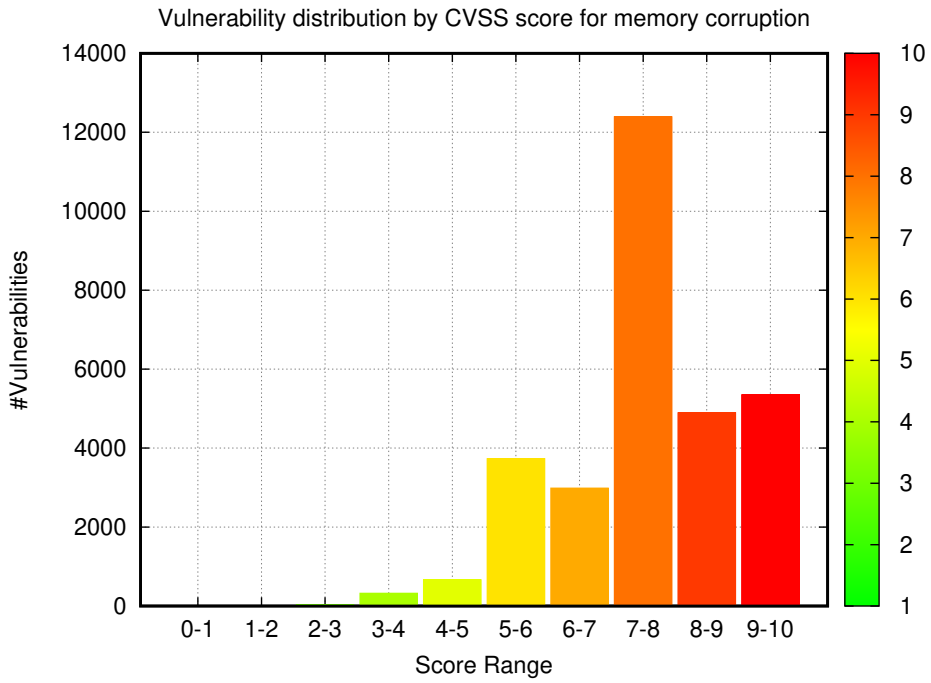


Figure 2.6: Distribution of memory corruption vulnerabilities by CCSS score.

uate existing fuzzing implementations and their experimental methodologies.

In relation to attack mitigation strategies, Larsen et al. [LHBF14] systematize the understanding on software diversity techniques and point out fundamental trade-offs in fully automated approaches.

2.6 Summary

Memory safety violations have been a major source of vulnerabilities historically and they are still in current systems. Over the decades, researchers have devised numerous methods to find and patch memory errors, or to protect systems from their exploitation. At the same time, attacks that are built upon memory corruption have been constantly evolving to circumvent newer defenses.

In this chapter, we have scrutinized the characteristics and details of memory errors, and the different attacks that exploit them, showing how attackers have devised new and increasingly complex attacks over the years. We have also analyzed the main detection approaches to prevent their ex-

ploitation, and current techniques to mitigate subsequent attacks. Despite all the proposed approaches, memory corruption persists and is actively exploited, suggesting that memory safety is still a serious concern, and that further research is needed. Our analyses of reported memory corruption vulnerabilities on the CVE database further confirm this perception and show the extent to which memory errors are present in programs over the years and their severity.

Chapter

3

Towards Concurrency Memory Corruption Vulnerabilities

Memory errors remain as one of the main and most dangerous threats to computer systems. Unfortunately, concurrency makes it even worse since enables memory corruption to take place in new and more difficult-to-detect ways. Concurrency errors and detection approaches have been extensively studied. However, concurrency vulnerabilities — especially concurrency memory corruption — have not received nearly as much attention. More worryingly, an increasing number of concurrency vulnerabilities are being reported each year. Therefore, a comprehensive analysis of concurrency memory corruption vulnerabilities and known detection methods remains an important open task. A systematic study in this field will lead to a better understanding and to the development of new detection approaches.

In this chapter, we aim at filling this gap by highlighting the differences between bugs and vulnerabilities, revealing the existence and growing trend of reported concurrency vulnerabilities in widely used software, and systematically analyzing the state-of-the-art in concurrency memory corruption detection for C and C++ programs.

The remainder of this chapter is organized as follows: Section §3.1 depicts the general taxonomy of concurrency bugs and reviews the state-of-the-art detection approaches. Section §3.2 details the characteristics of concurrency memory corruption vulnerabilities, and analyzes reported vulnerabilities in the CVE database. Section §3.3 explores the subtleties of exploiting concurrency vulnerabilities. Section §3.4 describes and classifies state-of-the-art approaches towards concurrency memory corruption detection. Section §3.5 discusses the most remarkable points with regard to the presence of concurrency memory corruption in real-world programs and its detection approaches. Section §3.6 reviews the different works that are related to this part of the dissertation. Finally, Section §3.7 concludes the chapter.

3.1 Concurrency Errors

Concurrency errors are common in multithreaded programs and lead to unintended program behavior, resulting in wrong outputs, program crashes, and security vulnerabilities, for instance. Explicitly exploring their consequences revealed that they often lead to memory corruption [ZSL10], which suggests that traditional memory error detection approaches may fail detecting these specific cases since they are usually triggered under a certain scheduling sequence.

Although a concurrency error is not a necessary requirement to trigger a concurrency vulnerability [CZM+19, CGX+20], their characteristics are remarkably similar. Moreover, many concurrency vulnerability detection approaches rely on or adapt existing concurrency error detection methods.

3.1.1 Classification

According to their observable properties, concurrency errors are usually classified into deadlocks, data races, atomicity violations and order violations [LPSZ08].

Deadlocks

Deadlocks arise as a consequence of improper coordination in the use of a synchronization mechanism (i.e., mutex locks). Specifically, a deadlock materializes when a set of threads are circularly waiting to each other to

obtain a resource (i.e., a lock) that is held by another thread, not being able to continue the execution.

Data Races

Data Races take place when multiple memory accesses from different threads access the same memory location (i.e., a shared variable) simultaneously without proper synchronization, and at least one of them is a write operation. As a result, memory operations may get their intended order changed, leading to inconsistent memory views from different threads, unexpected memory values, and so on. Surprisingly, data races are often intentionally introduced in programs (e.g., for optimization) [SBN+97].

It is worth noting that data races and race conditions are different concepts, even though they are often used interchangeably. While data races are exclusive to improper synchronization in concurrent memory accesses with at least one memory write, race conditions refer to timing dependent behavior in the execution.

Atomicity Violations

Atomicity Violations materialize when the desired serializability among multiple operations in a concurrent execution is interleaved with operations from another thread (i.e., a code region is assumed to be atomic, but the atomicity is not enforced during execution). Atomicity violation bugs are closely connected to unserializable interleavings, which are interleavings that are not equivalent to any sequential execution of the involved operations [LTQZ06, PLZ09].

Order Violations

Order Violations, in a conceptually similar vein to atomicity violation errors, take place when the desired order for a set of concurrent operations is assumed but not enforced during the execution. For instance, when a memory access is expected to precede another but the order is flipped at runtime.

3.1.2 Detection Approaches

Over the years, many different approaches have been proposed towards the challenge of identifying concurrency errors in multithreaded programs.

Some try to detect or predict specific bug categories whereas others work towards automatically fixing concurrency bugs. In reality, there is no absolute solution, but the different approaches have their own strengths and weaknesses in terms of false negatives, false positives, and performance. Interestingly, we found that Dynamic Binary Instrumentation (DBI) techniques are widely used in the development of concurrency error detection approaches.

We focus on non-deadlock bugs since their characteristics are the most similar to concurrency vulnerabilities, and have been shown to lead to memory corruption and even to concurrency vulnerabilities. In practice, there is a widely accepted way to handle deadlocks by monitoring the lock-wait time and restarting the corresponding thread or process when necessary. However, we encourage the reader to refer to existing literature for further information on deadlock detection and mitigation [JTZ+08, CC12]. Since this work is focused on unsafe C and C++ programs, we also do not consider approaches for other languages such as Java [CYW+21].

3.1.2.1 **Data Races**

Data races have been the focus of most concurrency bug detection works [PFH06, LPSZ08, FF09, SI09, HMR14], proposing many different static, dynamic, and systematic testing (e.g., stateless model checking) methods.

Static approaches are intended to approximate the runtime behavior of the program without it being executed, obtaining thread identifiers, identifying synchronization operations and checking memory operations that potentially access the same memory location statically. For instance, `RacerX` [EA03] uses flow-sensitive inter-procedural analysis to infer lock-protected operations, multithreaded contexts and dangerous shared accesses, whereas `LOCKSMITH` [PFH06] uses context-sensitive correlation analysis to determine whether the accessed memory locations are consistently protected by locks. Unfortunately, pure static approaches usually produce a high number of false alarms, increasing manual effort.

Dynamic solutions reason about synchronization operations and memory accesses in specific execution traces and have been proven to be very effective in data race detection. Existing algorithms can be broadly classified into three groups: *sound predictive analysis*, *lock-set analysis*, and *partial order* based techniques.

Firstly, *sound predictive analyses* are intended to explore all possible feasible reorderings of a given trace, thus being sound and complete in the-

ory. In practice, these techniques are expensive since each trace has exponentially numerous valid reorderings. Therefore, completeness is traded for performance by slicing traces into small fragments and analyzing these fragments individually. These approaches usually formulate race detection as a constraint solving problem and rely on SAT/SMT solvers [SWYS11, HMR14] to detect data races.

Secondly, *lock-set* based approaches track the set of locks that are held during memory accesses in the execution. They report a potential race whenever two threads access the same memory location without proper locking [SBN+97, EQT07].

Thirdly, *partial order* based methods formulate a partial order P on the events in a given trace so that event pairs unordered by P conform data races. The most common implementation uses the *happens-before relation* (HBR). HBR detects a data race when two conflicting accesses (i.e., accesses from different threads to the same memory location including at least one write operation) are not causally ordered so that neither is forced to happen before the other, meaning that a reordering exists. Many tools are based on HBR and have explored different ways to implement it efficiently [FF09, BCM10]. In an effort to overcome the limitations of HBR, other *partial order* techniques such as *schedulable happens-before* [MKV18], *weak-causally-precedes* [KMV17], and *doesn't commute analysis* [RGB18] have been proposed in recent years.

Lastly, hybrid approaches such as `ThreadSanitizer` [SI09] and `Helgrind` [hel] combine *lock-set* and HBR analyses to improve detection capabilities and are widely used in practice.

Other techniques aim at exposing concurrency bugs by systematically exploring different executions and multiple thread interleavings. For instance, testing methods for concurrent programs are based on some sort of controlled scheduling, since ordinary testing is inadequate due to nondeterminism [LLM+19]. To this end, existing solutions modify thread priorities dynamically [LZL+18, CGX+20], introduce delays at specific locations [LLM+19], or explore the interleaving space in a randomized way [NBMM12]. `Conzzer` [JBLH22] is a fuzzing framework for concurrent programs that uses a new context-sensitive coverage metric based on pairs of concurrently executed functions, augmented with their calling contexts (i.e., the runtime call stack). To explore as many thread interleavings as possible, it uses an adjacency-directed mutation strategy of the already covered concurrent call pairs to infer other potential concurrent call pairs. Then, it leverages previously injected breakpoints at function entries to

tamper with thread scheduling, to try to cover these new thread interleavings. To detect data races, it implements a dynamic *lock-set* analysis on the identified concurrent function pairs, and verifies them by injecting breakpoints at the racy instructions.

Moreover, controlled concurrency testing and stateless model checking are both actively and effectively applied techniques in the industry [ADH+21, BJA+21]. To illustrate, `Nekara` [ADH+21] is an open-source library that facilitates the development of systematic testing solutions for multiple platforms. It provides an API to model the set of supported concurrency primitives, and encapsulates state-of-the-art search heuristics to explore the interleaving space from existing frameworks (e.g., in a randomized fashion, or focusing on schedules with limited context switches).

3.1.2.2 **Atomicity Violations**

Following data races, atomicity violations are the second most discussed class of non-deadlock concurrency errors in the literature. Atomicity violation bugs widely exist because which part of the code needs to be atomic depends on programmers' intentions, who are often used to sequential thinking and assume non-atomic code regions to be atomic. Therefore, one of the most challenging tasks for the detection of atomicity violations is how to identify atomic regions.

On the one hand, some static approaches rely on user annotations and type systems to specify and then verify atomic code regions (i.e., functions) [FQ03]. On the other hand, dynamic methods such as `AVIO` [LTQZ06] propose *access interleaving invariants* and build a system that extracts these invariants to detect violations at runtime, whereas `Atomizer` [FF04] leverages *lock-set* and reduction algorithms to verify that the execution of atomic blocks is not affected and does not interfere with other threads. Other approaches build upon the concept of *conflict serializability* of execution traces, soundly identifying traces that cannot be transformed into equivalent serial traces and detecting violations dynamically [FFY08, MV20].

From a testing perspective, frameworks such as `CTrigger` [PLZ09] and `AssetFuzzer` [LCC10] focus on the directed exploration of specific thread interleavings inherently correlated to atomicity violations. They first perform analyses to identify candidate interleavings to then direct the execution towards them.

3.1.2.3 **Order Violations**

In concurrency bug detection research, order violations have received the least attention, although they are widely found among concurrency errors [LPSZ08]. Preventing and fixing them is particularly difficult, since even if two shared memory accesses are protected by the same lock or two conflicting code regions are atomic to each other, the execution order between them still may not be guaranteed.

Similar to atomicity violations, it is common among developers to assume an execution order but enforcing that order is notably challenging. Further, to detect order violations, one needs to be able to identify whether the executed instructions have been executed in the right order or not. Consequently, this raises the challenge of determining the expected orders between pairs of events.

Among the proposed approaches, not many target order violations in a specific fashion. Instead, there are general detectors that are not focused on a single class of concurrency bugs but target multiple classes in a more general fashion. For instance, `Falcon` [PVH10] uses a pattern-based dynamic analysis method that identifies conflicting interleaving patterns that access shared memory to then statistically rank them based on a suspiciousness measure. `ConMem` [ZSL10] targets concurrency errors that lead to program crashes, building on bug effects instead of on specific interleaving patterns, unlike most approaches. To this end, it predicatively detects errors by identifying potentially harmful memory operations and then checking timing conditions among them. In a similar vein, `ConSeq` [ZLO+11] proposes a backwards analysis approach that covers more failure patterns besides program crashes. Recently, other frameworks that target order violations specific to the Android platform have also been developed [FLJ18, WLS+19].

3.1.2.4 **Bugs in Specific Domains**

Some approaches are specifically tailored to particular problems and domains such as operating system kernels, embedded systems, file systems, and so on. Therefore, they do not target general-purpose concurrent programs and are designed to overcome unique challenges.

`PASAN` [KKR+21] is one of such approaches and seeks to protect peripherals in embedded systems from concurrency issues that corrupt on-going jobs and result in erroneous sensor values, for instance. To this end, it (i) parses device documents to identify MMIO address ranges for peripherals, (ii) extracts concurrency-related functions from firmware compiled

into LLVM bitcode, (iii) computes points-to information, identifies concurrently executable instructions by tracking process and thread life spans, and performs a context-sensitive *lock-set* analysis of the call stacks. Then, it (iv) identifies transaction spans (that should ideally be locked), and (v) verifies that the identified transaction spans that access the same MMIO address range are correctly locked.

Other approaches are focused on detecting data races in operating system kernels (i.e., Linux). `Razzer` [JKS+19] uses a combination of static analysis and fuzzing. It first identifies potential racy memory access instructions through context-insensitive and flow-insensitive points-to analysis, over-approximating race candidate pairs. Then, it injects breakpoints at the identified points, and performs fuzz testing seeking to execute the race candidates while controlling thread interleavings. `Krace` [XKZK20] is a fuzzing framework that targets kernel file systems. Apart from the common branch coverage metric, it uses a concurrency-coverage metric called alias instruction pair, which describes thread interleavings by tracking memory access instruction pairs from different contexts to each memory address. It also injects random delays at memory accesses to discover thread interleavings. To check for data races, it implements offline *happens-before* and *lock-set* algorithms.

In a different direction, researchers found that jointly exploring interleavings and test inputs in kernels has received little attention due to the big search space [GAFM21]. In that context, `Snowboard` [GAFM21] is a concurrency testing framework that first executes a corpus of sequential tests generated by `Syzkaller` [Goo], and collects non-stack memory accesses for the relevant thread using a customized hypervisor. It then identifies and clusters pairs of tests that read and write the same memory region respectively. Next, it concurrently executes test pairs, segregating threads in separate vCPUs, and executing one vCPU at a time to enforce different interleavings. To find bugs, it relies on existing bug detectors, and ranks them by frequency, manually inspecting the highest-ranked ones.

3.1.3 From Errors to Vulnerabilities

Some of the approaches described above are also used to detect security vulnerabilities stemming from the aforementioned concurrency errors, usually through manual inspection [JBLH22, JKS+19]. In this way, they have been able to identify denial of service opportunities, privilege escalation, and memory corruption instances.

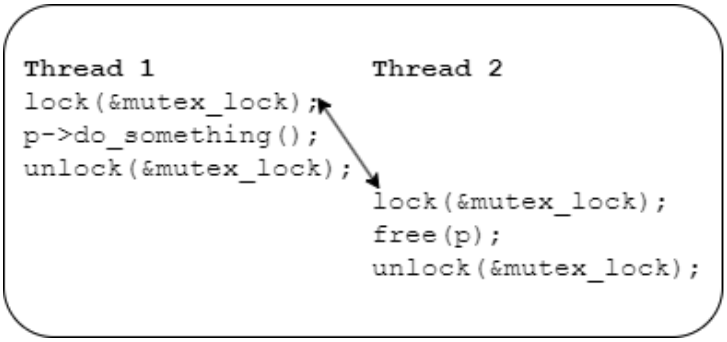


Figure 3.1: Example race-free code snippet with a potential concurrency use-after-free vulnerability.

However, these tools are geared towards detecting certain concurrency bugs, and even discerning benign races from harmful ones is often a costly manual task. To illustrate, researchers spent approximately 80 hours on manual inspection and reproduction of 100 data races [GAFM21]. To find and verify security consequences, it requires further manual inspection and extra effort — a time consuming process that requires meticulous code analysis.

3.2 Concurrency Vulnerabilities

In previous works [YCSS12, ZGQ+18], researchers analyzed and discussed how concurrency errors can be exploited to carry out several attacks. However, further research showed that even though vulnerabilities and common concurrency errors have similar characteristics, their surrounding conditions are different and neither is a necessary requirement for the other to take place [LZL+18, CZM+19]. For instance, concurrency vulnerabilities can exist in race-free executions [cvea], and triggering a concurrency bug and a vulnerability usually need different inputs and thread interleavings. Figure 3.1 illustrates this scenario with a simple code snippet example with a potential concurrency use-after-free in a race-free program slice. The two thread accesses to the shared variable *p* are correctly protected by the same lock and *thread 1* is expected to be executed before *thread 2*. However, if the expected execution order gets reversed and *thread 2* executes its statements before *thread 1* does, a concurrency use-after-free takes place.

Therefore, existing bug detection approaches may fail to detect vulnerabilities on their own. In the event that a vulnerability would arise out of an

error, they would be able to detect the corresponding error but not whether there could be a vulnerability or its triggering conditions, since they do not reason about harmful program behavior from a security perspective. Similarly, concurrency vulnerability detection approaches that build upon bug detectors may miss potential vulnerabilities.

Among the vulnerabilities that have their origin in concurrency, the most well-known include Time-Of-Check-to-Time-Of-Use (TOCTOU) vulnerabilities that target the file system [CGJ09], and double-fetch vulnerabilities [WKL+17]. However, concurrency can also lead to other vulnerabilities that for instance enable side-channel attacks [WW19]. In this research, we focus on memory corruption exclusive to concurrent executions.

3.2.1 Real-World Vulnerability Examples

In order to provide a practical illustration of real-world concurrency memory corruption vulnerabilities, we have manually selected two representative examples from the CVE database. Figure 3.2 shows a simplified version of the code in `drivers/block/nbd.c` in the Linux kernel, that is affected by CVE-2021-3348 [cvec], a concurrency use-after-free vulnerability.

When a Network Block Device (NBD) is being set up, the Linux kernel calls the function *nbd_add_socket*, which in turn calls *krealloc* to reallocate memory for the *config->socks* array to add new sockets to the configuration. Usually, *krealloc* will decide to expand or shrink the original memory block in place if possible, or to allocate a new memory block, copy the data, and free the original block. At that point, if an I/O request is received, *nbd_handle_cmd* will get called, dereferencing *config->socks* without any locking. Since this happens after *config->socks* is reallocated but before it gets assigned the new memory address, a use-after-free will take place.

To fix this vulnerability, locking accesses to *config->socks* was not a feasible solution, as it would introduce excessive overhead. Instead, kernel developers opted to freeze the request queue to not receive requests when adding sockets to the configuration.

In a similar vein, Figure 3.3 illustrates a shortened version of the relevant code with regards to CVE-2019-11366 [cveb] in atftp [Lef], a multi-threaded implementation of the Trivial File Transfer Protocol (TFTP).

In `tftpd_list.c`, atftp uses a global linked list of server threads called *thread_data*. These threads are started by the main thread when new requests arrive. To do so, the *main* function calls *pthread_create* to execute

```
static int nbd_handle_cmd(struct nbd_cmd *cmd, int index)
{
    struct request *req = blk_mq_rq_from_pdu(cmd);
    struct nbd_device *nbd = cmd->nbd;
    struct nbd_config *config;
    struct nbd_sock *nsock;

                                        ...
    config = nbd->config;

                                        ...
    nsock = config->socks[index];
    mutex_lock(&nsock->tx_lock);

                                        ...
}

static int nbd_add_socket(struct nbd_device *nbd, unsigned long arg,
                          bool netlink)
{
    struct nbd_config *config = nbd->config;
    struct socket *sock;
    struct nbd_sock **socks;

                                        ...
    socks = krealloc(config->socks,
        (config->num_connections + 1) * sizeof(struct nbd_sock *),
        GFP_KERNEL);
    if (!socks) {
        kfree(nsock);
        err = -ENOMEM;
        goto put_socket;
    }
    config->socks = socks;

                                        ...
}
```

Figure 3.2: CVE-2021-3348 in the Linux kernel.

tftpd_receive_request, which later calls *tftpd_list_add*. This is shown in the bottom lines of the code. Inside *tftpd_list_add*, atftp initializes a local pointer to the head of the thread list *thread_data*. Then, it checks whether the thread list is empty by checking whether *thread_data* points to NULL. In that case, it inserts the first element to the list. Otherwise, it walks the list using the local pointer *current* and inserts the element at the end of the list.

Since the assignment to *current* is not lock-protected, the harmful behavior occurs when *thread_data* is NULL when assigned to *current* in a thread, but modified by another thread before it is checked. In such a case, the program will try to walk the list and dereference *current->next*, resulting in a null-pointer dereference.

```
/* In tftpd_list.c */
struct thread_data *thread_data = NULL;

                                        ...
int tftpd_list_add(struct thread_data *new)
{
    struct thread_data *current = thread_data;
    int ret;

    pthread_mutex_lock(&thread_list_mutex);

                                        ...
    if (thread_data == NULL)
    {
         thread_data = new;
         new->prev = NULL;
         new->next = NULL;
    }
    else
    {
         while (current->next != NULL)
              current = current->next;
         current->next = new;
         new->prev = current;
         new->next = NULL;
    }
    pthread_mutex_unlock(&thread_list_mutex);
    return ret;
}

/* In tftpd.c */
void *tftpd_receive_request(void *arg)
{
    struct thread_data *data = (struct thread_data *)arg;

                                        ...
    if (!abort)
      stats_new_thread(tftpd_list_add(data));

                                        ...
}
int main(int argc, char **argv)
{

                                        ...
    if (pthread_create(&tid, NULL, tftpd_receive_request, (void *)new) != 0)

                                        ...
}
```

Figure 3.3: CVE-2019-11366 in atftp.

3.2.2 The Concurrency Vulnerability Landscape

To observe and analyze the characteristics of concurrency vulnerabilities, their long-term presence in real-world programs, and the possible trends in concurrency vulnerabilities (and concurrency memory corruption vulnerabilities) over the years, we used the series of scripts that automatically

Table 3.1: Common Weakness Enumeration (CWE) identifiers for concurrency vulnerabilities in the CVE database.

ID	Name
CWE-362	Race Condition
CWE-364	Signal Handler Race Condition
CWE-366	Race Condition within a Thread
CWE-367	Time-of-check Time-of-use (TOCTOU) Race Condition
CWE-421	Race Condition During Access to Alternate Channel
CWE-567	Unsynchronized Access to Shared Data in a Multithreaded Context
CWE-662	Improper Synchronization
CWE-663	Use of a Non-Reentrant Function in a Concurrent Context
CWE-667	Improper Locking
CWE-820	Missing Synchronization
CWE-821	Incorrect Synchronization

download, parse and classify all CVE entries from the National Vulnerability Database (NVD) data feeds [nvd] described in Section 2.4 in Chapter 2.

3.2.2.1 Categorization

Similar to the approach adopted in Section 2.4, we classify vulnerabilities and exploits by examining their assigned Common Weakness Enumeration (CWE) identifiers combined with custom heuristics that leverage pattern matching using regular expressions. Table 3.1 lists all the CWE identifiers that we manually identified along with their corresponding names. Note that all CWE identifiers deal with general concurrency issues and not specific cases such as concurrency memory corruption.

Unfortunately, it is again nearly impossible to obtain fully accurate results regarding concurrency vulnerabilities for several reasons. Besides many entries not having a CWE assigned, in the case of concurrency vulnerabilities their assigned identifiers usually do not fall into any of the categories of concurrency issues but into the categories of their consequent weaknesses (e.g., use-after-free), making it harder to identify them. In addition, searching for simple keywords such as “thread” or “concurrent” is not appropriate because they may wrongly match cases where they are used to describe software functionality or other unrelated execution aspects (e.g., multiple concurrent connections, a worker thread).

To reduce false positives as much as possible we go through some steps before classifying a CVE entry as a concurrency memory corruption vul-

Table 3.2: Number of reported CVEs for the selected categories and the total, along with their mean CVSS score.

Category	# Vulnerabilities	CVSS
Concurrency memory corruption	564	7.1±1.2
Concurrency vulnerabilities	1,922	6.7±1.4
Memory corruption	42,989	7.7±1.4
Web-related vulnerabilities	53,913	6.8±1.7
Total	203,652	-

nerability appart from looking at the CWE identifiers in Table 3.1. For instance, whenever a CVE entry has assigned more than one CWE identifier, we check whether it includes at least one identifier from the set in Table 3.1 and another one from the set in Table 2.1. Furthermore, we check several concurrency and thread-related regular expression matches in the CVE description. If any of these regular expressions matches, we then check whether the entry has a CWE identifier from the memory corruption set, or whether any of the memory-corruption-related regular expressions also match in the description.

3.2.2.2 Analysis of CVE Information

Using the method described above, we then collected and generated statistical information about the reported vulnerabilities ranging from 2002 to the end of 2022. Note that we are primarily interested in concurrency-related vulnerabilities and thus we only discuss other vulnerability categories as a reference in Table 3.2. There are many different categories and subclasses, as well as many possible classification criteria. In order to obtain a general understanding, we also show memory corruption since it is the sequential equivalent of the vulnerabilities we are interested in, and web-related vulnerabilities as a reference to another broad category. Within concurrency vulnerabilities, we further identify those that result in memory corruption.

Table 3.2 shows the number of reported vulnerabilities in the NVD database for each of the categories that we selected along with the mean and standard deviation of their Common Vulnerability Scoring System (CVSS) scores.

1,922 out of the 203,652 total reported vulnerabilities are concurrency

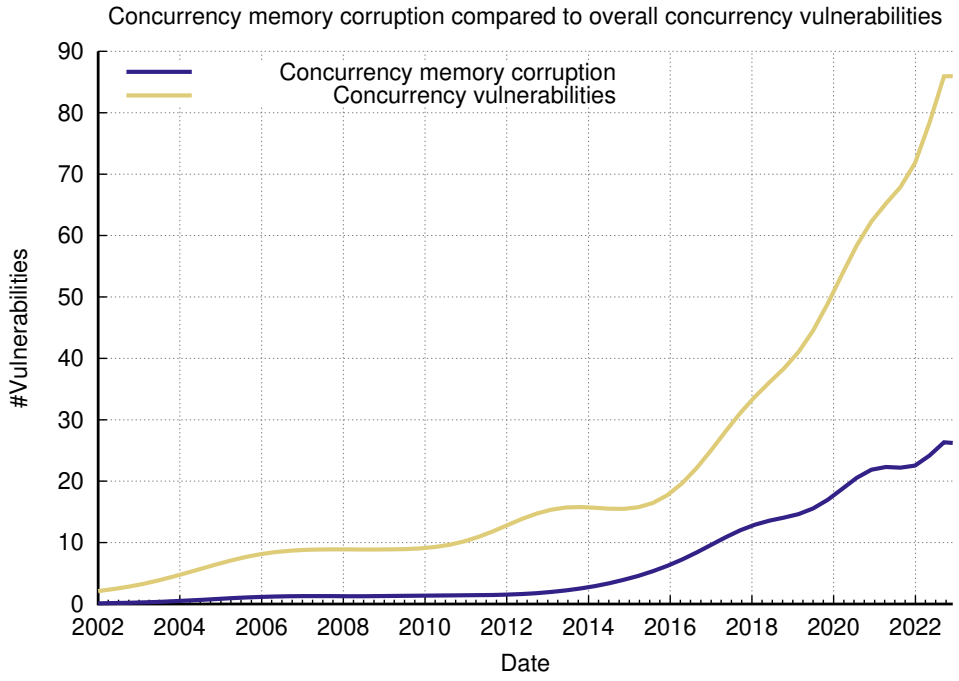


Figure 3.4: Concurrency vulnerabilities and concurrency memory corruption growth over the years.

vulnerabilities, of which 564 are concurrency memory corruption vulnerabilities. In contrast, other well-known and well-studied vulnerability categories such as memory corruption and web-related vulnerabilities account for 42,989 and 53,913 vulnerabilities respectively. Regarding CVSS mean scores, the four selected categories have similar values, resulting in a high severity rating in most of the cases. As expected, broad categories that have more diversity in the nature of the vulnerabilities show a higher standard deviation (i.e., web vulnerabilities). In addition, we found a total of 468 and 105 publicly available exploits for concurrency and concurrency memory corruption vulnerabilities respectively.

Figure 3.4 shows the total number of both reported concurrency vulnerabilities and the subclass of concurrency memory corruption vulnerabilities per 3 months from 2002 to end 2022. Not surprisingly, in the early years barely any vulnerabilities were reported in either of the two cases. Out of these, the vast majority were TOCTOU and other file system related race condition vulnerabilities. Afterwards, the number of reported concurrency vulnerabilities started to gradually increase until 2016, from where it experienced a substantial and continuous growth up to the present, reach-

Table 3.3: Number of concurrency and concurrency memory corruption vulnerabilities for each CVSS score ranked from 0 to 10. Conmem stands for concurrency memory corruption.

Severity	CVSS Score	# Concurrency	# Conmem
None	0	0	0
Low	0.1-1	0	0
	1-2	0	0
	2-3	12	0
	3-4	33	3
Medium	4-5	160	33
	5-6	212	53
	6-7	235	106
High	7-8	632	232
	8-9	187	66
Critical	9-10	34	21

ing its peak at nearly 90 vulnerabilities per 3 months. This growth can be attributed to both the rising development of increasingly complex concurrent software and the recently proposed detection approaches. Concurrency memory corruption vulnerabilities followed a similar trend. The number of reported vulnerabilities started to steadily grow from 2014 onwards. Although it initially looked like it may have stabilized as of 2018, it then continued to grow until it peaked in the present as well, at over 20 reported vulnerabilities per 3 months. It is worth noting that 76% of the total number of concurrency memory corruption vulnerabilities have been reported over the period from 2017 to 2022, following an upward trend.

Table 3.3 lists the number of concurrency and concurrency memory corruption vulnerabilities for each CVSS score range respectively. As explained in Section 2.4, vulnerabilities with a score of 0 are considered as non severe, a score from 0.1 to 3.9 is considered as low severity, a score from 4 to 6.9 is considered as medium severity, a score from 7 to 8.9 is considered as high severity, and a score from 9 to 10 is considered critical.

Figure 3.5 and Figure 3.6 depict the overall distribution of concurrency and concurrency memory corruption vulnerabilities respectively over the different severity levels based on their CVSS base score. We can observe

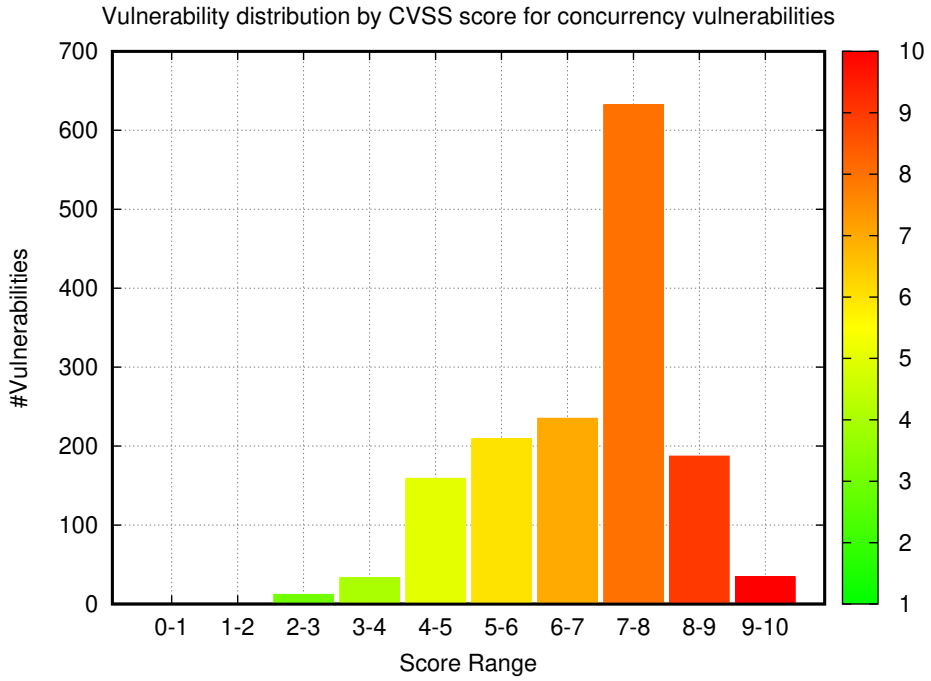


Figure 3.5: Distribution of concurrency vulnerabilities by CCSS score.

that both concurrency vulnerabilities and concurrency memory corruption vulnerabilities have their most reported vulnerabilities within the 7 to 8 range, meaning high severity vulnerabilities. Specifically, out of the total of 1,505 concurrency vulnerabilities, 853 have a score between 7 and 10, accounting for more than a half of the total. In the case of concurrency memory corruption, out of 514 vulnerabilities, 319 are within the 7 to 10 range.

3.3 Concurrency Exploitation

Some research works have been devoted to understand and improve the exploitation of concurrency vulnerabilities. OWL [ZGQ+18] was the first study that analyzed the problem of exploiting concurrency vulnerabilities stemming from data races. Their approach uses existing data race detectors and checks whether the corresponding memory region propagates to attack sites (e.g., *strcpy()* calls, NULL dereferences) through control-flow or data-flow via inter-procedural static analysis. Then, it tries to trigger the bug and execute an attack site.

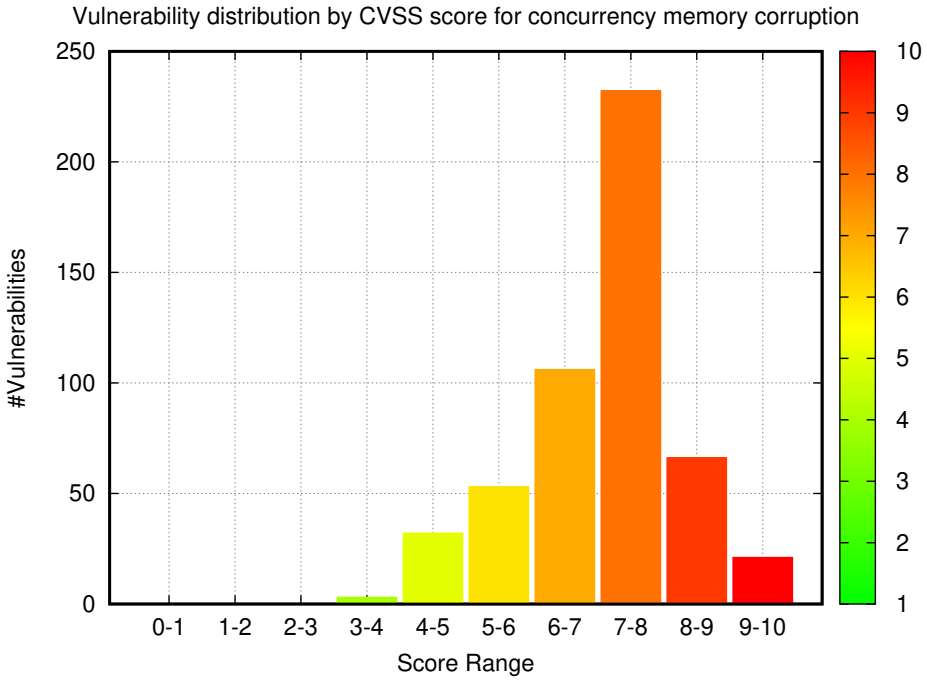


Figure 3.6: Distribution of concurrency memory corruption vulnerabilities by CCSS score.

They found that the duration of the vulnerable window (i.e., the timing window in which the error may materialize) is key to successfully exploit a concurrency error. While some concurrency errors have large vulnerable windows that enable their exploitation without much difficulty through triggering sequences of UI events or re-running commands a few times, those with shorter vulnerable windows are harder to exploit. Moreover, hardware cache leases and CPU time slices are often larger than these short timing windows, masking the errors.

In order to increase the probability of success of the attacks, they proposed two methods: (i) to retry multiple times until the exploit succeeds, and (ii) to provide carefully crafted inputs to enlarge the vulnerable window (e.g., by making a thread copy a large array, or triggering blocking operations such as disk I/O).

Afterwards, researchers in the security community verified that traditional race exploitation in practice relies on brute-force attacks until it succeeds [LML21]. Such attacks work for some kernel races and have been leveraged by most race-based privilege escalation attacks. However, brute-force is not effective at all in some cases. The exploitation complexity varies

for single-variable races and multi-variable races. They also found that some multi-variable races (named non-inclusive multi-variable races) are almost impossible to successfully exploit through brute-force, since they require a unique execution order that cannot be triggered without using debugging features.

Even though the probability of successful exploitation of single-variable races and inclusive multi-variable races (i.e., when the execution time between the two writes in one task is shorter than the time taken between the two reads in the other task) through brute-force appears to be low, exploitation is indeed possible and effective with many trials. In fact, publicly available brute-force exploits succeed in between 5 to 30 seconds. However, the probability of successful exploitation of non-inclusive multi-variable races (i.e., the execution time between the two writes in one task is greater than or equal to the time taken between the two reads in the other task) using brute-force is virtually zero due to timing constraints for the tasks to satisfy the required execution orders.

ExpRace [LML21] proposes a generic exploitation technique for non-inclusive multi-variable kernel data races, a type of data race that cannot be exploited through brute-force. The main objective is to intentionally enlarge the execution time of the target instruction sequence of the task that performs the read accesses using interrupts, so that a non-inclusive multi-variable race turns into an inclusive multi-variable race. Since the interrupt mechanisms are not accessible from user-space, ExpRace indirectly raises interrupts using different methods to enlarge the time window that enables exploitation. In particular, it focuses on inter-processor interrupts and hardware interrupts through system calls and interrupt requests to devices. They found that hardware interrupts yield the better results in terms of exploitation success and stability.

In a different direction, COMRace [GGL+22] seeks to automatically synthesize proof-of-concept (PoC) exploits that concurrently execute unsafe method pairs with conflicting member field accesses in vulnerable Component Object Model (COM) interfaces. To do so, it finds interface methods that access member fields without synchronization through reverse engineering and static analysis. Then, given a pair of interface methods with conflicting accesses, it automatically synthesizes PoC exploits that concurrently invoke these methods in a loop, expecting to trigger the vulnerable interleaving.

Table 3.4: Predictive detection approaches for concurrency memory corruption vulnerabilities along with their main characteristics.

	UFO	**ConVul**	**ConVulPOE**
Technique	Maximal Causal Model + Constraint Solving	Exchangeable Events Tracking	Trace Construction via Partial Orders
Target vulnerability	Use-After-Free	Use-After-Free, NULL-Pointer Dereference, Double-Free	Use-After-Free, NULL-Pointer Dereference, Double-Free
Output	Conflicting Call Stacks	Conflicting Call Stacks	Vulnerability-Exposing Traces
Validation Method	-	Enforced Scheduling	-

3.4 Detection Approaches

In Chapter 2, we observed that memory errors are widespread in current software and pose a major threat from a security perspective since they enable several attacks [SPWS13]. In consequence, numerous detectors and mitigations have been proposed in over three decades of research. Unfortunately, concurrency introduces new ways in which memory corruption may take place, giving rise to the need for new detectors that can find vulnerabilities that traditional tools cannot identify.

In this section, we bring together the relevant proposed concurrency memory corruption detectors in the literature. As in previous sections, we only consider approaches that target C and C++ programs. Thus, we do not include frameworks such as `ExceptioNULL` [FMRS12], which formulates null-pointer dereference prediction as a constraint solving problem, because it is specific to Java programs. The individual approaches are described in their own sections below.

3.4.1 Predictive Detection Algorithms

Tools within this category seek to predict vulnerabilities that can take place from an observed correct execution. They are usually based on existing concurrency bug detection algorithms that have been adapted to vulnerability detection. Many of the existing frameworks rely on Dynamic Binary Instrumentation (DBI) techniques as their base technology.

Table 3.4 summarizes proposed predictive detection algorithms for concurrency memory corruption detection in the literature. It details the underlying detection method of each tool. We also specify the vulnerability subclasses targeted by the different tools. In addition, we point out the output scheme of the reported vulnerabilities since it is a relevant aspect in the reproduction and verification of findings. The validation method refers to whether the proposed approaches try to trigger the identified vulnerabilities to verify their execution. Although this is intrinsic to testing approaches such as fuzzing, tools based on predictive detection may also benefit from it to avoid false positives.

UFO

UFO [Hua18] formulates concurrency use-after-free detection as a constraint solving problem and builds upon a maximal causality model with the intention to predict the maximal set of vulnerabilities from an observed concurrent execution trace. In contrast, most popular algorithms like happens-before are not maximal (i.e., not all feasible traces are captured).

Since the original maximal causal model [HMR14] is not aware of the semantic properties that define a use-after-free, UFO first extends it by including allocation, use, and free events to then encode the model and use-after-free violations as first order logical constraints to be solved using a SMT solver.

To this end, UFO first modifies `ThreadSanitizer` to produce the execution trace that contains occurrences of all the previously defined events in the model. This data is encoded and written to disk for offline analysis for correctness and efficiency. Afterwards, UFO builds the model's constraints from the traces and finds allocation and free operation pairs along with the conflicting memory accesses. In order to detect use-after-frees, it then carries out a *happens-before analysis* for each pair of use and free events on overlapping memory ranges. If the HBR is not satisfied either way by inter-thread synchronization, then UFO builds the conjunction of the model and the use-after-free constraints and invokes Z3 [MB08] to solve it. If it

produces a solution, UFO reports the specific scheduling that triggers the use-after-free.

The prediction relies on constraint solvers to determine the feasibility of the different schedulings. Without an efficient solver, the solving process can be time-consuming. In practice, a tool usually adapts a relaxed model and, consequently, compromises its ability

ConVul

In ConVul [CZM+19], the authors highlight the differences between data races and concurrency memory corruption vulnerabilities, and the ineffectiveness of data race detectors in finding concurrency vulnerabilities.

They propose the concept of *exchangeable events* to refer to pairs of events whose orders can be reversed. In its strict version, when for a pair of events both execution orders are observed, they are considered to be exchangeable. However, that approach is not practical. In consequence, they define a relaxed model that heuristically predicts whether two events are exchangeable based on their *synchronization distance*. The *synchronization distance* is the minimal number of synchronization edges that order the two events, being synchronization edges sequences of either lock acquire and release operations in the same thread, or release and later acquire operations by different threads. Taking this definition into account, they determine that, for a pair of events, if their *synchronization distance* is 3 or less, there is a high probability to reverse their execution order. Further, if there is a third event such that the distances from both events to it are close to 0, then there will be an increased probability.

Given the above definition, they implement a detection system on top of Pin [LCM+05] Dynamic Binary Instrumentation (DBI) framework that instruments synchronizations and memory accesses to track exchangeable events. Whenever two conflicting operations, as defined by the different algorithms for each targeted concurrency memory corruption vulnerability (see Table 3.4), are found to be exchangeable, a potential vulnerability is reported.

To verify reported vulnerabilities, ConVul tries to trigger the harmful execution order by suspending the thread that should execute the last operation (e.g., the use in a use-after-free) right before its execution.

ConVulPOE

In a similar vein to ConVul, ConVulPOE [YWC+21] focuses on exchangeable event pairs because they are more likely to cause concurrency memory corruption. In this approach, two events are considered exchangeable if two valid traces exist in which the execution order of these events is different.

The main objective of ConVulPOE is to construct feasible traces that manifest vulnerabilities from an observed execution trace. To that end, it leverages partial orders that represent the execution orders of events. In the first place, it needs to identify vulnerability-potential event pairs from an observed trace, in accordance with the targeted vulnerabilities (see Table 3.4). Next, it assembles a feasible subset of events. These feasible event sets are the events that should be included in the new trace (i.e., the event pairs that can lead to a vulnerability if reordered) along with the other events that make the execution correct. To prove that a potential vulnerability may exist, it tries to generate a new feasible trace by building a partial order over the feasible event set to add ordering between events while meeting all necessary constraints. In other words, from a basic partial order, it then considers newly ordered events to add new orderings between two events incrementally.

ConVulPOE is composed of a trace recorder and a vulnerability predictor component. The trace recorder is implemented on top of Pin to instrument threading and dynamic memory allocation operations, as well as pointer dereferences heuristically. Taking the generated traces as input, the vulnerability predictor works as an offline analysis to find potential event pairs and construct partial orders incrementally. In large programs, the trace can be overly long, making the partial order generation process time-consuming. In these cases, it divides the trace into multiple parts and analyzes each of them.

3.4.2 Concurrency Testing

Testing methods specific to concurrency randomly or systematically explore different thread interleavings and program states. For instance, systematic state space exploration by model checking is an effective verification method. Since traditional model checkers are not practical for concurrent programs, stateless model checkers systematically explore traces of shared memory accesses without capturing all program states. A custom scheduler controls the non-determinism in the program and enumerates all execution paths.

Table 3.5: Proposed concurrency testing approaches towards memory corruption vulnerability detection along with their main characteristics.

	ConAFL	**MUZZ**	**AutoInter-fuzzing**	**PERIOD**	**GenMC**
Technique	Static Sensitive Operation Finding + Fuzzing	Thread-Aware Grey-Box Fuzzing	Static Operation Pair Finding + Fuzzing	Controlled Concurrency Testing	Stateless Model Checking
Target vulnerability	Buffer Overflow*, Use-After-Free*, Double-Free	General Memory Corruption	Use-After-Free, Double-Free, NULL-Pointer Dereference	Use-After-Free, Double-Free, NULL-Pointer Dereference	Use-After-Free, Double-Free, Uninitialized Memory Access
Output	Crashing Input and Interleaving	Crashing Input and Interleaving	Crashing Input and Interleaving	Buggy Schedule	Offending Execution Graph

* Although buffer overflow and use-after-free are specified as targeted vulnerability classes in ConAFL, it could not detect any in the experiments.

Furthermore, dynamic analyses through testing such as fuzzing and controlled concurrency testing aim at effectively triggering concurrency memory corruption vulnerabilities in actual executions. Their main objectives and challenges include the exploration of the thread interleaving space and the orchestration of thread scheduling.

Table 3.5 encapsulates concurrency testing approaches towards concurrency memory corruption detection. It specifies the underlying detection technique of each tool, and the vulnerability subclasses targeted by the different tools. In addition, we point out the output scheme of the reported vulnerabilities since it is a relevant aspect in the reproduction and verification of findings.

ConAFL

ConAFL [LZL+18] is an heuristic framework composed of a static analysis phase and thread-aware fuzzing using a priority-based strategy. In the first place, the static analysis locates sensitive concurrent operations on shared memory (e.g., calls to *memcpy()*) and categorizes them into a potential type of vulnerability from the targeted concurrency vulnerabilities (i.e., buffer overflow, use-after-free and double-free). Then, a custom fuzzer built upon AFL [Zal] adjusts thread priorities through instrumentation and executes the target program.

The static analysis is implemented upon the concurrency bug detector LOCKSMITH and can be divided into four steps: (i) discover shared memory by recording pointers that are either passed through *pthread_create* or point to a global variable in the different threads, (ii) mark sensitive operations by building a Data-Flow Graph (DFG) for the identified shared variables and labeling sensitive operations (e.g., calls to *free()*) on that DFG, (iii) find pairs of concurrent sensitive operations by merging data-flows that share a preceding node and verifying that the operation pairs are concurrent by checking the Control-Flow Graph (CFG), and (iv) categorize each sensitive operation pair into a potential vulnerability type (e.g., two calls to *free()* to a double-free).

To trigger potential vulnerabilities, the thread-aware fuzzer inserts an assembly code snippet around the identified operations from a sensitive operation pair. This code adjusts the priority of the thread so that the two threads would likely be executed in the order that would trigger the vulnerability.

MUZZ

General-purpose grey-box fuzzers are tied to single-threaded testing, and thus they are not aware of multithreading-specific contexts. To overcome these limitations on fuzzing multithreaded programs, MUZZ [CGX+20] proposes three new thread-aware instrumentation algorithms on top of AFL that provide useful runtime feedback to optimize seed generation and execution strategies to explore execution states originated from different thread interleavings.

Prior to instrumentation, MUZZ builds a thread-aware inter-procedural CFG (ICFG) that encodes threading operations information such as thread creation and exit, lock acquires and releases, and shared variables. Given that ICFG, it identifies code sequences relevant to multithreading that may interleave during execution (i.e., suspicious interleaving scope) to later focus instrumentation on them. The objective is to emphasize multithreading-relevant paths during fuzzing and to reduce the amount of instrumentation required.

- The coverage-oriented instrumentation tracks thread-interleaving induced coverage. It does so by extending a common coverage approach [Zal] to instrument basic blocks that include suspicious interleaving scope as well as specific instructions inside with certain probability, and to instrument fewer blocks that do not.

- Threading context instrumentation collects the thread context (i.e., a deputy instruction and thread ID tuple) at calls to lock, unlock and join. When execution ends, MUZZ computes a context signature via hashing that defines the overall thread context.

- Schedule-intervention instrumentation seeks to diversify thread interleavings by assigning a random priority value to threads on thread creation. It also stores a thread ID for the calling thread to maintain a structure.

During the fuzzing process, MUZZ prioritizes seeds that exercise new coverage as done in AFL, but giving precedence to seeds that cover new thread contexts by checking the context signatures. In addition, it also adjusts the number of repeated executions for seeds that produce different context signatures in order to discover new thread interleavings.

AutoInter-fuzzing

Similar to other approaches, `AutoInter-fuzzing` [KZK22] combines static analysis and fuzz testing. It leverages static points-to analysis on top of LLVM to extract operation pairs that access the same memory region and are executed in different threads, in a similar way to `Razzer`. These operations include load and store operations, external function calls using local pointers, and memory allocation and deallocation functions. To filter out pairs that can never execute concurrently, it builds the control-flow graph to check reachability for the identified pairs afterwards. To control the scheduling, it instruments the detected operations with synchronization barriers to check and enforce an execution order for the given operation pair.

During fuzzing, `AutoInter-fuzzing` uses the number of created threads, the number of operations executed per thread, and the number of covered operation pairs as a metric to determine the number of test cases to be created from a given seed. Whenever it encounters any operation pairs during regular fuzzing, it executes the program again and forces the execution of the opposite interleaving if it has not been tested yet. In this way and by leveraging `AddressSanitizer` [SBPV12], it is able to detect use-after-free, NULL-pointer dereference, and double-free vulnerabilities.

PERIOD

`PERIOD` [WHW+22] is one of the most recent controlled concurrency testing (CCT) approaches and is built around three components: (i) a schedule generator, (ii) a periodical executor, and (iii) a feedback analyzer.

The schedule generator systematically models program execution as a series of execution periods, forming a schedule. An execution period represents the number of key points (i.e., relevant instructions) executed by a given thread in that period. Concatenating periods results in a schedule and depicts how different threads are interleaved. Given a slice of key points and a maximum number of execution periods (starting from 2), it generates schedule patterns that order periods in lexicographical order of thread identifiers. Then, it systematically generates all schedules in order by assigning key points to the corresponding periods for each pattern.

The periodical executor takes the schedules and controls thread interleavings. To do so, it leverages Deadline Task Scheduling [The], a scheduling policy that assigns deadlines to tasks and picks the task with the earliest scheduling deadline to be executed next. `PERIOD` sets all threads to

the same period length and start time, and instruments key points to halt execution of threads until the next period when necessary. In addition, it gathers information about any errors that may be triggered during execution with the help of `AddressSanitizer`, and the activated slice of key points, so that it is passed to the feedback analyzer.

During execution, the executed slice may be different to the one that generated the schedule (i.e., new key points are executed) because different interleavings may result in the program taking different paths. If a slice obtained that way is previously uncovered, the feedback analyzer creates a new schedule job for it. To this end, it creates a schedule prefix, a partial schedule that contains everything from the previous schedule up to the first different key point. Finally, it adds a pattern period after it for the corresponding thread.

`PERIOD` explores all possible schedules for the given number of periods. Then, it increments the period number to explore new schedules. This process is repeated until it reaches a preset bound.

GenMC

Verification of concurrent programs is challenging because of the huge number of interleavings of the threads comprising a concurrent program, that leads to a massive number of program states. `GenMC` [KV21] is a state-of-the-art stateless model checker for C and C++ concurrent programs built upon the LLVM toolchain. It is able to verify concurrent programs according to different memory models.

Its execution exploration algorithm leverages execution graphs where nodes represent memory access events, and edges relations among them, such as the order of the events in a thread, and write-to-read relations. From an empty graph, an interpreter executes the program and adds the next event found to the graph, while checking whether the graph is consistent with respect to the memory model. In the process of generating a full graph (i.e., a complete execution), it also finds and stores alternative exploration options such as, given a read event, the different writes to the same location from which it can read. When the found event is a write, it also considers other read events in the graph that could read from it. In this way, `GenMC` generates and explores every possible program execution.

Moreover, whenever a new event is added to the current graph, it also checks whether the event entails an assertion violation or an error (i.e., from the error classes that `GenMC` is able to detect). Thus, it detects con-

current use-after-free, double-free, and accesses to uninitialized memory.

3.4.3 Static Analysis

Pure static analysis of concurrent programs such as standard data-flow analysis poses additional challenges due to the numerous possible thread interleavings, intensifying the usual trade-off between precision and performance. However, researchers actively devise new methods to effectively find vulnerabilities through static analysis.

Canary

Canary [CYZ21] is a value-flow analysis framework built upon LLVM that tracks how values are stored and loaded via data and interference dependence relationships, making it possible to check diverse multi-threaded software safety properties.

In the first place, it constructs a value-flow graph (VFG) for the given program. This process is driven by an intra-thread data dependence analysis and an inter-thread interference dependence analysis. The data dependence analysis seeks to resolve the data dependence relations to represent variables and the statements that define or use them as nodes, along with the execution constraints as edges to represent value-flow relations. Then, the interference dependence analysis adds new dependence edges to the graph on pairs of load and store nodes from different threads to shared memory objects. These edges are also annotated with the constraints for the corresponding value flow.

Then, Canary checks for errors by verifying the reachability of source-sink paths in the VFG, where the source is a free statement and the sink any use statement. Specifically, it checks for value-flows that connect source to sink across different threads. To validate whether a given value-flow path conforms a feasible interleaving execution, the formula that encodes the aggregated constraints is passed on to the Z3 SMT solver. In this way, Canary is able to detect concurrency use-after-free vulnerabilities.

DCUAF

DCUAF [BLCH19] aims to detect use-after-free errors in Linux device drivers. To identify driver function pairs that may be concurrently executed, it analyzes the lock usage of each driver individually as local information, and

combines the information from all drivers to perform a global statistical analysis.

During the first phase, it gathers lock-related calls in each driver source file and performs an alias analysis to check whether two different calls have an aliased lock variable. In these cases, the callers are considered as possible concurrent functions. Then, it checks the ancestors of each candidate function pair in the call graph and filters out those that have a common ancestor to avoid potential false positives. Finally, it gets the set of driver interfaces that call each function, and computes the Cartesian product of the sets of driver interfaces, discarding pairs where both driver interfaces are the same.

In the second phase, DCUAF takes each concurrent interface pair from the previous phase and calculates the percentage of source files that contain these driver interfaces that have the selected concurrent interface pair. If that percentage is greater than or equal to a given threshold, the given interface pair is considered global concurrent. Then, given a pair of global concurrent interfaces in a driver, the functions linked to these interfaces are annotated as a concurrent function pair for that driver.

To find use-after-free bugs, DCUAF uses a summary-based *lock-set* analysis for efficiency. For each concurrent function pair, it collects the lock set of each variable access for both functions and leverages function summaries to handle called driver functions (i.e., function name, source file, accessed variables, their lock sets, code path to each access, and location). Then, it analyzes pairs of variable accesses of function pairs to verify whether the variable is the same, the lock set intersection is empty, and one of the accesses is a call to a memory freeing function. In these cases, a concurrency use-after-free is reported.

3.4.4 Other Approaches

There are other approaches also found in the literature that are related to the problem of concurrency memory corruption detection. The main reasons for not including them in the above systematization are because their core objective is different, they target non C/C++ programs, they are too specific, or reimplement and integrate existing approaches.

For instance, EBF [AMM+22] combines standard Bounded Model Checking (BMC) and fuzzing techniques. It implements an open-source fuzzer that implements standard techniques: random delay injection at instruction-level to explore thread interleavings and branch coverage to guide the mu-

tation process. To find potential bugs and vulnerabilities, it uses common sanitizers such as `AddressSanitizer` and `ThreadSanitizer`. `EBF` separately performs BMC and fuzzing and aggregates the results. Also, whenever the BMC reports a failed verification outcome, it initializes the fuzzer seeds with the produced counterexample.

Another example is `COMRace` [GGL+22], which explores the Windows registry to find registered Component Object Model (COM) objects, and uses common reverse engineering methods to decompile and reconstruct virtual function tables from binary files. Then, it statically analyzes each interface method implementation to detect unsafe methods that access member fields without synchronization.

3.5 Discussion

Concurrency not only introduces new potential security vulnerabilities but also enables traditional ones to be triggered in different ways. In previous sections, we have explored concurrency issues from bugs to vulnerabilities, with a strong focus on concurrency memory corruption vulnerabilities. Having analyzed their characteristics, their presence in real-world programs and main detection approaches, in this section we discuss the most relevant points and observations.

3.5.1 Reported Concurrency Vulnerabilities

Evidence from the CVE database shows that there is a low number of reported concurrency memory corruption vulnerabilities and concurrency vulnerabilities in general when compared to other more established categories (e.g., web, memory corruption). Comparing to these categories is not necessarily fair since they cover a much larger variety of vulnerabilities, are not specific to concurrent programs, and have been the focus of research and bug hunting for many years. However, the reduced amount of reported concurrency vulnerabilities can be attributed to: (i) they have received little attention in both industry and academia, (ii) concurrency vulnerabilities are harder to find and exploit since they require not only specific inputs to the program, but specific thread interleavings to be triggered, and (iii) unlike sequential detection tools, there are substantially fewer and less mature detection systems designed towards concurrency vulnerabilities. However, there has been a significant growth in reported concurrency

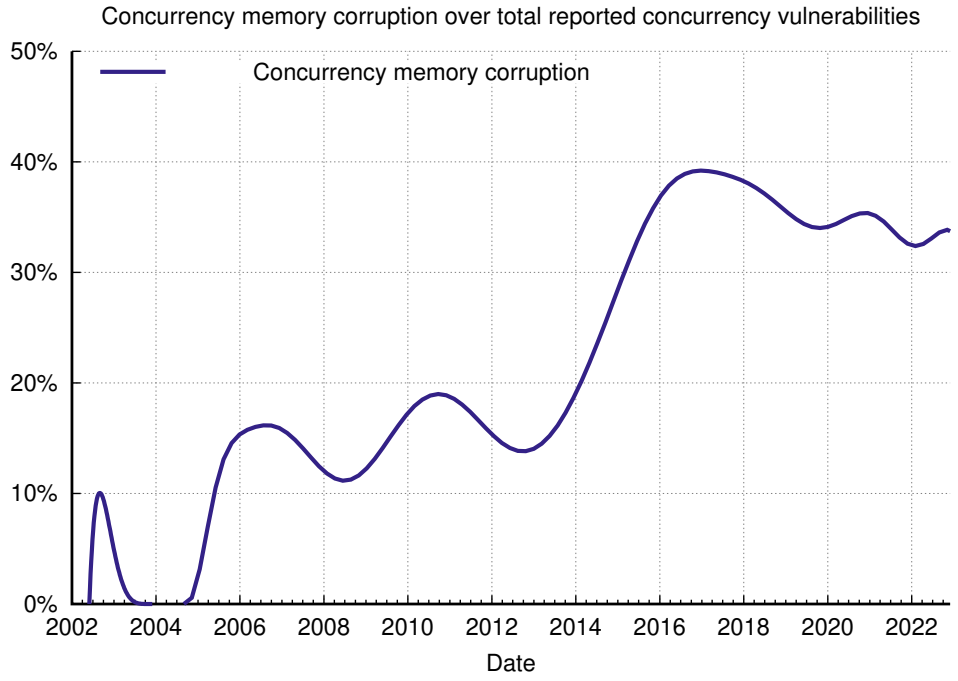


Figure 3.7: Concurrency memory corruption vulnerabilities as a percentage of total reported concurrency vulnerabilities.

vulnerabilities in the last few years, in line with the development of increasingly complex concurrent programs, the recent security research, and the newly proposed detection approaches.

Most of the concurrency memory corruption vulnerabilities have been reported in recent years. Specifically, 76% of them are found from 2017 onwards and approximately half off the total from 2019 to 2022, which indicates that they are an emerging threat that is also gaining attention, reflecting the number of proposed approaches in this period. Figure 3.7 outlines the relative number of concurrency memory corruption vulnerabilities in relation to the total number of reported concurrency vulnerabilities. As of 2022, approximately one third of the aggregated concurrency vulnerabilities actually lead to memory corruption. To illustrate, there are 85 out of 216 in 2021 and 118 out of 350 in 2022. Not surprisingly, we can observe that memory errors are even prevalent and still a major source of vulnerabilities even in multithreaded programs.

It is worth to note that the number of reported vulnerabilities could vary to some extent due to the conservative approach we used to reduce false positives to identify them.

With regard to CVSS scores, concurrency memory corruption vulnerabilities show rather similar values to those of traditional memory corruption, entailing a high severity threat in most cases. However, it is noteworthy that they are slightly lower in overall. This can be caused by the fact that they are harder to exploit, increasing the attack complexity and often requiring additional conditions to be met, making it necessary to repeatedly execute the exploit until it succeeds [YCSS12, ZGQ+18], for example.

3.5.2 Vulnerability Detection

With regards to concurrency memory corruption detection, it is remarkable that virtually all approaches have been proposed in recent years. Given that this topic has been gaining attention lately, there are not many research works yet.

Proposed solutions use static, predictive, or testing algorithms, and are mostly focused on detecting temporal memory errors (e.g., use-after-free, double free) because they are considered to be directly caused by event orders. Therefore, their characteristics are inherently compatible with concurrency and are likely to be triggered in concurrent executions. On the contrary, spatial memory errors (e.g., buffer overflow) pose a greater challenge since there is a wider variety of operations involved and their triggering conditions are usually complex. In fact, MUZZ is the only approach capable of detecting new concurrency spatial memory errors, although it has difficulties to find temporal memory corruption.

In the field of concurrency issues, bug and vulnerability reproduction is also an important and challenging task due to non-determinism. Being able to consistently reproduce a vulnerability to some extent reduces false positives and greatly helps developers to understand and fix it properly. Approaches built upon fuzzing techniques already provide the program input that crashed the application in the generated report. However, verifying vulnerabilities reported by tools based on predictive algorithms poses a major challenge. In that context, ConVul is the only predictive approach that tries to actually trigger vulnerabilities by manipulating thread scheduling to execute the harmful execution order.

There is no single approach that performs absolutely better than the rest, but each one comes with its own strengths and limitations. For instance, ConAFL is limited by its thread-aware static analysis because it does not scale to large programs (i.e., with more than 10K lines of code). Similarly, its fuzzer component does not explore new paths or thread interleav-

ings, it tries to enforce the vulnerable order for a given pair of sensitive operations, limiting the findings to the results of the static analysis.

UFO relies on a constraint solver to determine the feasibility of different schedulings. Since it usually generates long traces with a high number of constraints, it has to trade detection capability for efficiency in order to solve the constraints in reasonable time. Similar to other predictive approaches, it may miss vulnerabilities that would materialize in a different execution trace given a different program input.

ConVul leverages the *happens-before relation* to identify exchangeable events. It performs practical prediction and practical validation methods, which may result in missing potential vulnerabilities. Further, it uses a heuristic approach to predict exchangeable events that limits the *synchronization distance* to 3, missing potential exchangeable events with longer distances, and thus missing potential vulnerabilities.

ConVulPOE constructs feasible traces that expose vulnerabilities from partial order graphs. However, it cannot construct a trace when there are unordered conflicting event pairs that cannot be resolved, missing potential vulnerabilities. Moreover, if it is not able to identify pairs of events in the trace according to its definitions of vulnerability-potential event pairs, it does not detect vulnerabilities in that execution. Further, if the target program crashes, it does not detect any vulnerabilities even if it was caused by a vulnerability.

Prior to fuzzing, MUZZ performs static analysis to extract a suspicious interleaving scope to later focus coverage-oriented instrumentation on it. However, it does not include lock-protected operations, which have been proven to lead to concurrency vulnerabilities [CZM+19, YWC+21]. Therefore, it may intensely exclude statements that involve interesting interleavings. Moreover, controlled concurrency testing approaches such as PERIOD do not deal with program inputs, i.e., do not generate or try to discover new inputs but rely on a predefined set. Hence, they potentially benefit from other methods that seek to find new test cases, such as fuzzing.

In general, many approaches rely on points-to static analysis, which is associated to high false positive rates as precise and concrete control-flow and data-flow information can only be known at runtime. In addition, it also requires precise concurrency information, increasing the complexity of the analyses.

To sum up, temporal memory errors have been the focus of most existing concurrency vulnerability detectors due to their close alignment with concurrency, whereas spatial errors seem to remain as a greater challenge.

Among the proposed tools, those based on predictive analysis showed a better performance in detecting temporal errors, whereas fuzzing performed better for spatial memory corruption. There is room for further exploration in both cases to develop new approaches to effectively detect the already addressed error classes, or to target those that have received less attention.

3.6 Related Work

There are no existing works that explicitly focus on measuring and analyzing concurrency vulnerabilities and concurrency memory errors, and how they are detected. However, other measurement papers are closely related to individual parts of this research.

Concurrency bugs have been a popular topic of research for several years [FF09, MKV18, PLZ09]. Researchers have previously studied concurrency bug patterns, their manifestation conditions, and fix strategies [LPSZ08]. From the analyzed errors, they found that around one third were caused by different execution orders from those intended by the developers, and that most concurrency bugs can be reliably triggered, among other findings.

Given that memory corruption is still one of the major concerns in the security community, researchers have thoroughly analyzed how it is used to launch several attacks as well as the main mitigation and protection strategies [SPWS13], also from a binary-only perspective [SWS+16]. Part of these approaches are also applicable or could be adapted to the concurrent domain.

Fuzzing has been shown to be very effective in finding vulnerabilities even in multithreaded programs, and thus several different algorithms that improve or specialize this process have been proposed in recent years [Zal, CGX+20, XKZK20]. In consequence, researchers have analyzed the different techniques and designs implemented by fuzzers to build a taxonomy [MHH+19], and examined experimental methodologies in the literature to define a proper evaluation methodology to be carried out by fuzzers [KRC+18]. Concurrency vulnerability detection will also benefit from advances in fuzzing.

3.7 Summary

To take advantage of the increased processing capacity of modern processors, concurrent programs have become increasingly pervasive. Unfortunately, concurrency introduces new problems in the form of bugs and security vulnerabilities. Furthermore, it also enables traditional vulnerabilities such as memory corruption to be triggered in alternative ways specific to concurrent executions.

In this chapter we have reviewed the particular properties of concurrency bugs, how they are commonly classified according to their characteristics, and the main detection algorithms and tools proposed in the literature. In addition, we have pointed out how concurrency bugs and vulnerabilities are related but not required for each other to take place. We have also analyzed the presence and trends of reported concurrency and concurrency memory corruption vulnerabilities in the CVE database, and showed that they are an emerging threat. Lastly, we have systematically surveyed, classified, and discussed the most relevant detection approaches that target concurrency memory corruption vulnerabilities.

We hope that this research will draw attention towards concurrency vulnerabilities and the concurrent version of memory errors, and serve as a basis for future research to explore new detection methods and perspectives, as well as further security issues.

Chapter

4

Unveiling Dynamic Binary Instrumentation Techniques

Concurrency memory corruption vulnerabilities are one of the most hard to detect vulnerabilities in C and C++ programs. To get the necessary means and tools to detect concurrency memory corruption vulnerabilities, detectors usually rely on Dynamic Binary Instrumentation (DBI) frameworks. However, there are plenty of implementations and their designs and used techniques significantly vary from each other. Worse, there is no consensus on what exactly is considered DBI and what is not. This makes it difficult to know which techniques and solutions are adequate in which circumstances, and to understand the advantages and disadvantages of each of them.

In this chapter, we fill this gap with a comprehensive and systematic study of DBI. Specifically, this work dissects the building blocks of DBI, and identifies, classifies, analyzes, and evaluates the existing instrumentation techniques, pointing out the different characteristics and capabilities. We evaluate the performance of the techniques by testing a set of reference implementations over a complete comparative framework. Our analysis does not pretend to compare tools or implementations against each other, since their capabilities and performance are often influenced by design and

implementation decisions (e.g., some tools insert instrumentation routines written in languages such as Python or Javascript, which are significantly less efficient than C/C++). Instead, we focus on the underlying instrumentation techniques upon which existing DBI frameworks are built.

In particular, the contributions of this chapter are:

- The definition and drilldown of the DBI building blocks. Namely, instrumentation, introspection and execution manipulation.
- A comprehensive characterization and classification of known instrumentation techniques in relation to the fundamental instrumentation primitives they can provide. We further compare existing DBI frameworks that implement each technique, covering process-level and whole-system approaches.
- A thorough performance evaluation of instrumentation techniques for each instrumentation primitive, over a wide range of programs and conditions. Our findings show that the performance of the techniques often depend on different factors such as the nature of the program and the frequency of the triggered instrumentation events during execution, and that no single technique stands out from the rest in every possible test. We release all our raw results for use in future research.

The remainder of this chapter is organized as follows. Section 4.1 depicts the underlying principles of DBI, namely, instrumentation, introspection, and manipulation. Section 4.2 classifies and details the inner workings of the different instrumentation methods. Section 4.3 describes the existing tools that implement those techniques and the variety of run-time events they can instrument. Section 4.4 analyzes the transparency-related concerns, threats and approaches in DBI systems. Then, Section 4.5 evaluates the performance of the instrumentation techniques in different scenarios by means of their practical implementations. Section 4.6 discusses the main contributions of this work and reviews their impact, while Section 4.7 reviews the different works that are related to our research. Finally, Section 4.8 concludes the chapter.

4.1 DBI Building Blocks

We define DBI systems as those that take control of the execution of a target program, allowing to execute user-defined code on different execution

events and with different granularity, making it possible to examine the program state and alter the execution of the program in an automated way. While certain DBI approaches are focused on single processes running in a low privilege level (e.g., a single userland process), others undertake critical and complex processes such as the OS kernel or even entire emulated, virtualized, or physical systems.

Considering the proposed definition, we may argue that every DBI system has 3 fundamental building blocks that determine their capabilities: (i) one or more instrumentation techniques that enable execution control, including event generation and callback registration mechanisms, (ii) the introspection capability (i.e., the ability to observe the architectural state of the target), and (iii) the capability to alter the execution, either directly or by modifying the program state.

4.1.1 Process-Level & Whole-System Approach

Depending on the scope, DBI systems are classified into process-level and whole-system. Process-level DBI includes every approach whose scope is restricted to a single process usually in user-space, being only able to manage one program at a time. The DBI runtime can reside in the same address space as the target or in a remote location, so they can spawn or attach to a running process from the user-space [Bru04, LCM+05, NS07, BM11, Rav22, GDB22] or kernel-space [VY06]. On the contrary, whole-system solutions recreate entire systems and are composed of a host and a guest system. Therefore, instrumentation and analysis logic is generally implemented out of the analyzed target [YJZY12, LMP+14, DGHH+15, CKC11, ZLS+15].

Each approach comes with its own benefits and disadvantages. Whole-system tools provide a full-system view, allowing to observe kernel activity and process interaction. However, there is a clear increased performance cost compared to process-level approaches. Process-level DBI often share space with the instrumented program, while whole-system tools perform the instrumentation and analysis from outside the virtual machine, providing better transparency and stronger isolation. However, since a new layer is introduced between the host and the guest system, a significant challenge emerges regarding introspection — the semantic gap problem [JBZ+14].

4.1.2 Instrumentation

The backbone of DBI is formed by the different instrumentation methods that enable the controlled execution of a target program and the execution of user-defined code.

Using several techniques such as inserting breakpoints, trampolines, or dynamic compilation, DBI systems can interleave code or temporarily stop the execution at specific points (e.g., when a breakpoint is hit, or whenever an instruction sequence is compiled), or when certain events are triggered (e.g., interactions with the OS). These systems offer a set of instrumentation primitives (i.e., facilities through an API that allow the user to run callback routines at different granularities). Depending on the technique and, in certain cases, the Instruction Set Architecture (ISA) abstractions provided by the DBI framework, diverse events are exposed, such as the execution of instructions, memory accesses, or OS and hardware events. Moreover, the design of the DBI engine determines whether these user-defined callbacks are executed in the same context as the target under analysis, in a separate process (i.e., a debugger process), or in a higher privilege context like the kernel (Ring 0) or the hypervisor (Ring -1).

Instrumentation techniques, capabilities, and related primitives are classified and further detailed in Section 4.2.

4.1.3 Introspection

Another relevant aspect of DBI is related to the ability to observe the program state at different points of the execution.

In general, DBI frameworks provide means through the API to access the CPU register state and memory contents. Similarly, they may also facilitate introspection by means of higher level abstractions passed to analysis calls as arguments. Besides, some tools may provide complementary methods to access devices like hard disk drives, and so on. Even though register values and raw memory contents are sufficient for some tasks, a high-level representation closer to the original source, such as variables, data structures, or functions, is extremely useful in many cases (e.g., vulnerability analysis, data manipulation). To this end, the conventional method is to leverage standard debugger techniques that explore the symbol table and debugging information if present [GDB22, Hex22].

In whole-system approaches, introspection poses a greater challenge since they must confront the semantic gap problem. The semantic gap is

defined as a disparity in abstractions. It fundamentally arises when the DBI system has an external point of view (e.g., at hypervisor-level), and high-level OS abstractions, such as threads and processes are not visible. In contrast, hardware-level abstractions such as physical memory pages and hardware device operations can be observed. To bridge the semantic gap, numerous research works have been proposed in the area of Virtual Machine Introspection (VMI) [DGLZ+11, FL12, SFL14, ZDXG17]. This process tries to retrieve information about the current state of the guest by reconstructing high-level semantic information, such as lists of running processes, or active network connections from low-level sources and detailed knowledge of the OS [Coh22]. The reader can refer to existing literature [JBZ+14] for further information on this topic.

The Semantic Gap Problem

The semantic gap is defined as a disparity in the abstractions. It fundamentally arises when the instrumentation system has an external point of view (e.g., hypervisor-level) and high-level OS abstractions, such as files and processes, are not visible. In contrast, hardware-level abstractions such as physical memory pages and hardware device operations can be observed. To bridge the semantic gap, numerous research works have been proposed in the area of Virtual Machine Introspection (VMI) [DGLZ+11, FL12, SFL14, ZDXG17]. This process tries to retrieve information about the current state of the guest OS by reconstructing high-level semantic information, such as lists of running processes, or active network connections, from low-level sources and detailed knowledge of the OS.

Kernel data structure reconstruction generally relies on a learning and search methodology. On the one hand, the learning phase usually involves extracting data structure signatures, for which three main approaches have been proposed. Firstly, hand-crafted signatures can be built based on expert knowledge of the OS internals [Wal22]. Secondly, tools may rely on source code analysis or debugging information [LRZ+11, Coh22]. Finally, some approaches leverage dynamic analysis to train on a trusted OS instance to then generate invariants and classify data structures [DGSTG09]. The searching phase is performed either by linearly scanning kernel memory, or by traversing data structure pointers, starting from public symbols.

Alternative methods also exist that do not seek to directly reconstruct data structures, but also attempt to bridge the semantic gap. The simpler approach consists in injecting an agent into the guest that reports semantic

information back. In that context, process implanting [GDXJ11] deploys a monitoring process within a randomly-selected process already present in the guest. Further, process out-grafting [SWJX11] relocates a suspect process to a trusted VM while redirecting system calls to the monitored VM, which allows tools in the trusted VM to directly examine the process. VMST [FL12] extends this approach to inspect the entire OS by selectively redirecting memory accesses to the monitored VM.

4.1.4 Execution Manipulation

Regardless of the level of introspection that every approach may offer, the objective of DBI is not only to monitor the target under analysis, but to possibly alter its behavior through the intentional manipulation of the execution.

To this end, DBI frameworks may provide facilities through their APIs to add, replace, or delete instructions by design. Moreover, other convenient facilities include replacing specific function calls with user-defined functions, returning prematurely, or modifying arguments and return values, for instance. Nonetheless, program behavior and execution flow can also be altered through the manipulation of the program state, which is usually accessible to the user. Writing CPU registers or memory contents can lead the program to follow a different path, by modifying the program counter or overwriting a branch condition related value, for instance.

4.1.5 DBI Setup

DBI systems have the mandatory requirement of controlling the inspected target. To this end, the system usually resides in a high-privileged environment or acquires the privilege to observe and control the target through other means.

To tackle this challenge in process-level approaches, the most common method is to load a shared library into the target's address space — a procedure known as injection. In Linux-based systems, the `ptrace` system call can be used to allocate memory in the target process and load the corresponding library, or otherwise add the library path to the `LD_PRELOAD` environment variable. In a similar vein, in Windows several registry keys exist that are used to list libraries to be loaded into particular processes that meet some given conditions (e.g., make specific API calls, and so on). In addition, other methods make use of debugging facilities, and process

and thread manipulation functions. There are some further alternatives that are specific to the different operating systems.

Regarding whole-system approaches, they already get some sort of higher privilege since they recreate entire systems, residing in a higher privileged ring (e.g., hypervisors). However, the execution of the target introduces a new problem: it can be manually started, but for automated analysis it requires other methods such as starting the execution by an in-guest agent, or leveraging VMI techniques to implement a process hijacking mechanism [LMP+14].

4.2 Instrumentation Techniques

The variety of events that a DBI system is able to instrument is subject to its underlying techniques and implementation decisions. Fundamental observable events (i.e., primitives) include code execution and memory accesses at different levels. The DBI approaches proposed so far can be divided into different categories depending on the techniques they are using in order to instrument the execution of the target. These techniques have different implications in terms of performance, instrumentation granularity, and the variety of events they can monitor.Table 4.1 summarizes the main primitives in relation to the techniques that enable their instrumentation.

In contrast to previous work [DCN+19], we do not include high-level abstractions that are implementation dependent such as function calls, since our goal is to express basic aspects of program execution. Nevertheless, DBI systems frequently use symbol information or function identification methods.

We also take a restrictive approach. We consider a technique to be capable of instrumenting a primitive only if it is able by itself and if it is reasonably applicable. For example, single-stepping (i.e., tracing every instruction individually) cannot selectively instrument a specific instruction address, even though it can single-step the entire program and examine every instruction until that particular address. On the contrary, other techniques can instrument specific instructions because the technique itself makes it possible.

Table 4.1: Instrumentation techniques and fundamental primitives. ✓indicates full capability. ✗means that the technique is not capable. ○indicates extraordinary case or partial capability. Specific cases are detailed in their corresponding subsections.

Instrumentation Primitive	***JIT-DBT***	***DPI***	***SW Breakpoints***	***HW Breakpoints***	***Single-Stepping***	***Page Faults***	***CPU Interpretation***
Execution of a Single Instruction	✓	✓	✓	✓	✗	○	✓
Execution of Instructions in Memory Range	✓	✗	✗	○	✗	✓	✓
Execution of Every Instruction	✓	✗	✗	✗	✓	✗	✓
Execution of Specific Instructions (e.g., branches)	✓	✗	✗	✗	○	✗	✓
Execution of Execution Block	✓	✗	✗	✗	✗	✗	✗
Block/Instruction Translation/Decoding	✓	✗	✗	✗	✗	✗	✓
Memory Address R/W	○	✗	✗	✓	✗	○	○
Memory Range R/W	○	✗	✗	○	✗	✓	○
Whole Memory R/W	○	✗	✗	✗	✗	✗	○

4.2.1 JIT-assisted Dynamic Binary Translation

Dynamic Binary Translation (DBT) is a widespread technique extensively used in technologies like emulation [Bel05] that also enables instrumentation [HPY+14, DGHH+15, Uga22]. It dynamically translates code from one ISA to another. However, it is common that host and guest architectures coincide [LCM+05, Bru04]. Moreover, approaches may work on the native instruction set or use an intermediate representation [NS07].

The process is assisted by a JIT compiler and a code cache. The original code is not executed but usually compiled one block at a time, and stored in a software-based code cache, which is the one that is actually executed. All the instructions in this chain are executed one after the other unless some interrupt or control flow instruction is reached. This optimizes the performance as it avoids switching between the emulator and the emulated code on every single instruction.

These engines allow registering callbacks on the translation or execution of code at different granularity: typically at execution block level and instruction level, depending on the implementation. In addition, some systems may abstract memory accesses, exceptional control flows, or other

CPU and cache related events. These primitives can be used to emulate any kind of hook by selectively inserting callbacks in JIT translated code (e.g., at a certain memory address). Concrete memory accesses cannot be instrumented at translation time, since in most cases the addresses are computed at run-time. Instead, it has to examine executed instructions, or directly monitor memory accesses in cases where the implementation uses a software Memory Management Unit (MMU). It is worth to note that JIT-DBT is the only technique that naturally implements the execution block granularity level, facilitating coarse-grained instrumentation and analysis.

4.2.2 Dynamic Probe Injection

The Dynamic Probe Injection (DPI) approach dynamically replaces instructions in the original program with trampolines that jump into analysis code. To this end, instructions are saved and replaced by an unconditional branch instruction. When this trampoline is executed, the control flow changes to a pre-processing sequence that pushes CPU registers into the stack to save the program state and execute the user-defined code. When the callback returns, the program state is restored, the instructions that were displaced are executed, and the control flow returns to the original location [BM11, Rav22].

This method is typically used for function-level hooking (i.e., intercept calls to functions and execute code before or after the call), although it is possible to target instructions at specific addresses to instrument. In order to identify functions for instrumentation, DBI frameworks must rely on symbol tables, debugging information, or other techniques such as previous static analysis to recover function boundaries.

4.2.3 Traps and Interrupts

Techniques commonly found in debuggers trigger traps or interrupts at specific execution points, redirecting the execution to user code that will typically have privileges over the target, as part of a debugger process or a hypervisor. In this category we find software and hardware breakpoints, watchpoints, single-stepping, and page-fault-based instrumentation.

Software-based breakpoints start by saving a specified instruction in the program to replace it with an exception-triggering instruction. For instance, in x86-64 platforms it is usually a special single-byte instruction (i.e., `INT 3`). After handling the exception and executing callback code,

the original instruction is written back and executed [DZX13, LMP+14, rVM17]. Some approaches are built around architecture-specific mechanisms such as the System Management Mode (SMM). In that case, software breakpoints are implemented by replacing instructions with a write to an Advanced Configuration and Power Interface (ACPI) port (e.g., 0x2b on Intel) to trigger a System Management Interrupt (SMI) and enter SMM [ZLS+15]. On ARM, they are also implemented placing Secure Monitor Call (SMC) instructions into the guest kernel to transfer control to a hypervisor [PLM+18]. Similar to DPI, although this method could theoretically instrument every instruction or instructions in a memory range, it would require setting and managing successive breakpoints on each address, if possible.

Hardware-based breakpoints are implemented at CPU-level by using dedicated debug registers (e.g., DR0-DR3 in the x86-64 architecture, or DBGBVR in ARM). These registers are a limited resource and can store a breakpoint address. When the program counter matches a value and a set of conditions are met (i.e., determined by the DR7 register in x86-64, or DBGBCR in ARM), a debug exception is triggered. Some architectures such as PowerPC may also include hardware support for ranged breakpoints [GDB22].

Watchpoints are a special kind of breakpoint that breaks when data is accessed rather than when some specific instruction is executed. Hardware-assisted watchpoints are set leveraging architecture-specific debug registers [PLM+18], with support for ranged watchpoints on architectures such as PowerPC [GDB22]. In contrast, we do not contemplate purely software-based approaches since they require single-stepping the program and monitoring every memory access to compare the address against a list of watched addresses. Nevertheless, several optimizations have been proposed in that context [GXLA12].

Single-stepping includes any technique that executes and handles one step (i.e., an instruction) at a time, usually through mechanisms provided by CPU architectures. In x86-64, the EFLAGS register contains the Trap Flag (TF), that can be set to generate a debug exception on each executed instruction [DRSL08, NSJ+09]. In addition, the Branch Trap Flag (BTF) modifies the behavior of TF to only trigger exceptions on branch instructions. Similarly, the Monitor Trap Flag (MTF) is a debugging feature specific to Intel hardware virtualization extensions that causes VMExits on instruction boundaries in VMX non-root operation (i.e., the guest system) [DZX13, rVM17]. Further, when a performance counter overflows an interrupt is

triggered, enabling instruction-level stepping by setting the value of a performance counter that measures all retired instructions to the maximum value each time. The CPU architecture determines stepping options by the kind of events that can be measured. For instance, it is also possible to perform branch-level stepping in this way. The triggered interrupts can be configured to transfer control to the hypervisor, SMM, or the ARM secure domain [VE12, ZLS+15, NZ17]. On ARM, single-stepping can also be implemented placing successive SMC instructions, taking advantage of the fixed-width ISA [PLM+18].

Page Faults operate at the memory page level and tamper with page access bits to trigger page fault exceptions during execution, that are handled by a previously set custom page fault handler. To this end, a page that contains an instruction of interest can be set to non-present [VY05]. Further, when using Intel virtualization extensions and hence Extended Page Tables (EPT), read, write, and execute page access bits are independent of each other, providing complete flexibility to trap the execution on different access types, triggering a VMExit [YJZY12]. This method is not exclusive to code pages, but is also applicable to data pages, allowing to implement a mechanism similar to watchpoints. For instance, the privilege (i.e., supervisor bit) or other access permission flags of the page can be manipulated to produce an exception on every access [VY06, DZX13].

In order to monitor the execution of concrete instructions or specific memory accesses, it is necessary to check on every page fault the memory address that caused it. Following the same principle, it is also possible to monitor memory ranges not aligned to the page size for either read, write or execution.

4.2.4 CPU Interpretation

The fetch-decode-execute cycle is the basic operational process the CPU follows to execute instructions. This process is closely reproduced in software by some emulators. In particular, interpretation-based emulators work in a loop by fetching an instruction from memory, decoding it to determine its semantics, fetching the operands, performing the operations, and writing the result back. Instruction execution is usually enclosed in an instruction-specific method that is called after its decoding. This rigorous interpretation enables instruction-level execution control and instrumentation [Law96, Pie22].

Similar to JIT-DBT, this technique needs to examine each executed in-

struction to check whether it is a memory operation to make memory accesses visible for instrumentation. However, interpretation-based emulators usually emulate virtual memory in software, making this task simpler.

4.3 DBI Implementations

In this section, we describe and relate existing DBI frameworks to their implemented instrumentation techniques mapped to instrumentation primitives. Please note that we have excluded tools that mix or are built upon other DBI systems (e.g., PEMU [ZFL15]), and tools whose details are not publicly available. Similarly, we do not include tools with a more specific purpose such as TinyInst [Goo22] (whose goal is to instrument specific modules in a process). We also do not consider the latest versions of QEMU that include TCG plugins support since they lack fundamental introspection and execution manipulation capabilities.

4.3.1 Process-Level Approaches

Table 4.2 shows the implemented instrumentation techniques within known process-level tools that enable the instrumentation of different primitives.

Existing JIT-DBT tools share address space with the target program and have a similar structure [DCN+19], except for a few cases. Cobra was developed as a kernel module to enable the analysis of user and kernel code, whereas Unicorn is based on the CPU emulation component of QEMU, which enables emulation and instrumentation of arbitrary binary code such as firmware blobs. Further, Frida injects a JavaScript engine into the target process to enable DBI through JavaScript code.

Each tool implements different designs, translation strategies and optimizations, and operate on native code. The exception is Valgrind, which translates through the VEX intermediate representation and enables heavyweight analyses. Other differences include DynamoRIO and Valgrind not having an instruction-level event to register callbacks. The user needs to subscribe to execution block events and then iterate through the instruction list.

Alternative approaches do not share the same address space and operate from an external point, e.g., from a different process or a different physical machine (i.e., Malt). DynInst implements several static analyses to identify and build abstractions that enable the instrumentation of more complex primitives using DPI. To illustrate, analysis tools can build the

Table 4.2: Process-level DBI tools mapped to instrumentation techniques and primitives. J stands for JIT-DBT, D for dynamic probe injection, SB for software breakpoints, HB for hardware breakpoints, SS for single-stepping, PF for page faults, and M for software emulated memory. ✗denotes not capable.

	JIT-DBT							DPI		Others	
Instrumentation Primitive	*Pin* [LCM+05]	*DynamoRIO* [Bru04]	*Valgrind* [NS07]	*Frida Stalker* [Rav22]	*Unicorn* [QV22]	*Cobra* [VY06]	*DynInst* [BM11]	*Frida Interceptor* [Rav22]		*MALT* [ZLS+15]	*GDB* [GDB22]
Execution of Single Instruction	J	J	J	J	J	J	D	D		SB, HB	SB, HB, PF
Execution of Instructions in Memory Range	J	J	J	J	J	J	✗	✗		✗	HB*, PF
Execution of Every Instruction	J	J	J	J	J	J	✗	✗		SS	SS
Execution of Specific Instructions (e.g., branches)	J	J	J	J	J	J	D*	✗		SS	✗
Execution of Execution Block	J	J	J	J	J	J	D*	✗		✗	✗
Block/Instruction Translation/Decoding	J	J	J	J	✗	J	✗	✗		✗	✗
Memory Address R/W	J	J	J	J, PF	M	PF	D*	PF		✗	HB, PF
Memory Range R/W	J	J	J	J, PF	M	PF	D*	PF		✗	HB*, PF
Whole Memory R/W	J	J	J	J	M	✗	✗	✗		✗	✗

* Partial capability, specific to a CPU architecture, or tool-specific implementation.

control-flow graph of specific functions and instrument basic block boundaries (different from execution blocks in JIT-DBT). Similarly, it also enables the instrumentation of load and store instructions in particular functions.

In GDB, page-permission-based traps are not implemented by default. However, the user can call arbitrary functions in the debugged process space. This enables their implementation by calling `mprotect` to modify page permissions. Resulting page faults are handled by the debugger process, permissions are restored back in the faulting page, the instruction is executed in single-step, and page permissions are adjusted again before the execution continues. GDB also implements ranged hardware breakpoints and watchpoints for PowerPC embedded processors. Although GDB provides software watchpoints, they are implemented by single-stepping the program and checking the expression on each instruction. Therefore, we do not take this method into account.

Alternatively, MALT combines the SMM and performance counters to implement several stepping methods leveraging performance counter events to trigger an SMI: retired instructions, taken branches, far control transfers, and near returns. This also enables the instrumentation of these specific instructions tracked by performance counters.

4.3.2 Whole-System Approaches

Table 4.3 shows the implemented instrumentation techniques within whole-system tools that enable the instrumentation of primitives. These tools are built upon hardware virtualization, emulation, or a combination of both technologies.

JIT-DBT is the most common approach in emulation-based tools, many of which are based on QEMU's whole-system emulator. QEMU translates each guest virtual address into a guest physical address, and then translates that into a host virtual address. Therefore, the instrumentation for MMU-related events is straightforward. However, significant differences exist between tools. S2E was designed for symbolic execution using the KLEE symbolic execution engine [CDE08] and the LLVM tool chain [LA04]. PANDA features a record and replay system, and leverages the TCG-to-LLVM translation backend from S2E to provide a framework for complex analyses such as taint analysis. DECAF implements its own taint analysis engine. V2E is a hybrid approach that records the execution at page-level in KVM, and replays it in a modified version of TEMU [YS10], taking advantage of JIT-DBT to perform fine-grained instrumentation. In contrast,

Table 4.3: Whole-system DBI systems mapped to instrumentation techniques and primitives. J stands for JIT-DBT, D for dynamic probe injection, SB for software breakpoints, SS for single-stepping, PF for page faults, CI for CPU interpretation, and M for software emulated memory. ✗denotes not capable.

	Emulation							Hypervisor				
Observable Event	DECAF [HPY+14]	PANDA [DGHH+15]	V2E [YJZY12]	S2E [CKC11]	PyREBox [Uga22]	Bochs [Law96]	Ether [DRSL08]	Drakvuf [LMP+14]	rVMI [rVMI17]	HyperDbg [KGKM+22]		
Execution of Single Instruction	J	J	J	J	J	CI	✗	SB, PF	SB	SB, PF, D*		
Execution of Instructions in Memory Range	J	J	J	J	J	CI	✗	PF	✗	✗		
Execution of Every Instruction	J	J	J	J	J	CI	SS	✗	SS	SS		
Execution of Specific Instruction Types (e.g., branches)	J	J	J	J	J	CI	✗	✗	✗	PF*		
Execution of Execution Block	J	J	J	J	J	✗	✗	✗	✗	✗		
Block/Instruction Translation/Decoding	J	J	✗	J	✗	CI	✗	✗	✗	✗		
Memory Address R/W	M	M	M	M	M	M	✗	PF	HB	PF		
Memory Range R/W	M	M	M	M	M	M	✗	PF	✗	PF		
Whole Memory R/W	M	M	M	M	M	M	PF*	✗	✗	✗		

* Partial capability or tool-specific implementation.

Bochs is an interpretation-based emulator. Emulation logic is implemented in one big decode loop, which closely models the fetch-decode-execute actions of the CPU, enabling fine-grained instrumentation.

Other whole-system tools are built upon an hypervisor component such as Xen or KVM [LMP+14, rVM17, DRSL08]. For example, Drakvuf and HyperDbg leverage EPT to implement breakpoints and memory access traps. Since these events are handled by the hypervisor, they are transparent to the guest system. Ether was implemented when virtualization systems used shadow page tables instead of EPT, and is able to intercept memory writes by removing write permissions from all pages.

Numerous DBI frameworks have been developed for the general purpose Intel architecture. Nevertheless, ARM provides its own set of hardware features to build DBI systems around, such as virtualization extensions, for instance. Although hardware-assisted virtualization strongly differs from architecture to architecture, it always introduces a new privileged mode especially designed for the hypervisor (i.e., ring -1), and a second-level address translation mechanism.

In a recent work [PLM+18], the DRAKVUF framework was extended to support instrumentation and analysis of ARM binaries. They enhanced the Xen hypervisor on ARM to dynamically define and switch among different guest memory views, and proposed an alternative ARM-specific method to place breakpoints using Secure Monitor Call (SMC) instructions, which also enabled the implementation of a novel single-stepping mechanism. Likewise, NINJA [NZ17] leverages ARM hardware features such as the TrustZone hardware environment, the Performance Monitor Unit (PMU), and the Embedded Trace Macrocell (ETM) to implement tracing and classic debugging functionality.

4.3.3 Other Execution Events

During execution, the execution flow can be altered by exceptions and interrupts. Some of these come from the interaction with external devices such as the hard-disk drive, the keyboard or the network card. The OS is usually responsible for their management. Some DBI tools have visibility on these events, subject to implementation decisions and symbol availability. Table 4.4 shows the relation between DBI tools and the most common event types within this category. We conservatively take into consideration current implementations and default plugins, which usually expose these events through their APIs.

Table 4.4: Other visible events derived from execution, OS, or hardware operation.

	Process-Level										Whole-System									
Observable Event	*Pin* [LCM+05]	*DynamoRIO* [Bru04]	*Valgrind* [NS07]	*Frida* [Rav22]	*DynInst* [BM11]	*Cobra* [YY06]	*Unicorn* [QV22]	*GDB* [GDB22]	*DECAF* [HPY+14]	*PANDA* [DGHH+15]	*S2E* [CKC11]	*PyRebox* [Uga22]	*Ether* [DRSL08]	*DrakvufJ* [LMP+14]	*RVMI* [rVMI17]	*V2E* [YJZY12]	*Bochs* [Law96]	*HyperDbg* [KGKM+22]
Interrupts & Signals	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✗	✗	✓	✗	✓	✓	✗
Process Creation & Exit	✓	✓	✓	✗	✓	✗	✗	✗	✓	✓	✓	✓	✗	✓	✗	✓	✗	✓
Thread Start & Exit	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗	✗	✓
System Call	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓
Module Load & Unload	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✗	✗	✓	✗
Task Switch	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓	✗	✓	✗	✓	✓	✗
Network I/O	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
HDD I/O	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
TLB Operation	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

Process-level and whole-system tools are implemented with different goals in mind. Therefore, they observe events from different perspectives and some events may not be captured by a particular category. Specifically, as process-level approaches focus on a single program, they track events that take place in the content of the program under analysis, while whole-system DBI observes events in a system-wide fashion. As an example, process-level tools can intercept the creation of child processes originating from the target. Whole-system approaches can instead monitor the creation of every process. Likewise, process-level approaches capture module loads and unloads in the process (i.e., shared libraries), whereas whole-system solutions also cover kernel modules or drivers.

Existing tools adopt different strategies to handle interrupts. Process-level tools let the user subscribe to exception events (e.g., division by zero) or Linux-specific signal events, depending on the implementation. Valgrind and DynInst only support signals while Frida supports exceptions. The Unicorn Engine is rather exceptional since it is a CPU emulator and thus, although it provides callback functions for exceptions and INT instructions, it is the user's job to emulate the entire exception handling process. Whole-system approaches do not usually provide primitives to monitor signals or exceptions, but they have full visibility on the system, making it possible to create these abstractions by using VMI techniques.

4.4 DBI Security Concerns

Because of their nature, DBI systems face a great challenge regarding transparency. In order to execute applications without errors, execution under a DBI system must pose as a native execution and, therefore, perform its own operations transparently. However, correctness is not enough when analyzing evasive programs that will end the execution, behave differently, or even escape from DBI's control [DCN+19] if its presence is detected.

4.4.1 Transparency

Transparency can be defined as a property that represents how successfully an analysis/instrumentation system mimics the actual behavior of a regular system in order to appear indistinguishable to the target. This usually includes the capacity to react to evasion tricks. Although it is commonly associated with malware analysis [ESKK12], it also applies to any program

that would conceal or alter its functioning in the presence of an analysis/instrumentation system, such as proprietary software.

To build a transparent system, previous work defined a set of requirements to be fulfilled. In [DRSL08], authors specify five different requirements: (i) higher privilege than the maximum a target can gain, (ii) no non-privileged side effects visible to the target, (iii) identical basic instruction execution semantics, (iv) transparent exception handling, and (v) identical measurement of time.

Similarly, the authors of NINJA [NZ17] propose three alternative requirements. In the first place, the environment must be isolated. Second, the environment must exist in an off-the-shelf bare-metal platform without modifying the software or hardware. Finally, the analyzer program should not leave any detectable footprints.

4.4.2 Known Evasion Techniques

During the last decade several survey papers that gather, describe, and classify known evasion methods and countermeasures have been conducted [CAM+08, PCM+17, BY17, ANSB19, DCN+19]. We encourage the reader to refer to these papers for a more complete understanding since we only provide a quick summary of the findings.

Common evasion techniques can usually be classified into two complementary groups: those intended to detect an analysis environment, and those aiming to ensure that the system is of a regular human user. Exception exists when more targeted malware will only execute in a particular physical machine and a particular configuration (e.g., APTs, Stuxnet, etc.) or only in a specific time period.

Environment detection Environment detection techniques can target specific platforms and also particular fingerprints. The platforms and fingerprints they usually target are the following:

- *Environmental artifacts.* These artifacts are system-related artifacts that uniquely reveal it. This category includes, for instance, disk files, specific devices with identifiable attributes, additional drivers, running processes, or usernames.

- *Timing.* Timing-based detection tries to detect discrepancies in timing when executing a set of operations in a native environment com-

pared to an instrumentation environment. However, more sophisticated attacks can be carried out [BBR16]. This detection approach is probably the most difficult to tackle due to the many timers and counters present on modern systems in addition to external timers.

- *Process introspection.* Some systems need to make program modifications for proper operation, thus introducing several in-process observables that can be detected by examining the program state. These cases include, for instance, injected libraries, code modification such as function hooks, debugger presence, or even improper handling of self-modifying code.

- *Network artifacts.* Several systems within the malware analysis use case usually provide carefully adjusted network features to prevent potential risks. In consequence, uncommon network activity such as extremely fast internet service, or network isolation could be used as an indicator of analysis.

- *CPU emulation.* Red pill is the common term used to refer to specific instructions that behave differently when executed inside an emulator than on a real CPU as a result of, for example, incorrect emulation [RKK07, PMRB09, SAM14]. Furthermore, as determining perfect emulation is considered an undecidable problem [DRSL08], red pills will be always a major concern.

Reverse Turing Tests In order to detect previous or active human interaction, evasive programs check for several signs such as coherent mouse and keyboard inputs, recently opened files, or the so-called wear-and-tear artifacts [MANP17], which are aspects of a system that are affected as a result of normal use (e.g., browsing history, event log, DNS resolver cache).

4.4.3 Known Countermeasures

The existing mitigation approaches are devoted to block the evasion attempts performed by the target under inspection. These countermeasures are usually classified in two different categories: (i) active approaches that involve detection and mitigation of the evasion techniques, and (ii) passive approaches that increase the analysis transparency changing the environment. Among these classification, the following types of mitigation actions can also be described according to their behavior.

Binary modification Taking advantage of operations known to be used for detection, a mitigation strategy attempts to dynamically identify and safely handle them at code level while preserving the semantics. As an example, Cobra scans JIT-generated code blocks and replaces relevant instructions in the code cache with what they call stealth implants.

Artifact hiding Detection methods abuse any kind of evidence left by the system, therefore, a straight-forward approach to reduce the probability of detection seeks to hide artifacts that could reveal either the presence of instrumentation primitives or the environment. For example, Ether relies on the trap flag to perform instrumentation and, consequently, it hides the trap flag by intercepting relevant instructions to provide the expected state of the flag. Similarly, it also controls queries about time to answer with expected values in order to prevent time-based detection.

Path exploration Although path exploration has not been directly applied to evasion mitigation, it is extensively used in related malware analysis tasks such as binary unpacking [UPBSB16], force execution [PDZ+14], or trigger-based behavior discovery [BHL+08]. In fact, this conditionally executed behavior could match an evasion attempt, which could be mitigated by forcing the execution under a different path.

State modification Similar to binary modification, some works [KYH+09, ASBM14] proposed a dynamic state modification approach that modifies relevant registers and memory with appropriate values to simulate a physical environment in the presence of evasion code.

Bare-metal analysis In search of a general increase of transparency, researchers have been turning between emulator and hypervisor platforms, although both of them have been shown to be detectable [RKK07, ZLS+15]. As an alternative, several proposed systems such as MALT run on bare-metal, thus reducing the likelihood of detection. However, as companies are increasingly adopting virtualization technology, malware started to also target virtualized environments, not evading them like in the past.

Realistic system images In order to overcome several reverse Turing tests and specifically wear-and-tear artifact-based evasion, two solutions were proposed. Firstly, a clone of a real user system could be used as a starting

system image. Secondly, a fresh operating system image could be taken as basis, to then artificially age it by automatically simulating user actions such as software installation, or webpage browsing.

4.5 Performance Evaluation

In this section, we detail the conducted experiments to measure and evaluate the performance of the different instrumentation techniques. We seek to understand the behavior of instrumentation techniques in different scenarios, such as targeting different primitives, or different event frequency.

4.5.1 Instrumentation Techniques in Practice

Certain techniques such as JIT-DBT are quite complex and, as a consequence, tools implementing the same technique may show performance differences due to their particular implementation or design choices, among other factors. Therefore, our main goal is to evaluate instrumentation techniques and not their particular implementations or available tools.

However, in order to conduct these experiments we had to select a set of tools as reference implementations of the techniques. On the one hand, we prioritized actively maintained projects written in C/C++ with a large community. The implementation language of both the tool and the instrumentation code have a significant performance impact. For instance, a tool written in C/C++ that calls a Python snippet needs to enter the Python run-time on every event, imposing a high overhead. This overhead is derived from a design choice and not by the instrumentation technique itself. All the tools we selected are fully implemented in C/C++. On the other hand, we conducted our experiments on the x86-64 architecture, which is fully supported by most tools. While there are DBI tools that support other architectures, they follow similar techniques and principles. Also, it would not be appropriate to compare x86-64 implementations against other architectures.

In this way, we selected tools to cover every possible instrumentation technique for process-level and whole-system. Each of these tools implement one or more of the instrumentation techniques described in Section 4.2.

More specifically, we used the following process-level technique implementations: the JIT-DBT engine Pin (3.17), the DPI tool Dyninst (10.2.1),

Table 4.5: Selected instrumentation tools.

Tool	Alias
Pin and PANDA for JIT-based instrumentation	`pin` `panda`
Dyninst for Dynamic Probe Injection	`dyninst`

GDB for software breakpoints, hardware breakpoints, single-stepping and page faults	`gdb_sb`
	`gdb_hb`
	`gdb_ss`
	`gdb_pf`

Drakvuf for software breakpoints and page faults	`drakvuf_sb` `drakvuf_pf`
Bochs for execution interpretation	`bochs`

and the GDB (9.2) debugger for software breakpoints, hardware breakpoints, and page-fault-based instrumentation. While GDB is a debugger and is not considered a DBI tool on its own, it implements several instrumentation techniques and satisfies the definition of DBI provided in Section 4.1. Also note that we configured GDB to prevent all output to avoid introducing unnecessary overhead.

As whole-system implementations, we selected: the interpretation emulator Bochs (2.6.11), and the latest available versions of the JIT-DBT emulator PANDA and the hypervisor-based tool Drakvuf. These tools cover a wide range of techniques, described in Section 4.2. In cases where callbacks can be inserted in different programming languages (e.g., PANDA supports plugins in Python), we used the C/C++ interface.

Note that in the rest of this section, in order to make a distinction between the different techniques implemented by the same tool, we append the acronym of the technique at the end of the tool name as listed in Table 4.5 (i.e., `gdb_sb`, `gdb_hb`, `gdb_ss`, `gdb_pf`, `drakvuf_sb`, and `drakvuf_pf`).

4.5.2 Technique Evaluation

Our goal is to analyze current instrumentation techniques, evaluating their performance when implementing different instrumentation primitives. It is essential to minimize the impact of specific implementations. These

rules out complex test cases that use multiple instrumentation options, or high demanding tasks such as memory profiling or information flow tracking, because tool-specific optimizations would come into play. Under that premise, our experiments are guided by and reflect basic instrumentation capabilities.

We did not cover the execution of specific instruction types, execution of execution blocks, or code translation primitives. While these are useful primitives in many contexts and applications, reporting their performance statistics would not add any additional value since they are either based on the same principles as other primitives in the techniques, or not applicable to more than one technique. For instance, execution block primitives are exclusive to JIT-DBT by design.

It is worth noting that some instrumentation techniques were really demanding, leading to overly long execution times and making the measurement impractical. To overcome this difficulty, we empirically set an event count threshold for these cases: 2 million instructions when single-stepping, and 200,000 page faults when using a page-fault-based approach.

No instrumentation (*none*). For every tool, we run the target under the DBI framework with no instrumentation besides the necessary to measure performance. This serves as a baseline to understand the base overhead introduced by every tool.

In Bochs, identifying the process start and end cannot be done using lightweight methods and requires using the instruction-level callback. Unfortunately, it introduces excessive overhead for this experiment. To overcome this limitation, we compiled Bochs without any instrumentation enabled, and connected an agent in the guest system with the host through a network socket in order to properly detect the beginning and end of the program without adding any expensive overhead.

Instrument execution of a single instruction (*exec_single*). This primitive can be implemented in different ways. JIT-DBT engines can decide whether to instrument an instruction at translation time based on the address being translated. This is the case of Pin and PANDA. DPI techniques can insert trampolines at specific addresses – we implemented this approach in Dyninst. Software and hardware breakpoints trigger an exception at a specific code address. A similar behavior is possible by removing the execution permission of the whole memory page containing the address. We implemented these three approaches in GDB and Drakvuf (except hardware breakpoints for the latter). Finally, when an interpreter runs every instruction one by one, it can call a callback before instruction execution.

We tested this approach with Bochs.

Instrument execution of instructions in a memory range (*exec_range*). Similar to previous cases, JIT-DBT engines can check at instruction translation whether the current program counter value is within the range of a specified memory range. We tested this with Pin and PANDA. This technique is also possible through page faults. When the memory range is not aligned to page boundaries, it is necessary to check (as part of the instrumentation code) whether the faulting address is within the specified range. We implemented this technique in GDB, and Drakvuf. As in the previous case, the Bochs interpreter can check the address instruction by instruction.

Instrument every executed instruction (*exec_all*). Not every instrumentation technique has the capacity to instrument every possible instruction in a straightforward manner. This primitive can be implemented either by using single-stepping (that will force an exception on every instruction) or by monitoring every instruction execution under a JIT-DBT engine or an interpreter. We tested the first technique with GDB, and the second one with Pin for process-level, and PANDA and Bochs for whole-system instrumentation.

Instrument memory read/writes to a given memory address (*rw_single*). We implemented this primitive using different techniques. Hardware breakpoints trap the execution on memory accesses. It is also possible to implement this behavior at page granularity by removing read and write permissions of memory pages. We implemented these two techniques in GDB. For JIT-DBT and interpretation, the only way is to monitor every memory access to check whether the accessed address matches. Due to indirect memory addressing, many of the addresses are computed at run-time, and it is not possible to instrument this primitive at translation time. We implemented this approach in Pin, PANDA, and Bochs.

Instrument memory read/writes to a given memory range (*rw_range*). We implemented this instrumentation primitive by using the same techniques described previously. We implemented them in Pin, GDB (only using page faults), PANDA, and Bochs. In a similar vein, we included Drakvuf in this experiment by setting a memory access trap for read and write accesses at the specified memory pages at the EPT level.

Instrument every memory read/write made by the program (*rw_all*). This final primitive was only implemented by using JIT-DBT and interpretation, monitoring every single memory access at run-time. We implemented this test in Pin, PANDA, and Bochs.

4.5.3 Testing Benchmarks

To conduct our experiments, we used the SPEC CPU2006 suite, which contains diverse programs written in C or C++, integer or floating-point intensive, and with varying execution times, instruction footprints and cache sensitivity. We discarded those that did not compile or execute correctly in our system, or those for which their execution times were excessively long. Out of the three different sizes of input data that SPEC CPU2006 provides (i.e., test, train, and reference), we used the train inputs since we observed that the reference workload was too heavy, making execution times excessively long.

The SPEC suite defines multiple executions in a single benchmark for some programs, i.e., using different command-line options. We present these cases individually. Hence, we finally used a total of 16 benchmarks from 10 unique binaries.

For the experiments that require specific addresses to instrument, we manually searched for instructions and memory addresses that were respectively executed or accessed a significant number of times in each binary.

4.5.4 Evaluation Methodology and Metrics

To manage and automate experiment execution, we developed a launcher program that executes every test in every tool and every SPEC program in the evaluation set, and recording execution information. We executed each benchmark 10 times to minimize the effect of system irregularities on the results.

To keep consistency, we use the execution of `main` as the starting point and the execution of `exit` as the ending point. When the execution reaches these, the tool notifies the launcher and retrieves time and status information. Since each tool provides different methods to identify the start and end points, we selected the least intrusive method to avoid adding overhead. Note that since we measure from `main` to `exit`, we avoid measuring time that tools would need to repeat their initialization (e.g., creating a code cache) on every test case, minimizing the impact of the tools on the measured executions.

Regarding Pin, Dyninst, and GDB; we use standard tool features such as hooks and breakpoints to catch the execution of `main` and `exit`. In Drakvuf, we place a breakpoint in the `do_syscall` kernel function to trap system

```
struct dbi_test {
    std::string test_name;
    std::string program_name;
    std::vector<std::string> program_args;
    unsigned long main_addr;
    unsigned char memory_mark[20];
    unsigned long exit_addr;
    unsigned long watched_addr;
    unsigned long watched_inst;
};

dbi_test perlbench_1 = {"perlbench_1",
    "perlbench_base.gcc43-64bit",
    {"-I/lib",
        "diffmail.pl",
        "2", "550", "15", "24", "23", "100"},
    0x555555563030,
    {0xf3, 0x0f, 0x1e, 0xfa, 0x41, 0x55, 0x49, 0x89, 0xf5,
        0x41, 0x54, 0x41, 0x89, 0xfc, 0x48, 0x83, 0xec,
        0x08, 0x80, 0x3d},
    0x7ffff7cb6bc0,
    0x55555567da00,
    0x5555555c4240};
```

Figure 4.1: Example configuration data for the SPEC CPU2006 benchmark perlbench_1.

calls. This allows us to watch the start and end of the specific processes. To this end, we only need to place a breakpoint in `main` when it started. To identify the starting point in PANDA, we use the instruction translation callback to only instrument instructions whose program counter matches the specific `main` address. Whenever it is executed, we compare the next 20 bytes from `main` to prevent misidentifying the start. To identify the ending point, we look into TCG blocks before host code generation to find syscalls. In the case that a syscall is found (i.e., opcode 0F 05), we insert a function call to verify whether it is an actual exit called by our process (i.e., 0x3c or 0xe7 syscall number stored in RAX register). Identification in Bochs is conceptually similar to PANDA. Before instruction execution, we verify both if the program counter and the following 20 bytes match for the starting point, and whether the current instruction is an exit syscall for the ending point. Unlike PANDA, callbacks cannot be disabled and there is no less intrusive method. Figure 4.1 shows the configuration data used in our framework for the `perlbench_1` benchmark as an example.

We obtain diverse information about the process from each execution. Once the launcher is notified, it retrieves wall-clock time from a system-wide real-time clock and cpu-time from a high-resolution per-process timer from the CPU. In addition, the launcher creates a copy of `/proc/pid/stat` and `/proc/pid/status`. When the process ends, it computes the difference between the starting and ending points.

For process-level, we use cpu-time as a reference time for our stats, except for Dyninst and GDB, which are executed as a separate process (i.e, a tracer process that controls the execution of a child process). In these two cases we consider a combination of the time spent by the child process (monitored process) and the cpu-time of the process corresponding to the instrumentation tool. Additionally, we obtained the event count for the instrumented primitive. The event is defined by the executed experiment, e.g., for *exec_all*: the total number of executed instructions, for *rw_single*: the access count to the specified memory address, and so on.

For whole-system, we recorded wall-clock time from process start until the end. We found that this is the most consistent way to evaluate the approaches under the same conditions, considering that some of them run emulated clocks inside the guest, while others may use the host's clock. Some run as user-mode (e.g., emulators), while others run partially inside a VM, partially in the hypervisor and partially in the host.

We conducted all the experiments on an Intel Core i7-8700 machine with 16GB of RAM, running Ubuntu 20.04 with kernel version 5.8.0-63. For the guest machine, we used an Ubuntu 20.04 cloud image with kernel version 5.4.0-54 and 4GB of RAM. We also disabled ASLR, kASLR and cpu mitigations in both host and guest machines.

4.5.5 Process-level Techniques Results

Table 4.6 shows the raw results of our process-level experiments for Pin, GDB software breakpoints (*gdb_sb*), GDB hardware breakpoints (*gdb_hb*), GDB single-stepping (*gdb_ss*), GDB page faults (*gdb_pf*), and Dyninst. For every combination of experiment, tool, benchmark, and type of instrumentation, we show the average and standard deviation.

Figure 4.2 shows the base performance overhead of each process-level tool. Debugging-based tools use operating-system provided mechanisms to attach to the debugged process. When no instrumentation is done, the overhead of this kind of tools is minimal or nonexistent. Trampoline based tools install trampolines — or code detours — in a given address, gener-

Table 4.6: Average and standard deviation execution times for process-level techniques implementations.

		perlbench_1	perlbench_2	perlbench_3	perlbench_4	perlbench_5	bzip2_1	bzip2_2	bzip2_3
exec_single	pin	5.69±0.14	4.51±0.16	4.64±0.14	13.11±0.1	0.86±0.02	3.15±0.04	3.14±0.04	24.37±0.16
	dyninst	1.85±0.05	1.55±0.04	1.71±0.04	4.6±0.04	0.23±0.04	2.37±0.05	2.34±0.06	18.8±0.08
	gdb_hb	102.49±1.02	124.21±0.77	110.31±0.42	38.69±0.33	7.73±0.07	4.89±0.04	4.19±0.03	37.06±0.15
	gdb_sb	99.86±0.57	120.3±0.89	107.32±0.66	38.45±0.13	7.55±0.05	4.94±0.03	4.28±0.03	37.6±0.15
	gdb_pf	255.73±0.66	254.9±0.35	269.65±1.98	257.33±2.32	269.03±1.19	220.4±1.38	222.92±1.03	220.11±0.88
exec_range	pin	5.71±0.17	4.51±0.04	4.8±0.16	13.0±0.18	0.86±0.03	3.11±0.02	3.15±0.04	24.8±0.26
	gdb_pf	250.52±1.07	249.57±1.2	265.72±0.98	250.86±2.17	263.4±0.69	215.4±1.39	219.16±0.52	215.32±0.85
exec_all	pin	28.92±0.12	15.57±0.09	24.67±0.13	77.94±0.47	2.71±0.0	23.67±0.07	27.93±0.02	167.46±0.04
	gdb_ss	252.66±1.92	248.18±1.69	249.27±2.03	252.94±1.18	249.46±1.35	197.45±0.79	197.12±1.47	192.86±1.74
rw_single	pin	39.72±0.22	29.33±0.05	36.66±1.32	87.36±0.04	4.03±0.01	23.84±0.0	26.29±0.02	167.4±0.46
	gdb_hb	94.76±0.68	26.53±0.16	1.89±0.0	150.84±0.53	0.69±0.0	28.28±0.11	15.09±0.13	224.19±0.95
	gdb_pf	268.85±1.03	268.55±1.14	284.35±0.89	266.91±4.38	282.37±1.13	222.73±12.67	218.72±1.36	215.54±2.58
rw_range	pin	42.43±0.03	31.68±0.04	38.86±0.07	97.29±0.02	4.34±0.01	27.15±0.0	30.32±0.01	191.9±0.2
	gdb_pf	262.64±0.26	262.83±0.83	277.98±1.62	262.6±0.66	275.79±3.2	211.53±3.09	211.47±1.83	203.41±6.77
rw_all	pin	15.88±0.04	8.13±0.03	13.16±0.03	118.05±0.52	1.82±0.02	18.32±0.01	17.73±0.01	128.25±0.17
none	pin	5.33±0.02	4.15±0.02	4.43±0.05	12.19±0.02	0.79±0.0	3.02±0.01	2.96±0.0	23.04±0.04
	dyninst	1.79±0.01	1.48±0.01	1.65±0.0	4.58±0.01	0.16±0.01	2.3±0.01	2.28±0.02	18.76±0.12
	gdb	1.8±0.0	1.52±0.02	1.66±0.01	4.57±0.0	0.16±0.01	2.32±0.01	2.3±0.02	18.76±0.09

Table 4.6: Average and standard deviation execution times for process-level techniques implementations (*Continued*).

		gcc	mcf	namd	dealii	povray	libquantum	lbm	sphinx3
exec_single	pin	5.16±0.01	7.65±0.12	7.0±0.06	22.62±0.2	7.08±0.14	1.38±0.02	16.05±0.11	5.85±0.02
	dyninst	0.45±0.06	6.98±0.11	6.36±0.05	12.19±0.06	3.04±0.04	0.87±0.05	14.99±0.06	3.96±0.06
	gdb_hb	3.41±0.02	54.99±0.3	8.77±0.02	57.38±0.14	9.0±0.05	5.72±0.05	29.3±0.13	74.43±0.43
	gdb_sb	3.4±0.04	54.06±0.47	9.06±0.04	57.43±0.37	8.98±0.1	5.73±0.06	29.17±0.18	73.68±0.34
	gdb_pf	227.25±1.1	220.29±1.38	271.5±1.25	417.03±2.16	294.43±0.94	252.77±0.78	252.13±1.14	265.91±1.66
exec_range	pin	5.33±0.05	18.13±0.04	10.12±0.07	59.26±0.15	6.95±0.17	1.04±0.02	113.93±0.61	5.94±0.09
	gdb_pf	222.39±0.78	216.49±0.46	266.43±0.93	411.65±1.1	289.37±1.13	248.0±1.21	248.15±0.73	260.08±1.22
exec_all	pin	10.46±0.03	25.92±0.15	73.35±0.22	162.7±0.4	40.92±0.02	14.07±0.0	113.93±0.25	50.7±0.12
	gdb_ss	287.93±1.73	220.41±0.83	231.83±1.01	304.51±3.64	249.28±0.83	193.99±0.85	187.94±1.55	263.23±1.52
rw_single	pin	62.28±0.12	24.56±0.04	54.56±0.01	165.27±0.08	57.12±0.16	11.22±0.01	89.77±0.01	81.29±0.01
	gdb_hb	4.52±0.05	42.41±0.25	8.86±0.03	59.31±0.36	4.4±0.03	4.58±0.04	16.58±0.63	59.48±0.27
	gdb_pf	245.02±1.59	220.2±0.85	275.6±1.41	373.42±1.01	313.81±1.64	251.35±1.28	19.44±0.43	269.07±1.35
rw_range	pin	62.8±0.02	28.92±0.03	62.53±0.01	192.46±0.36	64.15±0.28	13.05±0.0	102.7±0.02	87.56±0.12
	gdb_pf	239.24±1.12	214.24±1.46	262.57±1.84	374.95±1.13	283.52±2.17	246.96±0.69	19.47±0.48	263.37±0.79
rw_all	pin	59.54±0.02	12.19±0.1	20.59±0.09	67.29±0.03	31.67±0.01	5.16±0.0	36.98±0.09	63.51±0.35
none	pin	4.93±0.02	7.22±0.02	6.75±0.02	21.56±0.03	6.73±0.02	0.97±0.01	15.23±0.09	5.11±0.0
	dyninst	0.41±0.01	6.95±0.16	6.34±0.01	12.05±0.04	2.95±0.02	0.78±0.01	15.02±0.15	3.82±0.01
	gdb	0.4±0.0	6.84±0.04	6.34±0.04	12.08±0.03	2.96±0.02	0.77±0.01	14.87±0.05	3.85±0.02

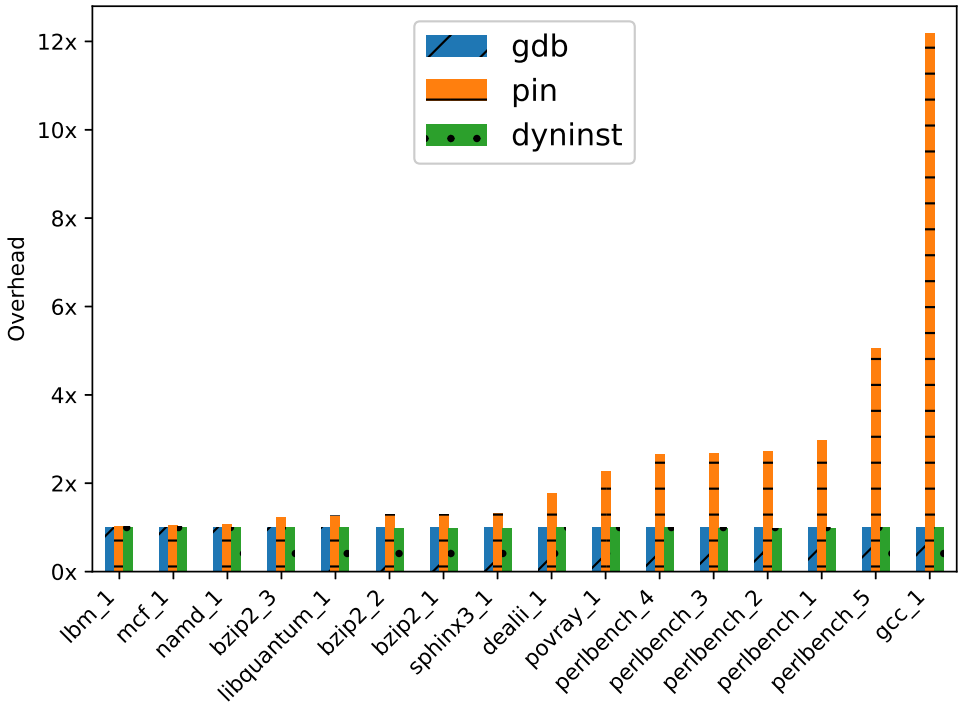


Figure 4.2: Base overhead of process-level DBI systems.

ating very little overhead as well. In contrast, dynamic compilation tools such as Pin, need to read and analyze the original code to give the opportunity to insert instrumentation routines before compilation. This process implies an overhead of up to 12 times the reference execution time on the native system even when no instrumentation is performed. Figure 4.2 also shows how the overhead remains constant for GDB and Dyninst, which execute the code natively, while some programs have different overheads for Pin. The reason is that the nature of the program and its execution also affect JIT-DBT, which may increase the time spent in branch resolution or register allocation [UCY+06]. In this case, instruction footprint and execution time are responsible for the increased overhead [LCM+05], especially in gcc benchmark. Having a large instruction footprint and a short execution time results in insufficient code reuse to compensate the compilation cost.

Even though binary rewriting inflicts a significant overhead with respect to native execution, this might be tolerable under certain circumstances. For instance, debuggers use software and hardware breakpoints or

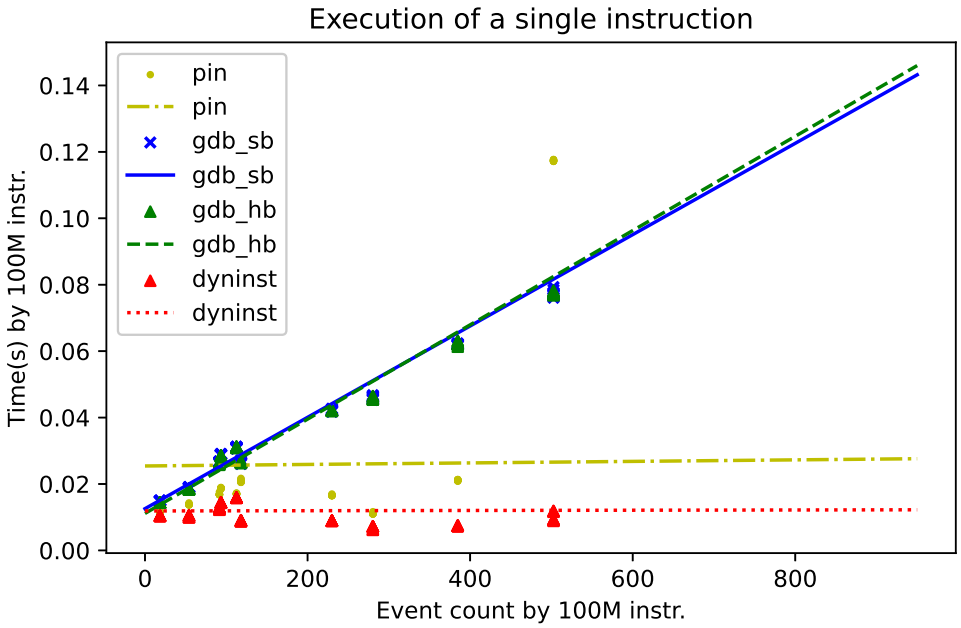


Figure 4.3: Time required to execute 100 million instructions by number of single instruction execution events. Trend lines represent the linear regression of the plotted points.

page faults to trigger interrupts. Every method that triggers an interrupt is followed by a context switch. The OS will analyze that interrupt and redirect the execution to the corresponding handler. This process implies the execution of many operations (e.g., saving the interrupted process's context, context switching, transitioning from Ring 3 to Ring 0, and back to Ring 3). Hence, while the number of instrumented events increases, the overhead of JIT-DBT techniques is compensated by how efficiently those approaches can instrument the code (e.g., no context-switch is needed).

Figure 4.3 shows the normalized time required to execute a single instrumented instruction for several programs with a number of instrumentation events (i.e., the selected instruction is executed a different number of times in each benchmark program). The X axis shows the number of instrumented events every 100M instructions, whereas the Y axis shows the time to execute those 100M instructions. Despite the high variability in the case of Pin (discussed previously), all the cases show a linear overhead growth as the number of instrumented events per 100M instructions increases. However, the growth rate of GDB is much higher and exceeds

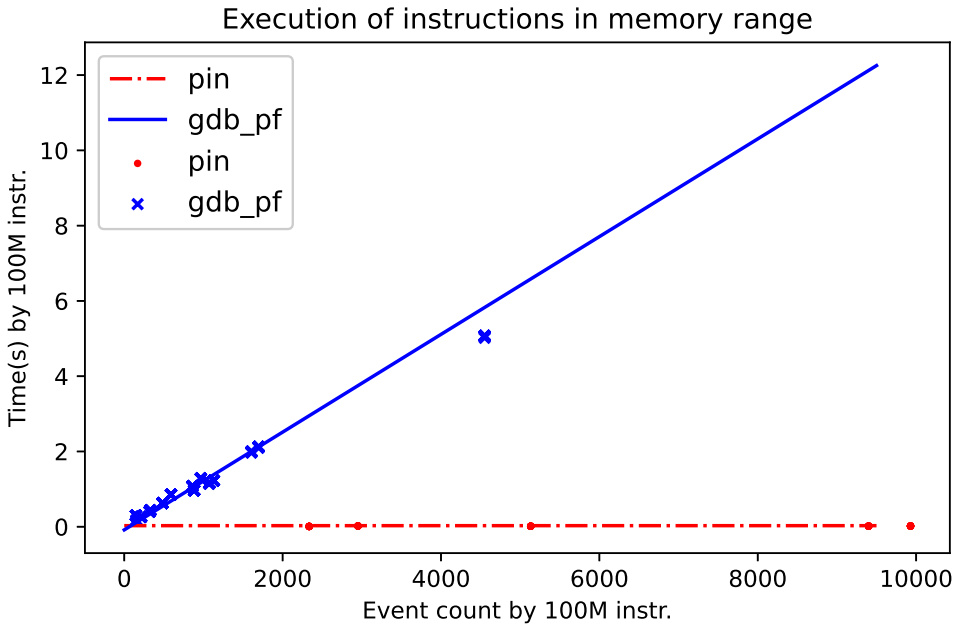


Figure 4.4: Time required to execute 100 million instructions by number of instruction executions in a memory range. Trend lines represent the linear regression of the plotted points.

the overhead of Pin around 100 events per 100M instructions. Despite this number is indicative and depends on many factors, Pin clearly outperforms GDB as soon as it needs to instrument a few events. Dyninst performs even better than Pin as it benefits from both not needing to switch context and not needing to implement binary rewriting.

Figure 4.4 shows the execution time for every 100M instructions for programs with a different number of instrumentation events in a memory range (i.e., a memory page) per 100M instructions. We observe that while the execution time for page-fault-based instrumentation in GDB grows linearly with the number of instrumented events, Pin remains almost constant. In this case, Pin decides which instructions ought to be instrumented at translation time, adding very little overhead since it only inserts instrumentation callbacks at instructions that are part of the target page.

Regarding the overhead imposed by GDB single-stepping when instrumenting the execution of all instructions (shown in Table 4.6), we can notice that execution time is extremely high compared to Pin. In fact, as aforementioned, we had to limit GDB to 20M instrumented events. If we

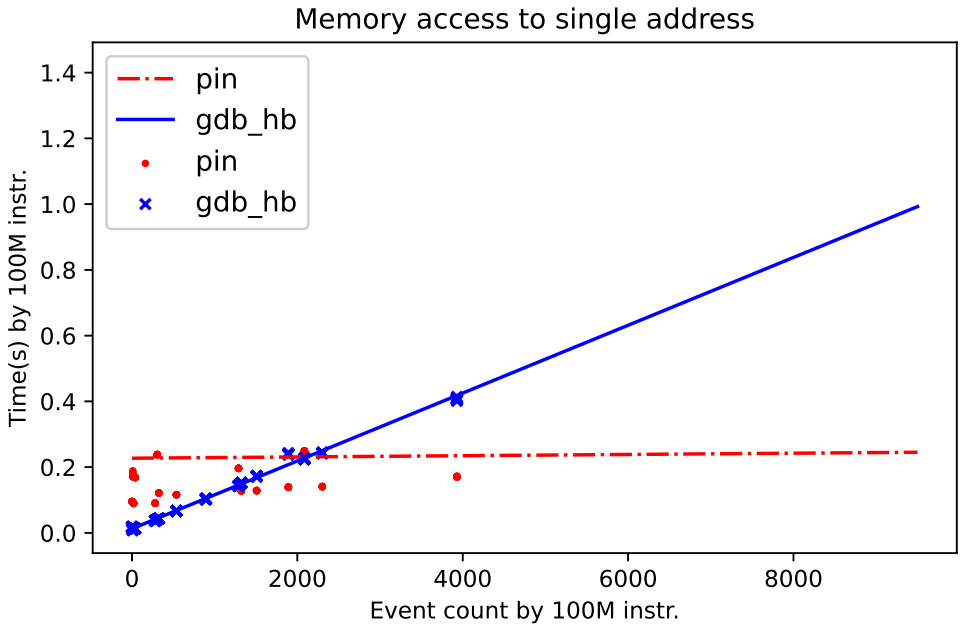


Figure 4.5: Execution time for 100 million instructions by number of access events per 100 million instructions. Trend lines represent the linear regression of the plotted points.

compare the normalized instrumentation time (total time minus the time required to execute the program with no instrumentation) for 20M instrumented events, GDB single-stepping is more than 100,000 times slower than Pin at instruction tracing. Similar patterns can be noticed for page-fault-based GDB. In that case, we had to limit GDB to track up to 200,000 events.

As we detailed in previous sections, Pin has to check every memory operation on every instruction at run-time to decide if it affects a target memory range or a memory address.

Figure 4.5 shows the time required to execute 100M instructions for our benchmark programs that trigger a different number of address access events per 100M instructions. Even if hardware breakpoints are very efficient to instrument accesses to specific memory addresses, and Pin needs to instrument every instruction, its overhead is compensated as soon as the number of events triggered is higher than 2,000 every 100M instructions. This case is particularly interesting because the *cutting point* after which Pin performs better than debugger-technique-based approaches is higher

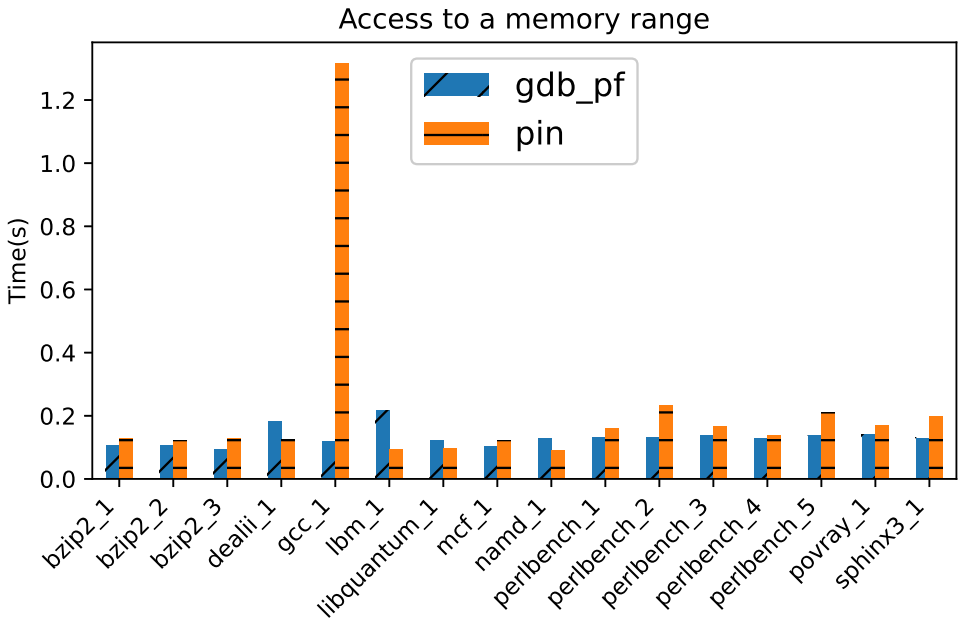


Figure 4.6: Overhead of process-level DBI systems for memory range read-/write.

than in the cases shown previously (e.g., single instrumented instruction execution).

Figure 4.6 shows the time required by GDB to instrument 100 page accesses using page faults for the different benchmark programs, and the time it takes Pin to check the target address for every memory operation in 100M instructions. This overhead in Pin depends not only on the number of instructions, but also on the type of operations and number of memory accesses of these operations (and thus there is some variability in Figure 4.6). However, we can see that Pin significantly outperforms GDB except for very specific set-ups where memory accesses are very frequent and the target memory page or target memory address will be accessed extremely infrequently. This can be noticed in Table 4.6.

While using page faults to monitor accesses to a single memory address (which is the same as that of accesses to a memory range) is significantly slower than Pin and hardware breakpoint based approaches, this difference is considerably smaller for the `lbm` benchmark, for which we instrumented an address that is accessed very infrequently (i.e., 1,312 times).

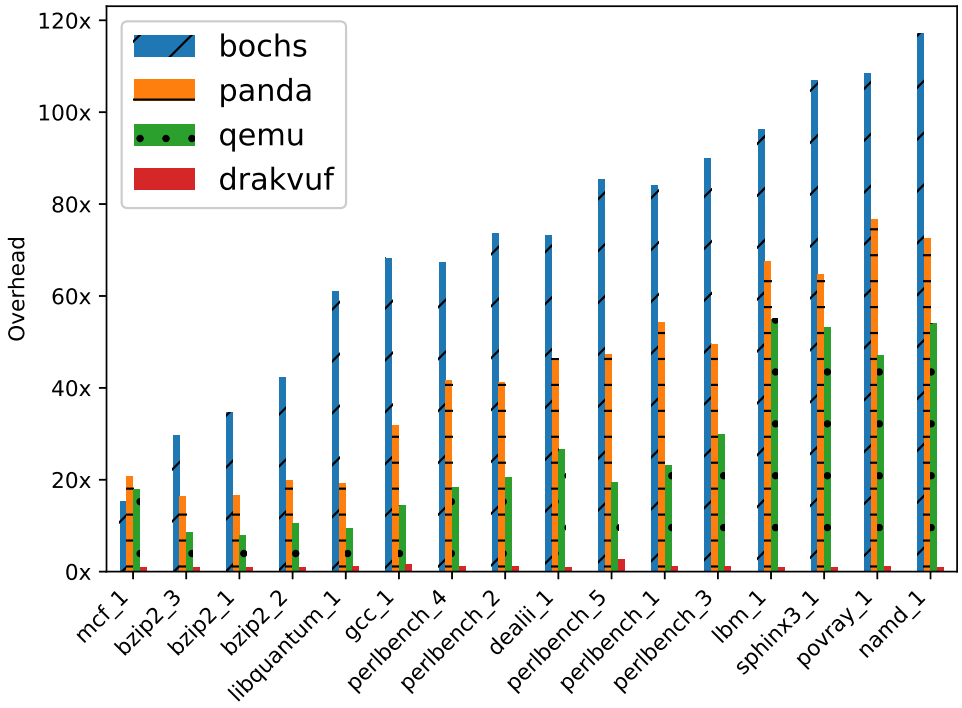


Figure 4.7: Base overhead of each whole-system framework.

4.5.6 Whole-System Techniques Results

Table 4.7 shows the raw results of our measurements on whole-system instrumentation techniques for each primitive. In all cases, we report the average and standard deviation for each iteration. Note that we have included a new alias for PANDA called `panda_osi`. This alias represents a specific configuration of the PANDA framework in which Operating System Introspection (OSI) plugins are enabled. These plugins build and provide semantic information about the underlying operating system for introspection purposes (e.g., task switch callbacks, methods to retrieve the list of running processes, etc.). Unfortunately, we observed that it introduces excessive performance overhead. Therefore, we only include these results in Table 4.7 as reference.

Figure 4.7 shows the base overhead of PANDA, Bochs and Drakvuf with no instrumentation applied with respect to native execution. We also include QEMU 2.9.1 (the version used in PANDA at the time of the experimentation) as reference. We can notice that Drakvuf imposes a negligible and almost constant overhead, since hypervisor-based systems execute

Table 4.7: Average and standard deviation execution times for whole-system techniques implementations.

		perlbench_1	perlbench_2	perlbench_3	perlbench_4	perlbench_5	bzip2_1	bzip2_2	bzip2_3
exec_single	bochs	233.54±5.21	147.7±0.47	215.29±1.43	517.84±10.59	18.81±0.11	134.72±0.45	160.86±0.29	959.99±8.38
	panda_osi	811.66±13.32	498.99±8.26	768.83±11.22	2041.15±503.95	66.34±2.44	451.18±21.87	504.01±24.97	3160.95±169.56
	panda	97.51±0.69	62.01±0.64	81.87±0.52	189.37±0.88	7.21±0.06	39.37±0.48	46.31±0.4	308.86±0.93
	drakvuf_sb	43.33±0.96	50.64±0.44	45.23±0.81	19.17±0.36	3.77±0.2	3.72±0.08	3.4±0.08	26.37±0.24
	drakvuf_pf	11.88±0.25	11.54±0.22	11.53±0.39	11.6±0.4	11.12±0.3	11.71±0.28	11.76±0.47	11.7±0.39
exec_range	bochs	247.41±2.38	158.65±0.4	224.04±0.66	545.88±40.03	19.82±0.09	141.93±0.56	168.75±0.29	999.32±1.54
	panda_osi	1159.94±382.73	812.07±295.42	934.3±147.83	2404.91±655.39	75.6±6.56	434.29±5.37	516.69±11.42	3212.68±143.4
	panda	98.22±0.94	66.69±0.93	83.73±0.65	193.62±1.16	7.31±0.08	38.54±0.27	45.42±0.2	305.21±1.21
	drakvuf_pf	11.87±0.19	11.52±0.26	11.55±0.15	12.07±0.42	11.51±0.27	11.7±0.3	11.76±0.47	11.7±0.39
exec_all	bochs	274.23±90.12	148.81±0.31	216.38±0.48	522.18±8.8	18.96±0.08	140.6±0.22	166.22±0.45	985.46±6.92
	panda_osi	970.89±56.5	586.13±16.48	911.09±29.72	2082.57±45.58	71.89±0.54	507.0±15.81	609.23±30.39	4297.66±1007.03
	panda	359.66±3.95	234.80±0.6	353.85±0.64	682.9±8.7	29.21±0.21	148.37±0.81	165.62±0.29	1058.66±3.45
rw_single	bochs	241.88±1.39	152.37±0.33	224.56±0.59	530.32±0.77	19.39±0.32	139.81±0.37	167.54±0.47	982.88±1.89
	panda_osi	859.25±26.34	540.37±4.63	830.29±23.26	2128.34±604.6	66.7±0.35	457.53±7.31	557.48±3.39	3496.82±697.94
	panda	359.13±1.44	240.61±0.68	366.06±3.25	676.34±4.43	28.88±0.22	161.53±1.47	187.1±1.63	1154.18±15.47
rw_range	bochs	293.06±81.98	162.74±0.47	231.22±0.67	538.98±1.15	19.72±0.09	145.45±1.18	174.2±0.89	1028.29±5.49
	panda_osi	670.02±24.81	388.34±15.24	618.27±26.04	1808.19±19.57	57.1±0.46	518.21±7.14	619.85±9.2	3439.41±137.5
	panda	359.5±1.07	237.98±1.31	360.21±0.46	676.15±0.78	28.59±0.06	160.49±0.33	189.94±7.3	1150.98±3.32
	drakvuf_pf	11.68±0.18	11.41±0.2	11.56±0.15	11.6±0.23	11.65±0.3	11.7±0.32	11.53±0.24	11.49±0.24
rw_all	bochs	250.55±1.05	156.36±0.76	248.28±5.61	558.71±4.85	20.06±0.18	148.66±0.26	176.86±0.33	1047.44±5.81
	panda_osi	655.36±6.21	391.51±14.9	591.38±8.44	1693.64±66.2	52.09±0.27	804.56±248.31	829.73±284.41	3582.07±108.79
	panda	351.4±1.08	233.37±0.6	357.57±3.47	673.52±8.34	28.85±0.21	159.09±0.5	185.54±0.32	1086.66±25.52
none	bochs	151.03±1.44	112.08±0.58	149.17±0.79	308.13±0.79	13.23±0.15	80.23±0.52	97.4±3.72	558.03±3.59
	panda_osi	614.03±44.48	361.46±24.63	501.81±7.85	1253.37±19.01	42.56±0.4	326.18±1.05	392.37±1.79	2912.61±1063.15
	panda	97.26±1.1	62.73±0.65	82.08±0.91	190.05±1.05	7.32±0.06	38.56±0.21	45.95±0.61	307.39±1.38
	drakvuf	2.2±0.02	1.79±0.01	1.91±0.02	5.05±0.09	0.41±0.01	2.47±0.03	2.44±0.03	19.11±1.95

Table 4.7: Average and standard deviation execution times for whole-system techniques implementations (*Continued*).

		gcc	mcf	namd	dealii	povray	libquantum	lbm	sphinx3
exec_single	bochs	38.05±0.31	169.9±1.36	962.24±2.91	1346.45±7.27	442.25±1.56	81.57±0.61	1770.39±3.19	541.87±1.29
	panda_osi	239.24±70.61	833.76±281.54	1643.09±499.79	3300.07±106.58	1614.5±528.31	306.83±13.16	2495.79±64.54	1242.79±407.72
	panda	12.7±0.16	141.11±1.01	473.89±1.15	576.05±1.35	230.75±0.91	15.05±0.22	1086.19±154.73	252.08±2.13
	drakvuf_sb	2.15±0.11	28.58±0.67	7.61±0.05	30.68±0.35	6.7±0.17	3.1±0.06	21.04±0.25	32.92±0.79
	drakvuf_pf	11.3±0.44	11.54±0.43	12.18±0.23	11.46±0.22	12.19±0.3	12.21±0.27	15.75±0.4	12.01±0.23
exec_range	bochs	38.84±0.14	171.69±0.26	988.18±0.77	1289.17±214.01	462.76±0.6	85.4±0.2	1856.45±1.66	570.91±1.14
	panda_osi	125.97±4.98	613.76±24.17	1295.32±12.41	3345.89±524.51	1378.1±424.55	345.09±105.33	2500.02±40.55	939.53±2.16
	panda	13.77±0.15	211.56±1.15	493.02±2.23	816.7±6.92	222.99±0.24	14.63±0.03	2399.87±168.04	248.8±0.21
	drakvuf_pf	11.52±0.25	11.41±0.17	12.83±0.24	11.47±0.21	12.18±0.27	12.07±0.26	15.68±0.4	12.14±0.16
exec_all	bochs	37.21±0.09	167.29±0.77	961.83±4.06	1381.23±3.33	451.63±1.34	84.93±0.3	1817.49±3.03	563.86±0.86
	panda_osi	145.45±4.45	713.11±23.79	1403.02±30.41	3774.97±656.91	1764.57±576.8	322.3±12.79	2525.22±8.54	989.88±22.21
	panda	45.44±3.93	239.78±0.52	933.63±6.33	1350.81±12.15	759.08±252.91	70.13±0.29	2318.74±29.7	498.34±6.47
rw_single	bochs	39.8±0.22	170.8±0.66	960.09±1.29	1358.85±1.83	455.34±1.17	82.13±0.28	1847.86±1.87	546.95±1.03
	panda_osi	125.36±8.43	671.27±9.52	1472.4±522.37	3347.43±423.37	1522.71±495.04	385.76±137.86	2079.27±42.17	911.22±32.49
	panda	40.45±0.16	275.58±0.64	790.13±4.19	1557.24±7.28	620.89±0.72	72.31±0.21	1591.37±26.92	524.35±150.65
rw_range	bochs	40.59±0.31	175.62±1.07	980.12±2.47	1417.88±7.79	469.97±0.75	84.22±0.2	1870.98±2.9	552.04±0.97
	panda_osi	281.11±14.03	1051.52±343.85	1160.09±3.98	3835.01±1054.01	1266.22±51.01	301.85±2.07	2009.74±58.65	968.57±37.34
	panda	40.13±0.2	274.51±0.84	829.76±116.76	1667.16±272.88	625.65±8.26	73.33±1.22	1665.82±27.52	457.44±1.19
	drakvuf_pf	11.59±0.2	11.63±0.28	12.42±0.19	12.56±0.31	12.18±0.27	12.18±0.09	15.58±0.04	12.67±0.13
rw_all	bochs	40.63±0.09	178.44±0.5	988.54±0.88	1441.05±28.26	474.37±3.88	87.7±0.37	1877.04±4.07	564.31±1.13
	panda_osi	120.3±2.66	653.98±10.96	1179.1±6.69	3824.59±973.34	1210.89±29.63	227.47±14.07	2048.07±27.38	879.66±7.61
	panda	42.33±7.94	272.55±0.56	778.57±1.7	1656.88±295.07	618.32±7.8	70.92±0.15	1643.76±11.03	459.34±8.57
none	bochs	27.53±0.37	105.06±0.49	742.31±4.8	885.86±4.55	320.21±0.86	46.96±0.42	1432.52±2.2	411.81±9.91
	panda_osi	90.94±1.08	511.61±5.5	876.97±32.35	2422.83±130.27	742.78±4.42	202.76±2.8	1488.01±363.11	1062.98±328.15
	panda	12.86±0.18	142.34±1.7	460.18±0.56	559.14±2.44	226.39±0.34	14.72±0.09	1003.73±0.6	249.56±1.6
	drakvuf	0.67±0.01	7.42±0.06	6.53±0.03	12.48±0.05	3.79±0.04	0.96±0.01	14.96±0.28	4.16±0.06

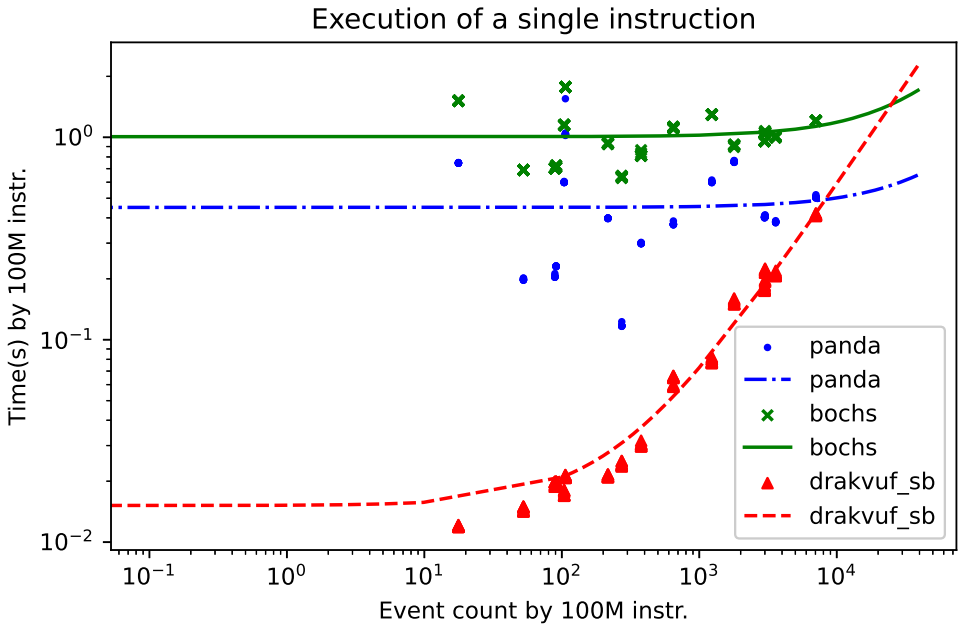


Figure 4.8: Execution time for 100M instructions by number of instruction execution events per 100M, in log scale. Trend lines represent the linear regression of the plotted points.

the guest natively in the CPU. In the case of emulators, the overhead is much higher and varies significantly, with the highest values found for the floating-point intensive benchmarks. Complex instruction set emulation is a non-trivial task, and its emulation can inflict varying overheads depending on the complexity of the operations performed by the program, the kind and frequency of memory accesses, and optimizations. Overall, Bochs has higher overhead than PANDA and ranges from 20 to 120 times the native execution time. It is worth to note that the mcf benchmark has the highest number of cache misses, and intensive pointer and integer arithmetic.

Figure 4.8 shows the time required to execute 100M instructions at different instrumentation frequencies for the single instruction instrumentation experiment. The results are plotted in logarithmic scale. We notice a particular point in which Drakvuf becomes slower than PANDA and Bochs: when the number of events per 100M instruction grows to 100,000. Similarly to process-level instrumentation, the overhead of context switches — and in this case, VM exit operations — has a non-negligible cost. This cost can compensate emulation overhead after a certain point. There is also

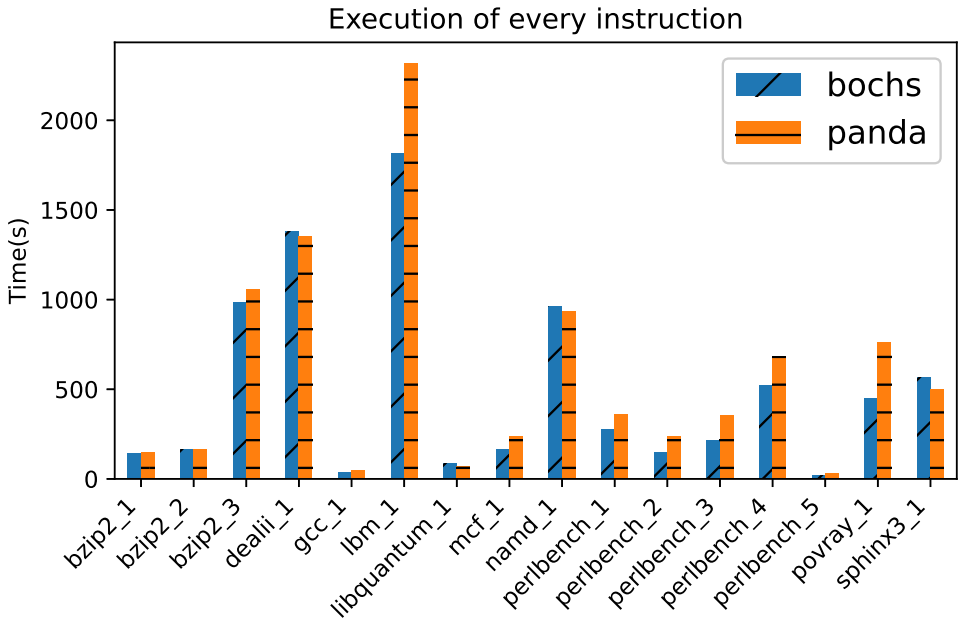


Figure 4.9: Execution time for instrumenting every instruction in each benchmark program for whole-system tools.

more variability in execution times for Bochs and Panda due to the different base overhead imposed by different programs (shown in Figure 4.7).

Figure 4.9, Figure 4.10, and Figure 4.11 show the execution times for the following primitives: execution of every instruction, memory accesses to an address, and accesses to every memory address respectively. In all cases, we observe similar performances for Bochs and PANDA, which emulate memory in software. The superior overhead imposed by Bochs in a non-instrumented execution is often compensated, but the results depend on the type of program being instrumented. We can claim that results for both tools in their current state suggest that the techniques at instruction-level granularity are comparable, and that their relative overhead is subject to specific implementation details and optimizations.

Figure 4.12 and Figure 4.13 show the normalized execution time (execution time per 100M executed program instructions) for different instrumentation event counts for the following experiments respectively: memory accesses to a memory range (i.e., a memory page), and instruction execution in a given memory range. While increased execution time remains constant for most programs in Drakvuf, the variability persists in Bochs

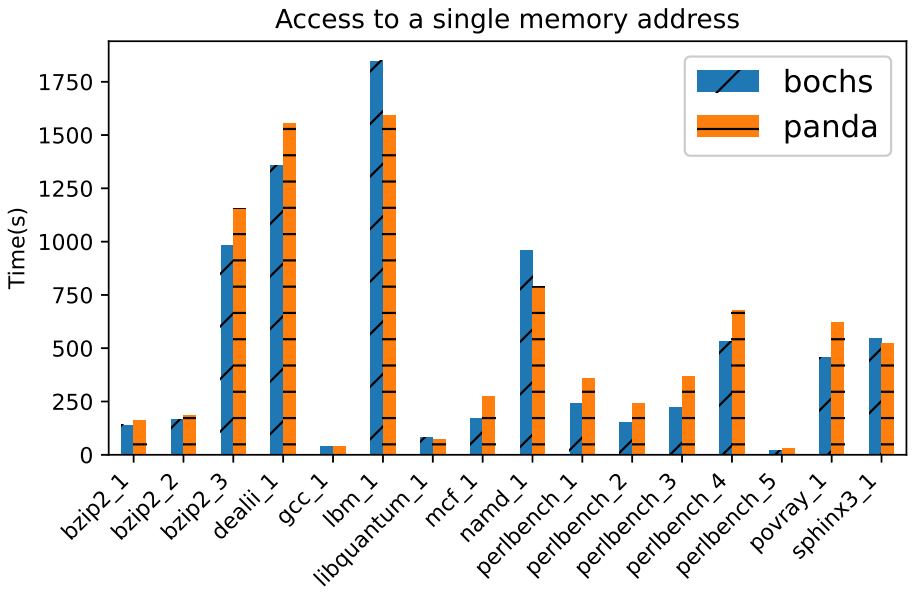


Figure 4.10: Execution time for instrumented accesses to a memory address in benchmark programs for whole-system.

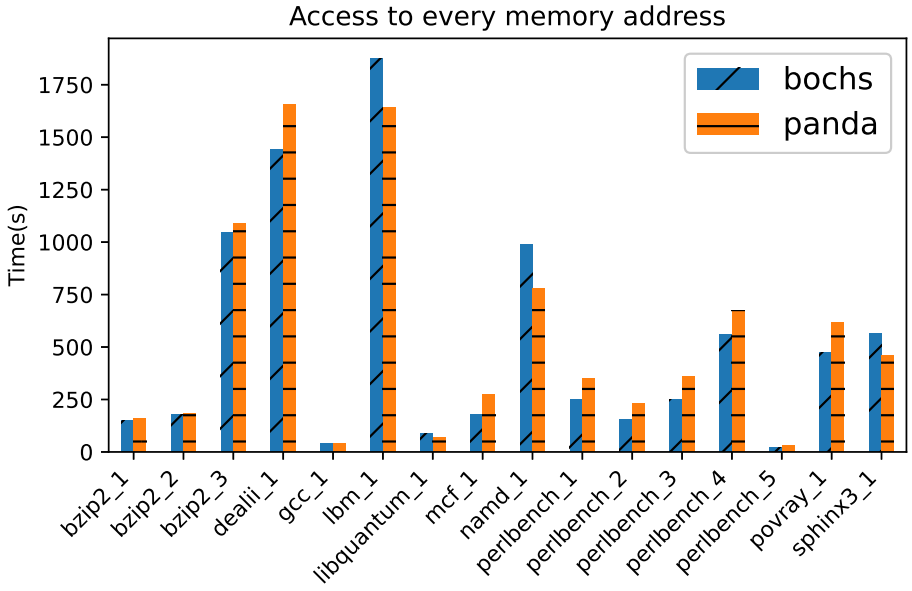


Figure 4.11: Execution time for every memory access instrumentation in benchmark programs for whole-system tools.

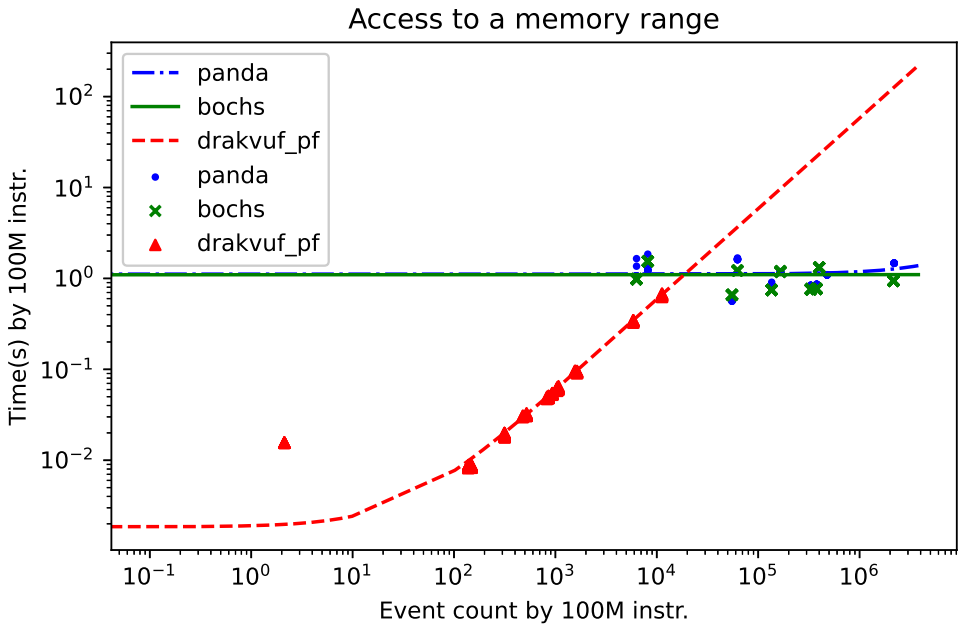


Figure 4.12: Execution time for 100M instructions by number of memory events per 100M instructions, in log scale. Trend lines represent the linear regression of the plotted points.

and PANDA as in previous cases. The axes are represented in logarithmic scale to improve their legibility. As expected, there is a *cutting point* beyond which the execution overhead of Bochs and PANDA is compensated due to the penalty imposed by the slower VM switches needed by Drakvuf.

4.6 Discussion

Several papers have covered DBI in the past, focusing on the transparency of specific approaches in adversarial environments rather than their foundations, convenience, capabilities and efficiency. The ultimate selection of a DBI framework for a specific task is determined by the expressiveness, transparency, and performance properties. Existing approaches make trade-offs in some of these properties in exchange for enhancements in others. There are further aspects that also play a role in the process of choosing a tool, such as platform and architecture compatibility, and ease of use. However, these aspects are purely implementation dependent and not related to the instrumentation techniques.

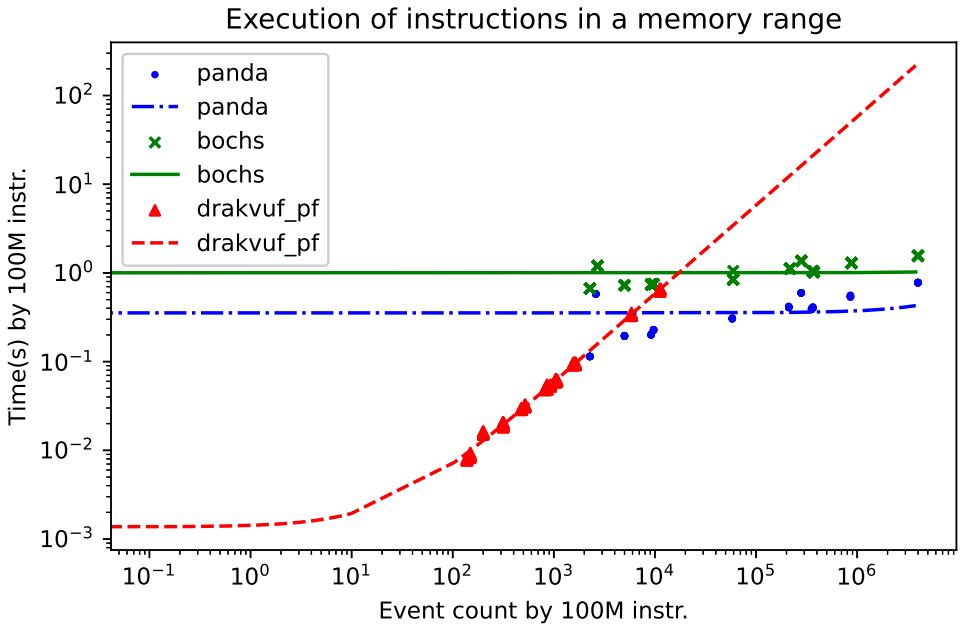


Figure 4.13: Execution time for 100M instructions by number of instruction execution events per 100M, in log scale. Trend lines represent the linear regression of the plotted points.

In this chapter, we made an additional effort to isolate instrumentation techniques from the tools that implement them, but the results show that implementation details have certain influence on the performance. For instance, tools such as GDB inevitably add some overhead to implement instrumentation from having to notify the debugger and so on. This means that some of the *cutting points* that we show must be interpreted as guidelines and not as absolute numbers. However, even though there may be some variability in the results due to specific tool and environment features, we believe that the conclusions presented would not change significantly.

In some experiments, the overhead was so high that they would have required an unaffordable and impractical amount of time. In these cases, we restricted the number of events to monitor to 20M in the case of single-stepping, and 200K in the case of page faults. To present our results, we have normalized the overhead as needed in order to compare approaches under the same conditions. We believe that these limits are high enough to have statistically representative data. Finally, we do not cover specific

instrumentation use cases, but the actual primitives that those use cases may rely on.

Hardware support in DBI. Hardware features (e.g., shadow page tables, EPT, debug registers, PMU) are strongly used and convenient in DBI, and usually guide the development of new DBI frameworks. For instance, the PowerPC architecture supports ranged breakpoints (apart from individual addresses). However, these are not supported on x86, which impedes existing and new DBI tools from taking advantage of them to efficiently implement range primitives. Similarly, some Intel processors include a feature called Memory Protection Keys (MPK) that enable fast memory protection management at page-level [mpk22]. We have observed that page-fault-based instrumentation is not practical in most scenarios due to its performance, but features such as MPK enable more efficient and functional DBI (i.e., because it does not require modification of the page tables). At the hypervisor-level, most frameworks rely on VMEXIT to implement all instrumentation logic. Nevertheless, VMFUNC is another Intel primitive that allows to change EPT page tables in a VM without exiting into the hypervisor. This enables and guides the implementation of more efficient hypervisor-based instrumentation [SCL+18, HD21].

Instrumentation technique capabilities. The fact that a given instrumentation technique is able to instrument a specific primitive does not mean that it is the right tool for the job. One clear example of this is instrumentation through page faults. We have observed numerous times in our experiments that page-fault-based instrumentation has a prohibitive overhead, orders of magnitude higher than dynamic binary rewriting and instruction by instruction monitoring. Our results show that there is not a clear winner for every situation. However, it is fundamental to ascertain that the capabilities of a given instrumentation technique match the specific requirements of the problem at hand. It is worth mentioning that JIT-DBT is the only technique that implements coarse-grained primitives (i.e., at block level) by its nature, convenient for tasks such as coverage retrieval in fuzzing. In addition, factors to consider include the number of times that a monitored event is expected to be triggered, and program characteristics that may affect the performance of techniques and their implementations. These characteristics include the amount and type of memory accesses, instruction footprint, execution time, and floating-point operation intensity, as observed during our experimentation.

Emulation overhead variability. Emulation and other binary rewriting

based approaches often need to decompose complex instructions such as the ones in the Intel architecture (CISC) into smaller operations. Hardware processors benefit from co-processors and hardware optimizations to execute these instructions in a fast way, but emulators may introduce varying overheads depending on the types of instructions. We have noticed this pattern in the results for Pin, Bochs, and PANDA. For this reason, it is not only important to consider the kind of instrumentation or the number of events that will be triggered, but also the nature of the program and its base emulation overhead.

Emulation overhead vs instrumentation overhead. As mentioned before, emulators and binary rewriting introduce a non-negligible overhead with respect to native execution or hypervisor-based virtual machines. However, these systems often run their instrumentation and callbacks in the same context and address space as the emulated program. This saves numerous context switches and interrupts and therefore avoids wasting millions of cycles to deal with these events. Furthermore, we have also observed that there is an expected increase in baseline performance overhead for PANDA as compared to its base emulator QEMU. This overhead difference is on average 1.8 times in our experiments, and can be attributed to the infrastructure that enables instrumentation and dynamic analysis via plugins. While emulators like Bochs or PANDA are well known for their high overhead, we have observed that this negative effect is soon compensated when the number of events to instrument is high.

The right tool for the right need. The results obtained highlight the need for selecting the proper instrumentation technique depending on particular needs. When designing a new project or experiment, it is fundamental to consider all the aspects that may influence performance as well as the present and future instrumentation needs. This chapter can serve as a comprehensive guide into the labyrinth of DBI techniques available nowadays.

4.7 Related Work

Prior work on DBI has focused on DBI in specific application domains, transparency properties of DBI systems, or specific instrumentation techniques (e.g., measuring the internal performance of Pin [UCY+06]). Moreover, other works related to ours include papers that analyze an individual building block of DBI or features used by instrumentation techniques.

DBI in Malware Research. Malware research is one of the most pop-

ular application domains of DBI, especially for whole-system approaches [ESKK12, LMP+14]. Several works have been devoted to analyze over a decade of malware analysis evasion techniques that hinder DBI and their countermeasures [SAM14, PCM+17, BY17, ANSB19, DIPQ22], while others are more focused on specific tools [KZBK18]. From a different perspective, previous work has also focused on the definition and fulfillment of transparency requirements for DBI systems [DRSL08, NZ17, DCN+19]. These works analyze specific aspects of DBI and different usability circumstances. Our work is different since we analyze DBI as a whole and fundamental aspects such as instrumentation capabilities.

DBI Elements and Analysis. Jain et al. [JBZ+14] proposed an analysis of introspection capabilities and approaches, describing the different VMI designs, the evolution of the semantic gap problem and proposed solutions, and the various attacks against VMI, as well as defenses, and trust assumptions. Since introspection is a building block of DBI, we draw on this systematization of knowledge to express introspection capabilities of whole-system approaches.

Das et al. [DWA+19] analyzed the usage of performance counters for security purposes, which are a feature used by some instrumentation techniques. They point out the different challenges that influence the correct operation of security approaches that make use of them.

A work close to ours [DCN+19], describes the inner workings and primitives of DBI frameworks, their usage for security research purposes, and discusses the transparency of such systems along with evasion and escape techniques, proposing some mitigations. In contrast to our work, they exclusively take into account JIT-based user-space DBI frameworks, and focus on the transparency issues of DBI, whereas we take a broader approach by analyzing every technique, and focusing on instrumentation capabilities and performance.

4.8 Summary

DBI systems are largely applied in a broad range of areas, especially in security research and in the development of concurrency vulnerability detection approaches. Numerous approaches have been proposed, implementing different techniques and designs with their particular advantages and limitations.

With this chapter we aimed to shed light into the heterogeneous ecosys-

tem of DBI. This work is the first to organize the building blocks and foundations of DBI, and to analyze existing instrumentation techniques. We evaluated their performance based on the practical implementations of these instrumentation methods for both process-level and whole-system, and showed that there is no one-fits-all approach. In fact, we showed that choice of the adequate set of techniques not only is dependent on the instrumented event, but also in the target program's nature and how frequent these events are. We believe that this research will bring a better understanding of the inner workings of DBI, and guide future research to adapt or develop new instrumentation techniques and frameworks.

Chapter 5

Conclusions

Memory errors are still pervasive in current and widely used software. Throughout the years, detection, exploitation and mitigation of memory corruption has been one of the main directions of the research in system security. However, memory errors do not take place exclusively in their traditional form. The role of concurrency makes it possible for them to be triggered in additional unconventional ways. This gives rise to concurrency memory corruption vulnerabilities that standard detection approaches fail to unmask and to which little attention has been paid. Recently proposed approaches heavily rely on instrumentation capabilities, and in Dynamic Binary Instrumentation (DBI) in particular. However, the state-of-the-art of DBI is diffuse and not clearly defined, which hinders the development of new DBI methods and efficient concurrency vulnerability detection approaches.

Troughout this dissertation manuscript, we have described, discussed and evaluated the contributions of the conducted research. In this chapter, we present the overall conclusions drawn from the development of this doctoral research and from each of the individual contributions.

The remainder of this chapter is organized as follows. Section 5.1 reviews the particular conclusions from each contribution and validates the initial hypothesis. Then, Section 5.2 discusses the future lines of work around this research.

5.1 Dissertation Summary

In Chapter 2, we have analyzed the properties of memory errors. We have also gone through the increasingly complex attacks that attackers have built over the years. Similarly, we have analyzed the main memory error detection approaches and defense mechanisms. Despite all the proposed approaches, our analyses of reported memory corruption vulnerabilities strengthen the insight that memory errors have not decreased over the years and remain as the most severe vulnerabilities.

In Chapter 3, we have studied the characteristics of concurrency bugs, and the internals of their detection algorithms, while pointing out the fundamental differences between bugs and vulnerabilities. We gathered and analyzed reported concurrency and concurrency memory corruption vulnerability information in the CVE databases, and our results indicate they are an emerging threat. Moreover, we have scrutinized concurrency memory corruption detection approaches and found that DBI is a core technology.

In Chapter 4, we have been the first to define the foundations of DBI, and to dissect the underlying instrumentation techniques mapping them to actual implementations. We developed a framework to evaluate the performance of both process-level and whole-system tools, trying to minimize the impact of the tool. Analyzing our results, we found that there is no one-fits-all approach. In fact, expressiveness, transparency and performance are the pillars of DBI systems, and the performance of the underlying techniques depends on the instrumented event, the target program's nature and the frequency of observed events.

The initially formulated hypothesis for this research was:

> «*Memory errors remain as one of the main threats to current systems, their concurrent variant has largely gone overlooked, and is harder to detect.*»

The different research works that we have carried out during this dissertation and that are summarized in the paragraphs above validate our starting hypothesis.

First, by collecting statistical information about the reported memory corruption vulnerabilities from 2002 to the end of 2022, we found that the number of reported memory corruption vulnerabilities has been consistent over the years until 2015, when it experienced a remarkable growth that resulted in around 1000 vulnerabilities being reported per 3 months.

Moreover, memory errors and their publicly available exploits account for more or less 20% of all reported vulnerabilities in a consistent fashion over the years. This clearly shows that memory errors remain as one of the main threats to systems since their early detection, being 17% of them critical.

Furthermore, the number of concurrency memory corruption vulnerabilities started to gradually increase from the early years until 2014, from where it experienced a substantial and continuous growth up to the present. Nearly 76% of the total number of concurrency memory corruption vulnerabilities have been reported over the period from 2017 to 2022, and approximately half off the total from 2019 to 2022, following an upward trend.

We also found that unlike sequential detection tools, there are substantially fewer and less mature detection systems designed towards concurrency vulnerabilities. Temporal memory corruption has been the focus of most of the proposed concurrency vulnerability detection methods due to their close alignment with concurrency, whereas spatial errors are a greater challenge. These detection approaches rely on instrumentation functionality to monitor synchronization primitives, memory accesses, orchestrate thread scheduling, and so on.

Moreover, by analyzing and evaluating Dynamic Binary Instrumentation (DBI) techniques and implementations, the results obtained pointed out the need for selecting the proper instrumentation technique depending on the particular needs. Expressiveness, transparency, and performance properties play an important role on the overall effectiveness and efficiency of the developed solutions.

5.2 Future Research

Concurrency memory corruption vulnerabilities are a relatively new topic and that is the main reason behind the little attention they have received, along with the increased complexity on their detection. However, this means that there are several open problems, challenges, and opportunities to contribute to this field and to collaboratively help to secure modern systems.

Regarding future lines of work around this doctoral research, we believe that there is room for further exploration to develop new approaches to effectively detect the already addressed concurrency memory corruption classes (i.e., temporal errors), or to target those that have received less

attention (i.e., spatial errors). Conversely, observing the problem from different perspectives and narrowing the scope to specific cases can also yield interesting results in concurrency vulnerability detection.

In particular, one of such perspectives are cross-thread stack accesses. This programming patter happens when a thread accesses another thread's stack. This kind of accesses are supported and even required by the POSIX standard and compilers such as GCC, which state that thread-local variables and addresses are accessible to all threads in the same process.

Currently, cross-thread stack accesses are only considered a bad practice. However, they can result in undefined behavior if, for instance, values of popped stack frames are read or overwritten, leading to potential exploit opportunities such as to leak stack memory contents. Furthermore, the newly discovered concurrency vulnerabilities suggest that more such accesses may take place in an unintended and unsafe manner.

Several approaches could be adopted towards this challenge. Given the knowledge base, the different techniques that leverage DBI should be evaluated to build a suitable solution. A research work in this direction will expose cross-thread stack accesses as a dangerous strategy and not just a bad practice.

Bibliography

[ACCH09] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, 2009.

[ACF12] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[ADH+21] Udit Agarwal, Pantazis Deligiannis, Cheng Huang, Kumseok Jung, Akash Lal, Immad Naseer, Matthew Parkinson, Arun Thangamani, Jyothi Vedurada, and Yunpeng Xiao. Nekara: Generalized Concurrency Testing. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

[AMM+22] Fatimah K Aljaafari, Rafael Menezes, Edoardo Manino, Fedor Shmarov, Mustafa A Mustafa, and Lucas C Cordeiro. Combining BMC and Fuzzing Techniques for Finding Software Vulnerabilities in Concurrent Programs. *IEEE Access*, 2022.

[ANSB19] Amir Afianian, Salman Niksefat, Babak Sadeghiyan, and David Baptiste. Malware Dynamic Analysis Evasion Techniques: A Survey. *ACM Computing Surveys (CSUR)*, 52(6):1–28, 2019.

[ASBM14] Abdulla Alwabel, Hao Shi, Genevieve Bartlett, and Jelena Mirkovic. Safe and automated live malware experimentation

on public testbeds. In *Proceedings of the USENIX Workshop on Cyber Security Experimentation and Test*, 2014.

[BBR16] Michael Brengel, Michael Backes, and Christian Rossow. Detecting Hardware-Assisted Virtualization. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016.

[BCD+18] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 2018.

[BCM10] Michael D Bond, Katherine E Coons, and Kathryn S McKinley. PACER: Proportional Detection of Data Races. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[BCN+17] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 2017.

[Bel05] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[BHL+08] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically Identifying Trigger-based Behavior in Malware. In *Botnet Detection*, pages 65–88. Springer, 2008.

[BJA+21] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, et al. Using Lightweight Formal Methods to Validate a Key-value Storage Node in Amazon S3. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[BJFL11] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *ACM Symposium on Information, Computer and Communications Security*, 2011.

[BLCH19] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. Effective Static Analysis of Concurrency Use-After-Free Bugs in Linux Device Drivers. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019.

[BM11] Andrew R Bernat and Barton P Miller. Anywhere, Any-Time Binary Instrumentation. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, 2011.

[Bru04] Derek Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[BY17] Alexei Bulazel and Bülent Yener. A Survey on Automated Dynamic Malware Analysis Evasion and Counter-Evasion: PC, Mobile, and Web. In *Proceedings of the Reversing and Offensive-oriented Trends Symposium*, 2017.

[BZ06] Emery D Berger and Benjamin G Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[BZ11] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *International Symposium on Code Generation and Optimization (CGO)*, 2011.

[CAM+08] Xu Chen, Jon Andersen, Z Morley Mao, Michael Bailey, and Jose Nazario. Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, 2008.

[CBG17] Xi Chen, Herbert Bos, and Cristiano Giuffrida. Codearmor: Virtualizing the code space to counter disclosure attacks. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

[CC12] Yan Cai and WK Chan. MagicFuzzer: Scalable Deadlock Detection for Large-Scale Applications. In *Proceedings of the*

34th International Conference on Software Engineering (ICSE), 2012.

[CCH06] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[CGJ09] Xiang Cai, Yuwei Gui, and Rob Johnson. Exploiting UNIX File-System Races via Algorithmic Complexity Attacks. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy (SP)*, 2009.

[CGS+20] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *Proceedings of the USENIX Security Symposium (SEC)*, 2020.

[CGX+20] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs. In *Proceedings of the USENIX Security Symposium (USENIX Security 20)*, 2020.

[CKC11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[Cla] Clang static analyzer. https://clang-analyzer.llvm.org/.

[Coh22] Michael Cohen. The Rekall Forensic and Incident Response Framework. https://github.com/google/rekall, 2022.

[Com] Cwe - 2021 cwe top 25 most dangerous software weaknesses. https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html.

[CRD20] Tobias Cloosters, Michael Rodler, and Lucas Davi. Teerex: Discovery and exploitation of memory corruption vulnerabilities in sgx enclaves. In *USENIX Security Symposium*, 2020.

[cvea] NVD - CVE-2010-5298. https://nvd.nist.gov/vuln/detail/CVE-2010-5298.

[cveb] NVD - CVE-2019-11366. https://nvd.nist.gov/vuln/detail/CVE-2019-11366.

[cvec] NVD - CVE-2021-3348. https://nvd.nist.gov/vuln/detail/CVE-2021-3348.

[CXS+05] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, 2005.

[CYW+21] Yan Cai, Hao Yun, Jinqiu Wang, Lei Qiao, and Jens Palsberg. Sound and Efficient Concurrency Bug Prediction. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

[CYZ21] Yuandao Cai, Peisen Yao, and Charles Zhang. Canary: Practical Static Detection of Inter-Thread Value-Flow Bugs. In *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2021.

[CZM+19] Yan Cai, Biyun Zhu, Ruijie Meng, Hao Yun, Liang He, Purui Su, and Bin Liang. Detecting Concurrency Memory Corruption Vulnerabilities. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019.

[DBXP20] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *IEEE Symposium on Security and Privacy*, 2020.

[DCN+19] Daniele Cono D'Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro, and Lorenzo Cavallaro. SoK: Using Dynamic Binary Instrumentation for Security (And How You May Get Caught Red Handed). In *Proceedings of the ACM Asia Conference on Computer and Communications Security*, 2019.

[Des97] Solar Designer. Return-to-libc attack. *Bugtraq*, *Aug*, 1997.

[DGHH+15] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable Reverse Engineering with PANDA. In *Proceedings of the Program Protection and Reverse Engineering Workshop*, 2015.

[DGLZ+11] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2011.

[DGSTG09] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. Robust Signatures for Kernel Data Structures. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2009.

[DIPQ22] Daniele Cono D'Elia, Lorenzo Invidia, Federico Palmaro, and Leonardo Querzoni. Evaluating Dynamic Binary Instrumentation Systems for Conspicuous Features and Artifacts. *Digital Threats: Research and Practice (DTRAP)*, 3(2):1–13, 2022.

[DRSL08] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2008.

[DWA+19] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters

for Security. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2019.

[DY18] Gregory J Duck and Roland HC Yap. Effectivesan: type and memory error detection using dynamically typed c/c++. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018.

[DZX13] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. Spider: Stealthy Binary Program Instrumentation and Debugging via Hardware Virtualization. In *Proceedings of the Annual Computer Security Applications Conference*, 2013.

[EA03] Dawson Engler and Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[EQT07] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a Race and Transaction-Aware Java Runtime. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[ESKK12] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A Survey on Automated Dynamic Malware-Analysis Techniques and Tools. *ACM Computing Surveys (CSUR)*, 44(2):6, 2012.

[exp] Exploit Database - Exploits for Penetration Testers, Researchers, and Ethical Hackers. `https://www.exploit-db.com/`.

[FF04] Cormac Flanagan and Stephen N Freund. Atomizer: a Dynamic Atomicity Checker for Multithreaded Programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2004.

[FF09] Cormac Flanagan and Stephen N Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[FFY08] Cormac Flanagan, Stephen N Freund, and Jaeheon Yi. Velodrome: a Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[FL12] Yangchun Fu and Zhiqiang Lin. Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2012.

[FLJ18] Xinwei Fu, Dongyoon Lee, and Changhee Jung. nAdroid: Statically Detecting Ordering Violations in Android Applications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO)*, 2018.

[FMRS12] Azadeh Farzan, P Madhusudan, Niloofar Razavi, and Francesco Sorrentino. Predicting Null-Pointer Dereferences in Concurrent Programs. In *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2012.

[FQ03] Cormac Flanagan and Shaz Qadeer. A Type and Effect System for Atomicity. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[FRF22] Ailton Santos Filho, Ricardo J Rodríguez, and Eduardo L Feitosa. Evasion and Countermeasures Techniques to Detect Dynamic Binary Instrumentation Frameworks. *Digital Threats: Research and Practice (DTRAP)*, 3(2):1–28, 2022.

[GAFM21] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. Snowboard: Finding Kernel Concurrency Bugs through Systematic Inter-thread Communication Analysis. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[GCJ17] Xinyang Ge, Weidong Cui, and Trent Jaeger. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. In *Pro-*

ceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2017.

[GDB22] GDB. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/, 2022.

[GDXJ11] Zhongshu Gu, Zhui Deng, Dongyan Xu, and Xuxian Jiang. Process Implanting: A New Active Introspection Framework for Virtualization. In *Proceedings of the IEEE International Symposium on Reliable Distributed Systems*, 2011.

[GGL+22] Fangming Gu, Qingli Guo, Lian Li, Zhiniang Peng, Wei Lin, Xiaobo Yang, and Xiaorui Gong. COMRace: Detecting Data Race Vulnerabilities in COM Objects. In *Proceedings of the USENIX Security Symposium (USENIX Security 22)*, 2022.

[GKK+18] Enes Göktas, Benjamin Kollenda, Philipp Koppe, Erik Bosman, Georgios Portokalidis, Thorsten Holz, Herbert Bos, and Cristiano Giuffrida. Position-independent code reuse: On the effectiveness of aslr in the absence of information disclosure. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018.

[Goo] Google. Syzkaller - Kernel fuzzer. https://github.com/google/syzkaller.

[Goo22] Google. TinyInst - A lightweight dynamic instrumentation library. https://github.com/googleprojectzero/TinyInst, 2022.

[GXLA12] Joseph L Greathouse, Hongyi Xin, Yixin Luo, and Todd Austin. A Case for Unlimited Watchpoints. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[HCA+15] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of data-oriented exploits. In *USENIX Security Symposium*, 2015.

[HD21] Jiaqi Hong and Xuhua Ding. A Novel Dynamic Analysis Infrastructure to Instrument Untrusted Execution Flow Across

User-kernel Spaces. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2021.

[hel] Helgrind: a thread error detector. https://valgrind.org/docs/manual/hg-manual.html.

[Hex22] Hex-Rays. IDA Pro. https://www.hex-rays.com/products/ida/, 2022.

[HJP+16] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. Typesan: Practical type confusion detection. In *ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[HMR14] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[HPY+14] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. Make It Work, Make It Right, Make It Fast: Building a Platform-Neutral Whole-System Dynamic Binary Analysis Platform. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2014.

[HSA+16] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE Symposium on Security and Privacy*, 2016.

[Hua18] Jeff Huang. UFO: Predictive Concurrency Use-After-Free Detection. In *Proceedings of the IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018.

[IAJP18] Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[JBC+17] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. Hextype: Efficient detection of type confusion errors for c++. In *ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[JBLH22] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Context-Sensitive and Directional Concurrency Fuzzing for Data-race Detection. In *Proceedings of the 29th Network and Distributed System Security Symposium (NDSS)*, 2022.

[JBZ+14] Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E Porter, and Radu Sion. SoK: Introspections on Trust and the Semantic Gap. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2014.

[JC13] Mateusz Jurczyk and Gynvael Coldwind. Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns. 2013.

[JKS+19] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding Kernel Race Bugs through Fuzzing. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[JMG+02] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: a Safe Dialect of C. In *USENIX Annual Technical Conference (ATC), General Track*, 2002.

[JTZ+08] Horatiu Jula, Daniel M Tralamazza, Cristian Zamfir, George Candea, et al. Deadlock Immunity: Enabling Systems to Defend Against Deadlocks. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2008.

[JZS+17] Xiangkun Jia, Chao Zhang, Purui Su, Yi Yang, Huafeng Huang, and Dengguo Feng. Towards efficient heap overflow discovery. In *USENIX Security Symposium*, 2017.

[KC12] Alain Ketterlin and Philippe Clauss. Profiling data-dependence to assist parallelization: Framework, scope, and

optimization. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[KGKM+22] Mohammad Sina Karvandi, MohammadHosein Gholamrezaei, Saleh Khalaj Monfared, Soroush Meghdadizanjani, Behrooz Abbassi, Ali Amini, Reza Mortazavi, Saeid Gorgin, Dara Rahmati, and Michael Schwarz. HyperDbg: Reinventing Hardware-Assisted Debugging. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[KKR+21] Taegyu Kim, Vireshwar Kumar, Junghwan Rhee, Jizhou Chen, Kyungtae Kim, Chung Hwan Kim, Dongyan Xu, and Dave (Jing) Tian. PASAN: Detecting Peripheral Access Concurrency Bugs within Bare-Metal Embedded Applications. In *Proceedings of the USENIX Security Symposium (USENIX Security 21)*, 2021.

[KMV17] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. Dynamic Race Prediction in Linear Time. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[KRC+18] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[KSP+18] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *USENIX Security Symposium*. 2018.

[KT14] Daniel Kroening and Michael Tautschnig. Cbmc–c bounded model checker. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2014.

[KV21] Michalis Kokologiannakis and Viktor Vafeiadis. GenMC: A Model Checker for Weak Memory Models. In *Proceedings of the 33rd International Conference on Computer-Aided Verification (CAV)*, 2021.

[KYH+09] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. Emulating Emulation-Resistant Malware. In *Proceedings of the ACM workshop on Virtual Machine Security*, 2009.

[KZBK18] Julian Kirsch, Zhechko Zhechev, Bruno Bierbaumer, and Thomas Kittel. PwIN–Pwning Intel piN: Why DBI is Unsuitable for Security Applications. In *European Symposium on Research in Computer Security (ESORICS)*, 2018.

[KZK22] Youngjoo Ko, Bin Zhu, and Jong Kim. Fuzzing with Automatically Controlled Interleavings to Detect Concurrency Bugs. *Journal of Systems and Software*, 2022.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[Law96] Kevin P Lawton. Bochs: A Portable PC Emulator for Unix/X. *Linux Journal*, 1996(29es):7, 1996.

[LCC10] Zhifeng Lai, Shing-Chi Cheung, and Wing Kwong Chan. Detecting Atomic-Set Serializability Violations in Multithreaded Programs through Active Randomized Testing. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, 2010.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[Lef] Lefebvre, Jean-Pierre. atftp. https://sourceforge.net/projects/atftp/.

[LHBF14] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. Sok: Automated software diversity. In *IEEE Symposium on Security and Privacy*, 2014.

[LLM+19] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. Efficient Scalable Thread-Safety-Violation Detection: Finding Thousands of Concurrency Bugs During Testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[LML21] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. ExpRace: Exploiting Kernel Races through Raising Interrupts. In *Proceedings of the USENIX Security Symposium (USENIX Security 21)*, 2021.

[LMP+14] Tamas K Lengyel, Steve Maresca, Bryan D Payne, George D Webster, Sebastian Vogl, and Aggelos Kiayias. Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System. In *Proceedings of the Annual Computer Security Applications Conference*, 2014.

[LNW+19] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. {PAC} it up: Towards pointer integrity using {ARM} pointer authentication. In *USENIX Security Symposium*, 2019.

[LPSZ08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes: a Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[LRZ+11] Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang. SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2011.

[LSL+15] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P Chung, Taesoo Kim, and Wenke Lee. Aslr-guard: Stopping address space leakage for code reuse attacks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[LTQZ06] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[LZL+18] Changming Liu, Deqing Zou, Peng Luo, Bin B Zhu, and Hai Jin. A Heuristic Framework to Detect Concurrency Vulnerabilities. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, 2018.

[MANP17] Najmeh Miramirkhani, Mahathi Priya Appini, Nick Nikiforakis, and Michalis Polychronakis. Spotless Sandboxes: Evading Malware Analysis Systems Using Wear-and-Tear Artifacts. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2017.

[MB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[MHH+19] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering*, 2019.

[MKV18] Umang Mathur, Dileep Kini, and Mahesh Viswanathan. What Happens-After the First Race? Enhancing the Predictive Power of Happens-Before Based Dynamic Race Detection. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2018.

[mpk22] Memory Protection Keys - The Linux kernel Documentation. `https://www.kernel.org/doc/html/latest/core-api/protection-keys.html`, 2022.

[MSC+17] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. DR.CHECKER: A soundy analysis for linux kernel drivers. In *USENIX Security Symposium*, 2017.

[MV20] Umang Mathur and Mahesh Viswanathan. Atomicity Checking in Linear Time using Vector Clocks. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[NBMM12] Santosh Nagarakatte, Sebastian Burckhardt, Milo MK Martin, and Madanlal Musuvathi. Multicore Acceleration of Priority-based Schedulers for Concurrency Bug Detection. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2012.

[NMW02] George C Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2002.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[NSJ+09] Anh M Nguyen, Nabil Schear, HeeDong Jung, Apeksha Godiyal, Samuel T King, and Hai D Nguyen. MAVMM: Lightweight and Purpose Built VMM for Malware Analysis. In *Proceedings of the Annual Computer Security Applications Conference*, 2009.

[nvd] NVD - Data Feeds. `https://nvd.nist.gov/vuln/data-feeds`.

[NZ17] Zhenyu Ning and Fengwei Zhang. Ninja: Towards Transparent Tracing and Debugging on ARM. In *Proceedings of the USENIX Security Symposium (SEC)*, 2017.

[NZMZ09] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[NZMZ10] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety

for c. In *International Symposium on Memory Management*, 2010.

[ÖRBG20] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Parmesan: Sanitizer-guided greybox fuzzing. In *USENIX Security Symposium*, 2020.

[PBG15] Mathias Payer, Antonio Barresi, and Thomas R Gross. Fine-grained control-flow integrity through binary hardening. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2015.

[PCM+17] Mario Polino, Andrea Continella, Sebastiano Mariani, Stefano D'Alessio, Lorenzo Fontana, Fabio Gritti, and Stefano Zanero. Measuring and Defeating Anti-Instrumentation-Equipped Malware. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2017.

[PDZ+14] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. X-Force: Force-Executing Binary Programs for Security Applications. In *Proceedings of the USENIX Security Symposium (SEC)*, 2014.

[PFH06] Polyvios Pratikakis, Jeffrey S Foster, and Michael Hicks. LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[PHY15] Aravind Prakash, Xunchao Hu, and Heng Yin. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[Pie22] Cody Pierce. PyEmu: x86 Emulator in Python. `https://github.com/codypierce/pyemu`, 2022.

[PLHM08] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. *Acm operating systems review*, 2008.

[PLM+18] Sergej Proskurin, Tamas Lengyel, Marius Momeu, Claudia Eckert, and Apostolis Zarras. Hiding in the Shadows: Empowering ARM for Stealthy Virtual Machine Introspection. In *Proceedings of the Annual Computer Security Applications Conference*, 2018.

[PLZ09] Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: Exposing Atomicity Violation Bugs from their Hiding Places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[PMRB09] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. A Fistful of Red-Pills: How to Automatically Generate Procedures to Detect CPU Emulators. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[PVH10] Sangmin Park, Richard W Vuduc, and Mary Jean Harrold. Falcon: Fault Localization in Concurrent Programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, 2010.

[PWL20] Ivan Pustogarov, Qian Wu, and David Lie. Ex-vivo dynamic analysis framework for Android device drivers. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2020.

[QV22] Nguyen Anh Quynh and Dang Hoang Vu. Unicorn CPU emulator framework. `https://github.com/unicorn-engine/unicorn`, 2022.

[Rav22] Ole Andre Vadla Ravnas. Frida - A world-class dynamic instrumentation framework. `https://www.frida.re/`, 2022.

[RBSS12] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 2012.

[RGB18] Jake Roemer, Kaan Genç, and Michael D Bond. High-Coverage, Unbounded Sound Predictive Race Detection. In

Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2018.

[RKK07] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. Detecting System Emulators. In *Proceedings of the International Conference on Information Security (ISC)*, 2007.

[rVM17] rVMI. rVMI - A New Paradigm For Full System Analysis. https://github.com/fireeye/rvmi, 2017.

[SAM14] Hao Shi, Abdulla Alwabel, and Jelena Mirkovic. Cardinal Pill Testing of System Virtual Machines. In *Proceedings of the USENIX Security Symposium (SEC)*, 2014.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems (TOCS)*, 1997.

[SBPV12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. 2012.

[SCL+18] Bin Shi, Lei Cui, Bo Li, Xudong Liu, Zhiyu Hao, and Haiying Shen. ShadowMonitor: An Effective in-VM Monitoring Framework with Hardware-enforced Isolation. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses*, 2018.

[SFL14] Alireza Saberi, Yangchun Fu, and Zhiqiang Lin. HybridBridge: Efficiently Bridging the Semantic Gap in Virtual Machine Introspection via Decoupled Execution and Training Memoization. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[Sha07] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM conference on Computer and Communications Security*, 2007.

[SHD+19] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz.

Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[SI09] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA)*, 2009.

[SLL+16] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing kernel security invariants with data flow integrity. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[SLR+19] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. Sok: sanitizing for security. In *IEEE Symposium on Security and Privacy*, 2019.

[SMB+10] David Sehr, Robert Muth, Cliff L Biffle, Victor Khimenko, Egor Pasko, Bennet Yee, Karl Schimpf, and Brad Chen. Adapting software fault isolation to contemporary cpu architectures. In *USENIX Security Symposium*, 2010.

[SMD+13] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy*, 2013.

[SN05] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference*, 2005.

[SPP+04] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *ACM conference on Computer and communications security (CCS)*, 2004.

[SPWS13] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP)*, 2013.

[STL+15] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *IEEE Symposium on Security and Privacy*, 2015.

[SWJX11] Deepa Srinivasan, Zhi Wang, Xuxian Jiang, and Dongyan Xu. Process Out-Grafting: An Efficient "Out-of-VM" Approach for Fine-Grained Process Execution Monitoring. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2011.

[SWS+16] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(State of) the Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP)*, 2016.

[SWYS11] Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. Generating Data Race Witnesses by an SMT-based Analysis. In *NASA Formal Methods Symposium (NFM)*, 2011.

[The] The Linux kernel user's and administrator's guide. Deadline Task Scheduling. https://www.kernel.org/doc/html/latest/scheduler/sched-deadline.html.

[UCY+06] Gang-Ryung Uh, Robert Cohn, Bharadwaj Yadavalli, Ramesh Peri, and Ravi Ayyagari. Analyzing Dynamic Binary Instrumentation Overhead. In *Workshop on Binary Instrumentation and Applications (WBIA) at ASPLOS*, 2006.

[Uga22] Ugarte-Pedrero, Xabier. PyREBox: Python scriptable Reverse Engineering Sandbox, a Virtual Machine instrumentation and inspection framework based on QEMU. https://github.com/Cisco-Talos/pyrebox, 2022.

[Und] Undefinedbehaviorsanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[UPBSB16] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. RAMBO: Run-Time Packer Analysis with

Multiple Branch Observation. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016.

[vdVAS+17] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrdia. The dynamics of innocent flesh on the bone: Code reuse ten years later. In *ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[VdVCB+12] Victor Van der Veen, Lorenzo Cavallaro, Herbert Bos, et al. Memory errors: The past, the present, and the future. In *International Conference on Recent Advances in Intrusion Detection (RAID)*, 2012.

[VDVGC+16] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2016.

[VE12] Sebastian Vogl and Claudia Eckert. Using Hardware Performance Events for Instruction-Level Monitoring on the x86 Architecture. In *Proceedings of the European Workshop on Systems Security (EuroSec)*, 2012.

[VY05] Amit Vasudevan and Ramesh Yerraballi. Stealth Breakpoints. In *Proceedings of the Annual Computer Security Applications Conference*, 2005.

[VY06] Amit Vasudevan and Ramesh Yerraballi. Cobra: Fine-grained Malware Analysis using Stealth Localized-executions. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2006.

[Wal22] AAron Walters. Volatility Framework - Volatile memory extraction utility framework. `https://github.com/volatilityfoundation/volatility`, 2022.

[WHW+22] Cheng Wen, Mengda He, Bohao Wu, Zhiwu Xu, and Shengchao Qin. Controlled Concurrency Testing via Periodical Scheduling. In *Proceedings of the IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022.

[WKL+17] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. How Double-Fetch Situations Turn into Double-Fetch Vulnerabilities: A Study of Double-Fetches in the Linux Kernel. In *Proceedings of the USENIX Security Symposium (USENIX Security 17)*, 2017.

[WLS+19] Diyu Wu, Jie Liu, Yulei Sui, Shiping Chen, and Jingling Xue. Precise Static Happens-Before Analysis for Detecting UAF Order Violations in Android. In *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019.

[WTLP14] Jiesheng Wei, Anna Thomas, Guanpeng Li, and Karthik Pattabiraman. Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014.

[WW19] Meng Wu and Chao Wang. Abstract Interpretation under Speculative Execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2019.

[WXL+20] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2020.

[XKZK20] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data Race Fuzzing for Kernel File Systems. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[XLCZ17] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. STACCO: Differentially Analyzing Side-Channel

Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[YCSS12] Junfeng Yang, Ang Cui, Sal Stolfo, and Simha Sethumadhavan. Concurrency Attacks. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism (HotPar 12)*, 2012.

[YJZY12] Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. V2E: Combining Hardware Virtualization and Software Emulation for Transparent and Extensible Malware Analysis. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2012.

[YS10] Heng Yin and Dawn Song. TEMU: Binary Code Analysis via Whole-System Layered Annotative Execution. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-3*, 2010.

[YSCX18] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2018.

[YWC+21] Kunpeng Yu, Chenxu Wang, Yan Cai, Xiapu Luo, and Zijiang Yang. Detecting Concurrency Vunerabilities Based on Partial Orders of Memory and Thread Events. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

[Zal] Michal Zalewski. American Fuzzy Lop. `https://lcamtuf.coredump.cx/afl/`.

[ZDXG17] Siqi Zhao, Xuhua Ding, Wen Xu, and Dawu Gu. Seeing Through The Same Lens: Introspecting Guest Address Space At Native Speed. In *Proceedings of the USENIX Security Symposium (SEC)*, 2017.

[ZFL15] Junyuan Zeng, Yangchun Fu, and Zhiqiang Lin. PEMU: A Pin Highly Compatible Out-of-VM Dynamic Binary Instrumentation Framework. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2015.

[ZGQ+18] Shixiong Zhao, Rui Gu, Haoran Qiu, Tsz On Li, Yuexuan Wang, Heming Cui, and Junfeng Yang. OWL: Understanding and Detecting Concurrency Attacks. In *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018.

[ZLO+11] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. ConSeq: Detecting Concurrency Bugs through Sequential Errors. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[ZLS+15] Fengwei Zhang, Kevin Leach, Angelos Stavrou, Haining Wang, and Kun Sun. Using Hardware Features for Increased Debugging Transparency. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[ZSL10] Wei Zhang, Chong Sun, and Shan Lu. ConMem: Detecting Severe Concurrency Bugs through an Effect-Oriented Approach. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[ZWCW14] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. Armlock: Hardware-based fault isolation for arm. In *ACM SIGSAC conference on Computer and Communications Security (CCS)*, 2014.

This dissertation was completed in Bilbao
on May 29, 2023.