UNIVERSITY OF CALIFORNIA,
IRVINE

Precise and Efficient Dynamic Analysis of Systems Software

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Dokyung Song

Dissertation Committee:
Professor Michael Franz, Chair
Professor Qi Alfred Chen
Professor Brent ByungHoon Kang

2020

DEDICATION

I humbly dedicate this work to my father, mother, sisters and brother in law, who have all stood by my side supporting me along the whole journey.

No one who achieves success does so without the help of others.
The wise and confident acknowledge this help with gratitude.

— Alfred North Whitehead

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF LISTINGS

ACKNOWLEDGMENTS

First and foremost, I am deeply grateful to my advisor, Professor Dr. Michael Franz for giving me the opportunity to conduct research when I barely had experience in the field of systems security. Without him, the intellectual joy I obtained throughout my study and research in the field would not have been there. More importantly, without his guidance, advice, support, and often a lot of patience throughout my Ph.D., I would not have seen the light at the end of my Ph.D. journey.

I am also deeply grateful to Professor Dr. Qi Alfred Chen, and Professor Dr. Brent ByungHoon Kang for serving on my Ph.D. dissertation committee.

Next, I would like to thank my previous and current labmates—Joseph, Paul, Matthew, Taemin, Alex, Anil, Prabhu, Mitchel, Fabian, Min-Yih, Chinmay, Jason, Tommaso, Mohaned, Brian, and Julian. I especially thank my officemate Joseph Nash for enjoyable daily conversations on matters ranging from academic ones—such as how to conduct good research and how to write better—to personal ones. I thank Paul Kirth for his generous help with improving my writing skills, as well as espresso brewing skills. I also thank Matthew Dees for many thought-provoking conversations on academic and research matters during my last year of Ph.D.; I very much enjoyed it. I would also like to thank previous and current post-doctoral scholars I have worked with in the lab—Professor Dr. Stijn Volckaert, Dr. Yeoul Na, Dr. Per Larsen, Dr. David Gens, and Dr. Adrian Dabrowski—for their close guidance and timely advice that was much needed along the way I was becoming an independent researcher.

Next, I would like to express my deep gratitude to my collaborators outside of my lab—Felicitas Hetzelt, Dipanjan Das, Dr. Jonghwan Kim, Dr. Chad Spensky, and Professor Dr. Jean-Pierre Seifert. Though we were (most of the time) physically apart, without their efforts and inputs, I would not have produced the same material presented in this dissertation thesis.

Last but not least, I would also like to thank Charles Lee and Lily Park for strongly supporting me as a friend when I was enduring the excruciating time of the pandemic in my last year in the Ph.D. program. I also thank Haehyun Cho, Sanghyun Hong, and Professor Dr. Doowon Kim for always being there listening to my ramblings when I was trying to survive the Ph.D. program. I also thank Professor Dr. Yonghwi Kwon and Professor Dr. Yuseok Jeon for their helpful advice. I am also grateful to my internship hosts—Kostya Serebryany, Matt Morehouse, Mitch Phillips, Aaron Green, Ryan Nakamoto, and Brian Rosenberg—for having me in their team and allowing me to gain valuable experience in the industry.

I also gratefully acknowledge that the material in this dissertation thesis is based upon work partially supported by the Defense Advanced Research Projects Agency under contracts FA8750-15-C-0124 and FA8750-15-C-0085, by the United States Office of Naval Research under contract N00014-17-1-2782, by the National Science Foundation under awards CNS-1619211 and CNS-1513837. Any opinions, findings, and conclusions or recommendations expressed in this material do not necessarily reflect the views of the funding agencies.

CURRICULUM VITAE

Dokyung Song

EDUCATION

Doctor of Philosophy in Computer Science	**2020**
University of California, Irvine	*Irvine, California*
Master of Science in Computer Science	**2019**
University of California, Irvine	*Irvine, California*
Bachelor of Science in Electrical and Computer Engineering	**2014**
Seoul National University	*Seoul, South Korea*

RESEARCH EXPERIENCE

Graduate Student Researcher	**2016–2020**
University of California, Irvine	*Irvine, California*
Visiting Researcher	**Summer 2019**
Techniche Universität Berlin	*Berlin, Germany*

TEACHING EXPERIENCE

Teaching Assistant	**Spring 2017**
University of California, Irvine	*Irvine, California*
ICS 46: Data Structure Implementation and Analysis	
Teaching Assistant	**Fall 2017**
University of California, Irvine	*Irvine, California*
ICS 46: Data Structure Implementation and Analysis	

REFEREED CONFERENCE PUBLICATIONS

Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints **USENIX Security 2020**
Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent ByungHoon Kang, Jean-Pierre Seifert, and Michael Franz.
USENIX Security Symposium

Distributed Heterogeneous N-Variant Execution **DIMVA 2020**
Alexios Voulimeneas, Dokyung Song, Fabian Parzefall, Yeoul Na, Per Larsen, Michael Franz, and Stijn Volckaert.
International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment

SoK: Sanitizing for Security **IEEE S&P 2019**
Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz.
IEEE Symposium on Security and Privacy

PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary **NDSS 2019**
Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz.
Network and Distributed System Security Symposium

PartiSan: Fast and Flexible Sanitization via Run-time Partitioning **RAID 2018**
Julian Lettner, Dokyung Song, Taemin Park, Per Larsen, Stijn Volckaert, and Michael Franz.
International Symposium on Research in Attacks, Intrusions and Defenses

ABSTRACT OF THE DISSERTATION

Precise and Efficient Dynamic Analysis of Systems Software

by

Dokyung Song

Doctor of Philosophy in Computer Science

University of California, Irvine, 2020

Professor Michael Franz, Chair

Today's mainstream operating systems (OSs) have *monolithic* kernels, in which low-level systems software such as device drivers, networking systems, and file systems all run within the kernel with no separation of privilege between them. This means that exploiting a single vulnerability present in any kernel subsystem gives adversaries access to the entire OS. It has been repeatedly demonstrated that OS kernels can be compromised locally or even remotely, through their wide attack surface—the system call interface as well as the peripheral interface.

There exists a significant body of research aimed at finding vulnerabilities in software. Fuzzing, among others, is widely regarded as a practical and effective approach to finding vulnerabilities. Most of the fuzzing research, however, has targeted user-space software. Unfortunately, fuzzing systems software running in kernel space can be more challenging than fuzzing user-space software, as existing kernel fuzzers suffer from unique challenges that arise in kernel space. More specifically, existing fuzzers for OS kernels are (i) *imprecise* in that they do not accurately model the full capabilities of possible attackers, e.g., attackers on the peripheral hardware side, or (ii) *inefficient* due to various delays caused by system crashes, asynchronous input processing, etc. This dissertation presents two dynamic analysis techniques that significantly alleviate these problems: (i) a technique that enables fuzzing the peripheral input space of OS kernels, which precisely models the capabilities of a strong attacker on the peripheral side, and (ii) a virtual machine checkpointing technique that can accelerate OS kernel fuzzing, making dynamic analysis more efficient. The dissertation concludes with a summary of these techniques as well as a recommendation for promising future work directions.

Chapter 1

Introduction

Operating system (OS) kernels form the foundation of multi-user, general-purpose computer systems. A core responsibility of an OS kernel is to ensure the *security* of the computer system. To this end, contemporary OS kernels provide abstractions and mechanisms that can be used to implement various security policies. For instance, an OS kernel may provide a *process* abstraction, which can be used as a means to establish trust boundaries between trustworthy and untrustworthy code. As a privileged component of a computer system, an OS kernel itself also maintains trust boundaries at the interfaces facing unprivileged components (e.g., the system call interface) to ensure its confidentiality, integrity, and availability at all times.

In other words, an OS kernel is a core part of the *trusted computing base* of a computer system [142], which, when compromised, can undermine the security of the entire system. An OS kernel can be compromised, for example, if it has a vulnerability that can be exploited by malicious user applications through the system call interface. Unfortunately, exploitable vulnerabilities in OS kernels abound. OS kernels are typically written in low-level programming languages such as C and C++, which are notoriously insecure for their lack of memory and type safety. Not surprisingly, programmers routinely fail to reason about memory and type safety, introducing subtle memory and type safety errors in OS kernel code. OS vendors confirmed such failures many times [162, 112]; the vulnerabilities stemming from the lack of memory and type safety occupy a large portion of OS kernel vulnerabilities.

Memory safety errors in OS kernels, when exploited, can compromise the confidentiality, integrity, and availability of the entire system, i.e., the underlying OS kernel itself as well as all programs running in user space [167]. A memory overwrite vulnerability in a program, for example, compromises the integrity of the running program when used by an attacker to corrupt data in memory, and the availability when the corrupted memory causes the program to crash. A memory overread vulnerability can be used by an attacker to read sensitive data in memory, compromising the confidentiality of the program. Even Turing-complete arbitrary code execution can, unintendedly, with the privilege of the victim, be achieved out of a single memory overwrite vulnerability when the attacker abuses that vulnerability to carefully overwrite critical control (or even non-control) data in memory [6, 119, 32, 74, 152].

Among the constituents of a modern OS kernel, device drivers, in particular, are notoriously known for vulnerabilities [40, 128, 162]; vulnerabilities in device drivers have frequently been discovered and exploited in OS kernel attacks [14, 34, 19, 17, 123, 47]. Device driver implementations are typically provided by third-party device hardware manufacturers, who are often less skilled in kernel programming and thus more prone to introduce programming errors than the main kernel developers. This problem has been known for decades, and, as a result, a plethora of techniques have been proposed so far to mitigate the consequences of vulnerabilities. A line of prior work attempted to isolate device drivers from the rest of the kernel so as to contain possible consequences of device driver vulnerabilities [25, 166, 72, 42, 60, 97]. Another line of work, which is more fundamental but intrusive, involves a complete re-design of the OS around the *microkernel* architecture [69, 177, 98], which effectively deprivileges most of the kernel subsystems including device drivers by placing them outside of the kernel. Unfortunately, however, these techniques have seen little (or limited) adoption in practice; many contemporary general-purpose OSs such as Linux and Windows still employ *monolithic* kernels where there is no privilege separation between kernel subsystems. This means that device driver vulnerabilities continue to pose a significant threat to the security of OS kernels.

A viable solution, therefore, is to find and fix vulnerabilities in OS kernel subsystems including device drivers during testing, before they are exploited by adversaries. To this end, many software analysis techniques have been developed in recent years, which aim to find vulnerabilities in various software. These analysis efforts are broadly classified into static and dynamic approaches. While static approaches reason about a program at varying forms and levels of abstraction without requiring any program inputs nor executing the program (e.g., Static Driver Verifier in Windows [110, 13, 12, 11]), dynamic approaches concretely reason about actual program executions by running the program with concrete inputs. Fuzzing, among other techniques, is widely considered a promising approach to automatically generating concrete inputs that trigger vulnerabilities. However, most of the fuzzing research thus far has primarily targeted user-mode programs, and research in OS kernel fuzzing, unfortunately, is relatively sparse.

This dissertation attempts to fill the aforementioned gap by developing new techniques that can extend the reach of the fuzzing research to *kernel-mode systems software* such as device drivers. To this end, the rest of this chapter introduces the reader to OS kernel fuzzing—in particular, what constitutes OS kernel fuzzing, and what makes OS kernel fuzzing challenging. Section 1.1 first details the input space (i.e., attack surface) of OS kernels exposed to potential adversaries, and possible consequences of exploiting OS kernel vulnerabilities that can be reached by adversaries through the described input space. Section 1.2 then highlights unique challenges that arise in the domain of OS kernel fuzzing, with a description of why existing approaches fall short of addressing them. Finally, Section 1.3 provides the blueprint of the rest of this dissertation, where the core contributions as well as the new techniques presented throughout the dissertation are summarized.

1.1 OS Kernel Attack Surface

System Call Interface. The primary attack surface for an OS kernel is *the system call interface*, through which the kernel is faced with user-space adversaries. User-space applications talk to the underlying OS kernel by invoking system calls; system calls are handled or rejected by the kernel according to the security policy (e.g., access control policies) employed by the system administrator. Different kernel subsystems make different system calls available to user-space applications. Kernel-mode device drivers, for example, expose `ioctl` system calls and other file-manipulating system calls such as `open`, `read`, and `write`, which abstract device functionalities as file operations on device files.

Adversaries in user space can compromise the underlying OS kernel by attacking this system call interface, i.e., triggering and exploiting vulnerabilities present in the kernel's system call handling code paths. The consequence of such kernel exploits is the attackers' escaping the process sandbox, thereby escalating their privilege from the privilege of the process owner to the kernel privilege. For a monolithic kernel, this means a compromise of the entire OS.

Peripheral Interface. The peripheral interface is another interface through which OS kernels are exposed to potential adversaries. Device drivers, in particular, expose this interface to peripheral devices. Peripheral devices should not be trusted by the OS kernel as dictated by the principle of separation of privilege [144]. More concrete reasons peripheral devices should be considered untrustworthy include: (i) peripheral devices may provide a remote attack vector to potential adversaries through their own physical channels (e.g., network devices may receive malicious packets over the air), and (ii) peripheral hardware is more resource-constrained and thus less capable than the main processor; they often lack basic defense mechanisms so it is easier for adversaries to compromise the peripherals than the main processor [113].

In fact, the untrustworthiness of peripheral devices has repeatedly been evidenced by *remote* peripheral attacks [55, 176, 7, 16, 18, 31], which remotely compromises peripheral devices without communicating with the OS running on the main processor at all. More concerningly, several recently published attacks even demonstrated that such remote peripheral compromise can be turned into a full compromise of the OS kernel running on the main processor by attacking the peripheral interface exposed by device drivers [19, 17]. The consequence of exploiting OS kernel vulnerabilities along its peripheral interface is the attackers' escalating their privilege from that of a single peripheral device to the kernel privilege of the main processor.

1.2 Challenges in Fuzzing OS Kernels

Fuzzing refers to an automated process of discovering vulnerabilities, which works by automatically generating test inputs and by repeatedly executing a target program with the generated inputs. By actually running the program with concrete inputs, fuzzers can detect a vulnerability or unexpected behavior in general, as it manifests as an *uncontrolled* program termination (caused by, for example, a segmentation fault), or as a *controlled* program termination (caused by, for example, an assertion inserted manually by developers or automatically by analysis tools).

Fuzzing has empirically proven to be a practical and effective technique to find vulnerabilities in software [111, 104], including OS kernel subsystems [146, 68, 44, 127, 80, 181, 66, 3, 4, 133, 129]. In particular, Syzkaller, the state-of-the-art OS kernel fuzzer, has discovered hundreds of vulnerabilities from a wide range of components of different OS kernels [63]. Prior work used various program analysis techniques to tackle the challenges of OS kernel fuzzing [80, 44, 181, 3, 68, 146, 127, 66]. For example, a line of work tackled the problem of inferring the dependencies between system calls for interface-aware fuzzing, by using hand-written input grammars [66] or different forms of static and dynamic analysis [68, 127, 44]. Despite recent developments in OS kernel fuzzing, however, many challenges still remain to be addressed. This section describes in detail the two distinct challenges in OS kernel fuzzing that this dissertation tackles.

1.2.1 Fuzzing Peripheral Input Space

Fuzzing the peripheral input space of OS kernels can be substantially different from fuzzing the input space of user-space programs. The input space of user-space programs is standardized (e.g., `stdin`, the file descriptor for standard input) across multiple OSs per the Portable Operating Systems Interface (POSIX) standard. Therefore, fuzzing the input space of user-mode programs, in many cases, is equivalent to repeatedly writing fuzzer-generated test inputs to a file opened using the standard input file descriptor, `stdin`.

The peripheral input space of OS kernels, by contrast, is much lower-level—device drivers interface and communicate with peripheral devices using hardware mechanisms—and, hence, fuzzing becomes more difficult. Consider the Peripheral Component Interconnect (PCI) interface as an example, which uses *memory* as the communication medium; that is, the device drivers running on the main processor use normal memory read and write instructions to communicate with peripheral devices (see Section 3.2). Memory accesses to I/O memory mappings are often indistinguishable from other memory accesses at the source code level, which makes it difficult for static analysis approaches to analyze the PCI interface. There exist dynamic analysis approaches to analyzing the peripheral input space of OS kernels, but they only target either a limited set of peripheral devices [145] or the Universal Serial Bus (USB) interface [147, 133, 1].

1.2.2 High-throughput Fuzzing

Another challenge is *high-throughput* OS kernel fuzzing. Fuzzing works by repeatedly generating test inputs and executing the program under test with those inputs. Therefore, achieving high throughput—defined as the number of test inputs generated by the fuzzer and processed by the program per unit time—is crucial for the overall efficiency of dynamic analysis. Unfortunately, it is difficult to achieve high throughput when fuzzing OS kernel subsystems. Taking device drivers as an example, their execution can easily be prolonged during their loading and initialization, or input processing in general. Low-priority, time-consuming tasks in kernel space are processed asynchronously and in a deferred manner, increasing the total input processing time.

To make matters worse, feeding a fuzzer-generated test input to the OS kernel may change its internal state, which, in turn, can negatively influence processing inputs subsequently generated by the fuzzer. This negative influence includes the system transitioning into unrecoverable error state [44, 155] or unstable state in general, when, for example, a memory corruption bug corrupts random bytes in memory. To achieve *clean-state* fuzzing where there is no interference between test inputs, one may write a clean-up routine and call it after feeding each input, and reboot the system when an input triggers a bug; however, manually-written clean-up routine may not completely recover the state, and a system reboot results in a significant reduction in fuzzing throughput.

As another approach to clean-state OS kernel fuzzing, prior work used a system-level snapshot created at system startup to always restore a clean state of the system before feeding each test input, which also allows to skip time-consuming reboots. However, snapshot techniques at the virtual machine level without optimizations can be too costly (e.g., QEMU's implementation of virtual machine snapshot [2]), and user-mode system snapshot techniques either suffer from similar performance problems [3] or require extensive driver porting efforts when a user-mode kernel is used [181]. Achieving high-throughput *and* clean-state OS kernel fuzzing, unfortunately, remains as a challenge unaddressed so far.

1.3 Contributions and Structure of the Dissertation

This dissertation is concerned with dynamic analysis of systems software—in particular, OS kernels—with the goal of finding vulnerabilities in them. Systems software such as OS kernels are written in low-level programming languages such as the C programming language.

Chapter 2 starts with a taxonomy of low-level vulnerabilities that might be present in software written in C and C++ (see Section 2.1). Next provided is an overview of existing dynamic analysis techniques for software written in C/C++—namely, (i) how to detect vulnerabilities at run time as they occur (see Section 2.2) and (ii) how to generate test inputs and drive dynamic analysis (see Section 2.3). The described dynamic analysis techniques are put into the context in Section 2.4, which discusses whether and how they can be applied to OS kernels. Chapter 2 is loosely based on the following conference article.

[157] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz; "**SoK: Sanitizing for Security.**" In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pp. 1275-1295. IEEE, May 2019.

Chapter 3 then presents a dynamic analysis and fuzzing framework that enables a thorough exploration of the peripheral input space of OS kernels from the perspective of a strong attacker on the peripheral side. The design of the framework contributes to the literature in that (i) it provides a new way to fuzz the peripheral interface of OS kernels, which received little attention compared to the system call interface, and (ii) it accurately models the capabilities of an attacker to uncover not only traditional memory corruption vulnerabilities, but also more subtle memory bugs such as double-fetch bugs. Chapter 3 is based on the following conference article.

[155] 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 *Proceedings of the 26th Network and Distributed System Security Symposium*. Internet Society, February 2019.

Next, Chapter 4 presents virtual machine checkpointing and checkpoint restoration techniques that can accelerate kernel-mode driver fuzzing. The core contribution here is the newly proposed primitive, the checkpointing primitive, which uses a dimension unutilized so far—similarities between test inputs generated by existing fuzzing algorithms—to accelerate kernel driver fuzzing. Chapter 4 is based on the following conference article.

[156] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent ByungHoon Kang, Jean-Pierre Seifert, and Michael Franz; "**Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints.**" In *Proceedings of the 29th USENIX Security Symposium*, pp. 2541-2557. USENIX Association, August 2020.

Chapter 5 concludes the dissertation with a summary of the presented techniques as well as a discussion of promising future work directions in the area of OS kernel fuzzing or dynamic analysis in general.

Chapter 2

Dynamic Analysis for C and C++

This chapter reviews existing dynamic analysis techniques for software written in C and C++, the programming languages that are most widely used to implement systems software such as OS kernels. The development of the C and C++ dynamic analysis literature has been centered around dynamic analysis of user-space C and C++ code. In a deliberate attempt to provide a more complete picture of the design space of dynamic analysis for C and C++, existing techniques and tools are comprehensively described in Section 2.2 and Section 2.3, including the tools that were originally developed for user-space code but not for kernel-space code. Note that the general discussion still applies to systems software running in kernel space. There are, however, some techniques and tools that need changing in one way or another for them to apply to kernel-space code; the unique elements of the kernel environment necessitating such changes are highlighted in Section 2.4.

2.1 Low-level Vulnerabilities in C and C++

Before describing techniques to find vulnerabilities, this section first provides, as a background, a taxonomy of low-level vulnerabilities that can be present in C and C++ code.

Listing 2.1 Intra-object overflow vulnerability which can be exploited to overwrite security-critical non-control data.

```
struct A { char name[7]; bool isAdmin; };
struct A a; char buf[8];
memcpy(/* dst */ a.name, /* src */ buf, sizeof(buf));
```

2.1.1 Memory Safety Violations

A program is *memory safe* if pointers in the program only access their *intended referents*, while those intended referents are valid. The intended referent of a pointer is the object from whose base address the pointer was derived. Depending on the type of the referent, it is either valid between its allocation and deallocation (for heap-allocated referents), between a function call and its return (for stack-allocated referents), between the creation and the destruction of its associated thread (for thread-local referents), or indefinitely (for global referents).

Memory safety violations are among the most severe security vulnerabilities and have been studied extensively in the literature [167, 171]. Their exploitation can lead to code injection [6], control-flow hijacking [154, 119, 152], privilege escalation [36], information leakage [163], and program crashes.

2.1.1.1 Spatial Safety Violations

Accessing memory that is not (entirely) within the bounds of the intended referent of a pointer constitutes a spatial safety violation. Buffer overflows are a typical example of a spatial safety violation. A buffer overflow happens when the program writes beyond the end of a buffer. If the intended referent of a vulnerable access is a subobject (e.g., a struct field), and if an attacker writes to another subobject within the same object, then we refer to this as an *intra-object overflow*. Listing 2.1 shows an intra-object overflow vulnerability which can be exploited to perform a privilege escalation attack.

Listing 2.2 Use-after-free vulnerability which can be exploited to hijack the control-flow of the program.

```
struct A { void (*func)(void); };
struct A *p = (struct A *)malloc(sizeof(struct A));
free(p);   // Pointer becomes dangling
...
p->func(); // Use-after-free
```

2.1.1.2 Temporal Safety Violations

A temporal safety violation occurs when the program accesses a referent that is no longer valid. When an object becomes invalid, which usually happens by explicitly deallocating it, all the pointers pointing to that object become *dangling*. Accessing an object through a dangling pointer is called a *use-after-free*. Accessing a local object outside of its scope or after the function returns is referred to as *use-after-scope* and *use-after-return*, respectively. This type of bug becomes exploitable when the attacker can reuse and control the freed region, as illustrated in Listing 2.2.

2.1.2 Use of Uninitialized Variables

Variables have an *indeterminate value* until they are initialized [77, 78]. C++14 allows this indeterminate value to propagate to other variables if both the source and destination variables have an unsigned narrow character type. Any other use of an uninitialized variable results in undefined behavior. The effects of this undefined behavior depend on many factors, including the compiler and compiler flags that were used to compile the program. In most cases, indeterminate values are in fact the (partial) contents of previously deallocated variables that occupied the same memory range as the uninitialized variable. As these previously deallocated variables may sometimes hold security-sensitive values, reads of uninitialized memory may be part of an information leakage attack, as illustrated in Listing 2.3.

Listing 2.3 Use of a partially-initialized variable which becomes vulnerable as the uninitialized value crosses a trust boundary.

```
struct A { int data[2]; };
struct A *p = (struct A *)malloc(sizeof(struct A));
p->data[0] = 0; // Partial initialization
send_to_untrusted_client(p, sizeof(struct A));
```

Listing 2.4 Bad-casting vulnerability leading to a type- and memory-unsafe memory access.

```
class Base { virtual void func(); };
class Derived : public Base { public: int extra; };
Base b[2];
Derived *d = static_cast<Derived *>(&b[0]); // Bad-casting
d->extra = ...; // Type-unsafe, out-of-bounds access, which
                // overwrites the vtable pointer of b[1]
```

2.1.3 Pointer Type Errors

C and C++ support several casting operators and language constructs that can lead memory accesses to misinterpret the data stored in their referents, thereby violating type safety. Pointer type errors typically result from unsafe casts. C allows all casts between pointer types, as well as casts between integer and pointer types. The C++ `reinterpret_cast` type conversion operator is similarly not subject to any restrictions. The `static_cast` and `dynamic_cast` operators do have restrictions. `static_cast` forbids pointer to integer casts, and casting between pointers to objects that are unrelated by inheritance. However, it does allow casting of a pointer from a base class to a derived class (also called *downcasting*), as well as all casts from and to the `void*` type. *Bad-casting* (often referred to as *type confusion*) happens when a downcast pointer has neither the run-time type of its referent, nor one of the referent's ancestor types. Listing 2.4 shows a bad-casting vulnerability.

To downcast safely, programmers must use the `dynamic_cast` operator, which performs run-time type checks and returns a null pointer if the check fails. Using `dynamic_cast` is entirely optional, however, and introduces additional run-time overhead.

Listing 2.5 Simplified version of CVE-2012-0809; user-provided input was mistakenly used as part of a larger format string passed to a `printf`-like function.

```c
char *fmt2; // User-controlled format string
sprintf(fmt2, user_input, ...);
// prints attacker-chosen stack contents if fmt2 contains
// too many format specifiers
// or overwrites memory if fmt2 contains %n
printf(fmt2, ...);
```

Type errors can also occur when casting between function pointer types. Again, C++'s `reinterpret_cast` and C impose no restrictions on casts between incompatible function pointer types. If a function is called indirectly through a function pointer of the wrong type, the target function might misinterpret its arguments, which leads to even more type errors. Finally, C also allows type punning through `union` types. If the program reads from a union through a different member object than the one that was used to store the data, the underlying memory may be misinterpreted. Furthermore, if the member object used for reading is larger than the member object used to store the data, then the upper bytes read from the union will take unspecified values.

2.1.4 Variadic Function Misuse

C/C++ support *variadic functions*, which accept a variable number of *variadic* function arguments in addition to a fixed number of regular function arguments. The variadic function's source code does not specify the number or types of these variadic arguments. Instead, the fixed arguments and the function semantics encode the expected number and types of variadic arguments. Variadic arguments can be accessed and simultaneously typecast using `va_arg`. It is, in general, impossible to statically verify that `va_arg` accesses a valid argument, or that it casts the argument to a valid type. This lack of static verification can lead to type errors, spatial memory safety violations, and uses of uninitialized values.

Listing 2.6 Simplified version of CVE-2017-5029; a signed integer overflow vulnerability that can lead to spatial memory safety violation.

```c
// newsize can overflow depending on len
int newsize = oldsize + len + 100;
newsize *= 2;
// The new buffer may be smaller than len
buf = xmlRealloc(buf, newsize);
memcpy(buf + oldsize, string, len); // Out-of-bounds access
```

2.1.5 Other Vulnerabilities

There are other operations that may pose security risks in the absence of type and memory safety. Notable examples include overflow errors which may be exploitable when such values are used in memory allocation or pointer arithmetic operations. If an attacker-controlled integer value is used to calculate a buffer size or an array index, the attacker could overflow that value to allocate a smaller buffer than expected (as illustrated in Listing 2.6), or to bypass existing array index checks, thereby triggering an out-of-bounds access. C/C++ do not define the result of a signed integer overflow, but stipulate that unsigned integers wrap around when they overflow. However, this wrap-around behavior is often unintended and potentially dangerous.

Undefined behaviors such as signed integer overflows pose additional security risks when compiler optimizations are enabled. In the presence of potential undefined behavior, compilers are allowed to assume that the program will never reach the conditions under which this undefined behavior is triggered. Moreover, the compiler can perform further optimization based on this assumption [94]. Consequently, the compiler does not have to statically verify that the program is free of potential undefined behavior, and the compiler is not obligated to generate code that is capable of recognizing or mitigating undefined behavior. The problem with this rationale is that optimizations based on the assumption that the program is free from undefined behavior can sometimes lead the compiler to omit security checks. In CVE-2009-1897, for example, GCC infamously omitted a null pointer check from one of the Linux kernel drivers, which led to a privilege escalation vulnerability [117]. Compiler developers regularly add such

Listing 2.7 Simplified version of CVE-2009-1897; dereferencing a pointer lets the compiler safely assume that the pointer is non-null.

```c
struct sock *sk = tun->sk; // Compiler assumes tun is not
                           // a null pointer
if (!tun) // Check is optimized out
  return POLLERR;
```

aggressive optimizations to their compilers. Some people therefore refer to undefined behavior as *time bombs* [51].

2.2 Vulnerability Detection

This section reviews dynamic analysis techniques that can detect the low-level vulnerabilities described in Section 2.1 as they occur and manifest at run time.

2.2.1 Memory Safety Violations

There are two types of techniques for detecting memory safety violations—dereferences of pointers that either do not target their intended referent (i.e., spatial safety violations), or that target a referent that is no longer valid (i.e., temporal safety violations). Their high-level goals and properties are summarized, followed by an in-depth discussion of the techniques existing tools employ to detect memory safety bugs.

Location-based Access Checkers. Location-based access checkers detect memory accesses to invalid memory regions. These checkers have a metadata store that maintains state for each byte of (a portion) of the addressable address space, and consult this metadata store whenever the program attempts to access memory to determine whether the memory access is valid or not. Location-based access checkers can use red-zone insertion [71, 151, 26, 70, 149] or guard pages [135, 108] to detect spatial safety violations. Either of these techniques can be combined with reuse delay to additionally detect temporal safety violations [71, 83, 151, 26, 149, 135, 108, 50, 46]. Location-based access checkers incur low run-time performance overheads, and are highly compatible with uninstrumented code. The downside is that these tools are imprecise, as they can only detect if an instruction accesses valid memory, but *not if the accessed memory is part of the intended referent of the instruction*. These tools generally incur high memory overhead.

Identity-based Access Checkers. Identity-based access checkers detect memory accesses that do not target their intended referent. These tools maintain metadata (e.g., bounds or allocation status) for each allocated memory object, and have a mechanism in place to determine the intended referent for every pointer in the program. Metadata lookups can happen when the program calculates a new pointer using arithmetic operations to determine if the calculation yields a valid pointer and/or upon pointer dereferences to determine if the dereference accesses the intended referent of the pointer. Identity-based access checkers can use per-object bounds tracking [83, 143, 49, 5, 56, 184, 53, 52] or per-pointer bounds tracking [87, 158, 10, 131, 114, 76, 118, 81, 179, 120, 88, 27] to detect spatial safety violations, and can be extended with reuse delay [27], lock-and-key checking [10, 131, 115], or with dangling pointer tagging [28, 92, 183, 170] to detect temporal safety violations. Identity-based checkers are more precise than location-based access checkers, as they cannot just detect accesses to invalid memory, but also accesses to valid memory outside of the intended referent. These tools do, however, incur higher run-time performance overhead than location-based checkers. Identity-based checkers are generally not compatible with uninstrumented code. They also have higher false positive detection rates than location-based checkers.

2.2.1.1 Spatial Memory Safety Violations

Red-zone Insertion. Location-based access checkers can insert so-called *red-zones* between memory objects [71, 151, 26, 70, 149]. These red-zones represent out-of-bounds memory and are marked as invalid memory in the metadata store. Any access to a red-zone or to an unallocated memory region triggers a warning. Purify was the first tool to employ this technique [71]. Purify inserts the red-zones at the beginning and the end of each allocation. Purify tracks the state of the program's allocated address space using a large shadow memory bitmap that stores two bits of state per byte of memory. Valgrind's Memcheck uses the same technique but reserves two bits of state for every bit of memory [151]. Consequently, Memcheck can detect access errors with bit-level precision, rather than byte-level precision.

Light-weight Bounds Checking (LBC) similarly inserts red-zones, but adds a fast path to the location-based access checks to reduce the overhead of the metadata lookups [70]. LBC does this by filling the red-zones with a random pattern and compares the data read/overwritten by every memory access with the fill pattern. If the data does not match the fill pattern, the access is considered safe because it could not have targeted a red-zone. If the data does happen to match the fill pattern, LBC performs a secondary slow path check that looks up the state of the accessed data in the metadata store, and triggers a warning if the accessed data is a red-zone.

Location-based access checkers that use red-zone insertion generally incur low run-time performance overhead, but have limited precision as they can only detect illegal accesses that target a red-zone. Illegal accesses that target a valid object, which may or may not be part of the same allocation as the intended referent, cannot be detected. Red-zone insertion-based tools also fail to detect intra-object overflow bugs because they do not insert red-zones between subobjects. While technically feasible, inserting red-zones between subobjects would lead to excessive memory overhead and it would change the layout of the parent object. Any code that accesses the parent object or one of its subobjects would therefore have to be modified, which would also break compatibility with external code that is not aware of the altered data layout.

Guard Pages. Location-based access checkers can insert inaccessible guard pages before and/or after every allocated memory object [135, 108]. Out-of-bound reads and writes that access a guard page trigger a page fault, which in turn triggers an exception in the application. This use of the paging hardware to detect illegal accesses allows location-based access checkers to run without instrumenting individual load and store instructions. Using guard pages does, however, incur high memory overhead, making the technique impractical for applications with large working sets. Microsoft recognized this problem and added an option to surround memory objects with guard blocks instead of full guard pages in PageHeap [108]. PageHeap fills these guard blocks with a fill pattern, and verifies that the pattern is still present when a memory object is freed. This technique is strictly inferior to red-zone insertion, as it only detects out-of-bounds writes (and not reads), and it does not detect the illegal writes until the overwritten object is freed.

Per-pointer Bounds Tracking. Identity-based access checkers can store bounds metadata for every pointer [87, 158, 10, 131, 114, 76, 118, 81, 179, 120, 88, 27]. Whenever the program creates a pointer by calling `malloc` or by taking the address of an object, the tracker stores the base and size of the referent as metadata for the new pointer. The tracker propagates this metadata when the program calculates new pointers through arithmetic and assignment operations. Spatial memory safety violations are detected by instrumenting all pointer dereferences and checking if a pointer is outside of its associated bounds when it is dereferenced.

Identity-based access checkers that use per-pointer bounds tracking can provide complete spatial memory violation detection, including detection of intra-object overflows. SoftBound [114] and Intel Pointer Checker [76] detect intra-object overflows by narrowing the pointer bounds to the bounds of the subobject whenever the program derives a pointer from the address of a subobject (i.e., a struct field). The primary disadvantage of per-pointer bounds tracking is poor compatibility, as the program generally cannot pass pointers to uninstrumented libraries because such libraries do not propagate or update bounds information correctly. Another disadvantage is that per-pointer metadata propagation adds high run-time overheads. CCured reduces this overhead by identifying *safe* pointers, which can be excluded from bounds checking and metadata propagation [118]. However, even with such optimizations, per-pointer bounds checking remains expensive without hardware support [48].

Per-object Bounds Tracking. Identity-based access checkers can also store bounds metadata for every memory object, rather than for every pointer [83, 143, 49, 5, 56, 184, 53, 52].

This approach—pioneered by Jones and Kelly—solves some of the compatibility issues associated with per-pointer bounds tracking [83]. Per-object bounds trackers can maintain bounds metadata without instrumenting pointer creation and assignment operations. The tracker only needs to intercept calls to memory allocation (i.e., `malloc`) and deallocation (i.e., `free`) functions, which is possible even in programs that are not fully instrumented. Since bounds metadata is maintained only for objects and not for pointers, it is difficult to link pointers to their intended referent. While the intended referent of an in-bounds pointer can be found using a range-based lookup in the metadata store, such a lookup would not return the correct metadata for an out-of-bounds (OOB) pointer. Jones and Kelly therefore proposed to instrument pointer arithmetic operations, and to invalidate pointers as they go OOB. Any subsequent dereference triggers a fault, which can then be caught to output a warning.

Jones and Kelly's approach, however, breaks many existing programs that perform computations using OOB pointers. In light of this, CRED supports the creation and manipulation of OOB pointers by tracking their referent information [143]. CRED links OOB pointers to so-called *OOB objects* which store the address of the original referent for each OOB pointer.

Baggy Bounds Checking (BBC) eliminates the need to allocate dedicated OOB objects by storing the distance between the OOB pointer and its referent into the pointer's most significant bits [5]. Tagging the most significant bits also turns OOB pointers into invalid user-space pointers, such that dereferencing them causes a fault. BBC compresses the size of the per-object metadata by rounding up all allocation sizes to the nearest power of two, such that one byte of metadata suffices to store the bounds.

Low-fat pointer (LFP) bounds checkers improve BBC by making allocation sizes configurable, which results in lower performance and memory overheads [53, 52]. The idea is to partition the heap into equally-sized subheaps that each supports only one allocation size. Thus, the allocation size for any given pointer can be obtained by looking up the allocation size supported by that heap. The base address of the pointer's referent can be calculated by rounding it down to the allocation size. LFP also differs from BBC in handling of OOB pointers. For better compatibility with uninstrumented libraries, LFP does not manipulate the pointer representation to encode the referent of an OOB pointer. Instead, LFP recomputes the referent of each pointer whenever the pointer is given to a function as input either explicitly (e.g., a pointer given as an argument) or implicitly (e.g., a pointer loaded from memory). However, this requires LFP to enforce an invariant that pointers are within bounds whenever they are given as input to other functions, which is likely too strict for real-world programs.

Per-object bounds trackers can support near-complete spatial safety vulnerability detection. However, techniques such as BBC and LFP do sacrifice precision for better run-time performance, as they round up allocation sizes and check allocation bounds rather than object bounds.

Per-object bounds tracking has other downsides too. First, per-object bounds trackers do not detect intra-object overflows. Second, marking pointers as OOB by pointing them to an OOB object, or by writing tags into their upper bits might impact compatibility with external code that is unaware of the bounds checking scheme used in the program. Specifically, external code is unable to restore OOB pointers into in-bounds pointers even when that is the intent.

2.2.1.2 Temporal Memory Safety Violations

Reuse Delay. Location-based access checkers can mark recently deallocated objects as invalid in the metadata store by replacing them with red-zones [71, 83, 151, 26, 149] or with guard pages [135, 108, 50, 46]. Identity-based checkers can similarly invalidate the identity of deallocated objects [27]. The existing access checking mechanism can then detect dangling pointer dereferences as long as the deallocated memory or identity is not reused. If the program does reuse the memory or identity for new allocations, this approach will erroneously allow dangling pointer dereferences to proceed. Some reuse delay-based tools reduce the chance of such detection failures by delaying the reuse of memory regions or identities until they have *aged* [71, 83, 151, 26, 149, 27]. This leads to a trade-off between precision and memory overhead as longer reuse delays lead to higher memory overhead, but also to a higher chance of detecting dangling pointer dereferences.

Dhurjati and Adve proposed to use static analysis to determine exactly when deallocated memory is safe to reuse [50]. They allocate every memory object on its own virtual memory page, but allow objects to share physical memory pages by aliasing virtual memory pages to the same physical page. When the program frees a memory object, Dhurjati and Adve convert virtual page into a guard page. They also partition the heap into pools, leveraging a static analysis called Automatic Pool Allocation [91]. This analysis can infer when a pool is no longer accessible (even through potentially dangling pointers), at which point all virtual pages in the pool can be reclaimed. Dang et al. proposed a similar system that does not use pool allocation, and can therefore be applied to source-less programs [46]. Similar to Dhurjati and Adve's approach, Dang et al. allocate all memory objects on their own virtual pages. Upon deallocation of an object, Dang et al. unmap that object's virtual page. This effectively achieves the same goal as guard pages, but allows the kernel to free its internal metadata for the virtual page, which reduces the physical memory overhead. To prevent reuse of unmapped virtual pages, Dang et al. propose to map new pages at the high water mark (i.e., the highest virtual address that has been used in the program). While this does not rule out reuse of unmapped virtual pages completely, the idea is that reuse is unlikely to happen given a 64-bit address space.

Lock-and-key. Identity-based checkers can detect temporal safety violations by assigning unique allocation identifiers—often called keys—to every allocated memory object and by storing this key in a *lock location* [10, 131, 115]. They also store the lock location and the expected key as metadata for every pointer. The checker revokes the key from the lock location when its associated object is deallocated. Lock-and-key checking detects temporal memory safety violations when the program dereferences a pointer whose key does not match the key stored in the lock location for that pointer. Assuming unique keys, this approach provides complete coverage of temporal safety violations [115]. Since this technique stores per-pointer metadata, it naturally complements identity-based checkers that also detect spatial violations using per-pointer bounds tracking. The drawbacks of lock-and-key checking are largely the same as those for per-pointer bounds tracking: compatibility with uninstrumented code is poor because uninstrumented code will not propagate the metadata correctly, and the run-time overhead is significant because maintaining metadata for every pointer is expensive.

Dangling Pointer Tagging. The most straightforward way to tag dangling pointers is to nullify either the value or the bounds associated with pointers that are passed to the `free` function [76]. A spatial memory safety violation detection mechanism would then trigger a warning if such pointers are dereferenced at a later point in time. The disadvantage of this approach is that it does not tag copies of the dangling pointer, which may also be used later.

Several tools tag not only the pointer passed to `free`, but also copies of that pointer by maintaining auxiliary data structures that link all memory objects to any pointers that refer to them [28, 92, 183, 170]. Undangle uses taint tracking [121, 164, 41] to track pointer creations, and to maintain an object-to-pointer map [28]. Whenever the program deallocates a memory object, Undangle can query this pointer map to quickly find all dangling pointers to the now deallocated object. Undangle aims to report not only the use but also the existence of dangling pointers. It has a configurable time window where it considers dangling pointers latent but not unsafe, e.g., a transient dangling pointer that appears during the deallocation of nested objects. Undangle reports a dangling pointer when this window expires, or earlier if the program attempts to dereference the pointer.

DangNull [92], FreeSentry [183], and DangSan [170] steer clear of taint tracking and instrument pointer creations at compile time instead. These tools maintain pointer maps by calling a runtime registration function whenever the program assigns a pointer. Whenever the program deallocates a memory object, the tools look up all pointers that point to the object being deallocated, and invalidate them. Subsequent dereferences of an invalidated dangling pointer result in a hardware trap.

Dangling pointer tagging tools that are not based on taint tracking have some fundamental limitations. First, they require the availability of source as it relies on precise type information to determine which operations store new pointers. Second, they fail to maintain accurate metadata if the program copies pointers in a type-unsafe manner (e.g., by casting them to integers). Third, and most importantly, they can only link objects to pointers stored in memory, and is therefore unaware of dangling pointers stored in registers. Taint tracking-based tools such as Undangle, have none of these disadvantages, but incur significant performance and memory overheads.

2.2.2 Use of Uninitialized Variables

Uninitialized Memory Read Detection. Location-based access checkers can be extended to detect reads of uninitialized memory values by marking all memory regions occupied by newly allocated objects as uninitialized in the metadata store [71]. These tools instrument read instructions to trigger a warning if they read from uninitialized memory regions, and they instrument writes to clear the uninitialized flag for the overwritten region. Note that marking memory regions as uninitialized is not equivalent to marking them as a red-zone, since both read and write accesses to red-zones should trigger a warning, whereas accesses to uninitialized memory should only be disallowed for reads.

Uninitialized Value Use Detection. Detecting reads of uninitialized memory yields many false positive detections, as the C++14 standard explicitly allows uninitialized values to propagate through the program as long as they are not used. This happens, for example, when copying a partially uninitialized struct from one location to the other. Memcheck attempts to detect only the uses of uninitialized values by limiting error reporting to (i) dereferences of pointers that are (partially) undefined, (ii) branching on a (partially) undefined value, (iii) passing undefined values to system calls, and (iv) copying uninitialized values into floating point registers [151]. To support this policy, Memcheck adds one byte of shadow state for every partially initialized byte in the program memory. This allows Memcheck to track the definedness of all of the program's memory with bit-level precision. Memcheck approximates the C++14 semantics but produces false negatives (failing to report illegitimate uses of uninitialized memory) and false positives (reporting legitimate uses of uninitialized memory), which are unavoidable given that Memcheck operates at the binary level, rather than the source code level. MemorySanitizer (MSan) implements fundamentally the same policy as Memcheck, but instruments programs at the compiler Intermediate Representation (IR) level [159]. The IR code carries more information than binary code, which makes MSan more precise than Memcheck. MSan produces no false positives (provided that the entire program is instrumented) and few false negatives. Its performance overhead is also an order of magnitude lower than Memcheck.

2.2.3 Pointer Type Errors

Pointer Casting Monitor. Pointer casting monitors detect illegal downcasts through the C++ `static_cast` operator. Illegal downcasts occur when the target type of the cast is not equal to the run-time type (or one of its ancestor types) of the source object. UndefinedBehaviorSanitizer [101] (UBSan) and Clang CFI [99] include checkers that verify the correctness of `static_cast` operations by comparing the target type to the run-time type information (RTTI) associated with the source object. This effectively turns `static_cast` operations into `dynamic_cast`. The downside is that RTTI-based tools cannot verify casts between non-polymorphic types that lack RTTI.

CaVer [93] and TypeSan [67] do not rely on RTTI to track type information, but instead maintain metadata for all types and all objects used in the program. This way, they can extend the type-checking coverage to non-polymorphic types. At compile time, these tools build per-class type metadata tables which contain all the valid type casts for a given pointer type. The type tables encode the class inheritance and composition relationships. Both tools also track the effective run-time types for each live object by monitoring memory allocations and storing the allocated types in a metadata store. To perform downcast checking, the tools retrieve the run-time type of the source object from the metadata store, and then query the type table for the corresponding class to check if the type conversion is in the table (and is therefore permissible). HexType similarly tracks type information in disjoint metadata structures, but provides a more accurate run-time type tracing [79]. HexType also replaces the default implementation of `dynamic_cast` with its own optimized implementation, while preserving its run-time semantics, i.e., returning NULL for failing casts.

Pointer Use Monitor. C/C++ support several constructs to convert pointer types in potentially dangerous ways; C-style casts, `reinterpret_cast`, and unions can all be used to bypass compile-time and run-time type checking. Extending pointer casting monitoring to these constructs can result in false positives, however. This is because programmers can legitimately use such constructs as the language standard allows it. For this reason, one might opt for pointer dereference/use monitoring over pointer casting monitoring.

Loginov et al. proposed a pointer use monitor for C programs [102]. The tool maintains and verifies run-time type tags for each memory location by monitoring load and store operations. A tag contains the scalar type that was last used to write to its corresponding memory location. Aggregate types are supported by breaking them down into their scalar components. The tool stores the tags in shadow memory. Whenever a value is read from memory, the tool checks if the type used to load the value matches the type tag.

LLVM Type Sanitizer (TySan) also maintains a type tag store in shadow memory and verifies the correctness of load instructions [57]. Contrary to Loginov et al.'s tool, however, TySan does not require that the types used to store and load from a memory location match exactly. Instead, TySan only requires type compatibility, as defined by the aliasing rule in the C/C++ standard. TySan leverages metadata generated by the compiler frontend (Clang) which contains the aliasing relationship between types. This metadata is used at run time to allow, for example, all loads through a character pointer type, even if the target location was stored using a pointer to a larger type. Loginov et al.'s tool would detect this as an error, but this behavior is explicitly permitted by the language standard.

EffectiveSan is another pointer use monitor that performs type checks as well as bounds checks at pointer uses [54]. EffectiveSan instruments each allocation site to tag each allocated object with its statically-determined type. It uses declared variable types for stack and global variables, as well as objects allocated using the C++ `new` operator. For objects allocated using `malloc`, it uses the type of the first lvalue usage of the object. EffectiveSan also generates type layout metadata at compile time, which contains the layout information of all nested subobjects for each type. At every pointer dereference, both type compatibility and object bounds are checked, using the object type tag in conjunction with the type layout metadata. EffectiveSan's bounds checking supports detection of intra-object overflows by using type layout information to derive subobject bounds at run time.

Several tools also detect pointer type errors in indirect function calls, that is, calling functions through a pointer of a type incompatible with the type of the callee [101, 130, 99]. Function-signature-based forward-edge control flow integrity mechanisms such as Clang CFI [99] can be viewed as bug finding techniques that detect such function pointer misuses. Since all the function signatures are known at compile time, these tools can detect mismatches between the pointer type and the function type without maintaining run-time tags.

2.2.4 Variadic Function Misuse

Dangerous Format String Detection. The most prominent class of variadic function misuse bugs are format string vulnerabilities. Most efforts therefore focus solely on detection of dangerous calls to `printf`. Among these efforts are tools that restrict the usage of the `%n` qualifier in the format string [169, 141]. This qualifier may be used to have `printf` write to a caller-specified location in memory. However, this dangerous operation [32] is specific to the `printf` function, so the aforementioned tools' applicability is limited.

Argument Mismatch Detection. FormatGuard prevents `printf` from reading more arguments than were passed by the caller [45]. FormatGuard does this by redirecting the calls to a protected `printf` implementation that increments a counter whenever it retrieves a variadic argument through `va_arg`. If the counter surpasses the number of arguments specified at the call site, FormatGuard raises an alert. HexVASAN generalizes argument counting to all variadic functions, and also adds type checking [20]. HexVASAN instruments the call sites of variadic functions to capture the number and types of arguments passed to the callee and saves this information in a metadata store. The tool then instruments `va_start` and `va_copy` operations to retrieve information from the metadata store, and it instruments `va_arg` operations to check if the argument being accessed is within the given number of arguments and of the given type.

2.2.5 Other Vulnerabilities

Stateless Monitoring. UndefinedBehaviorSanitizer (UBSan) is a dynamic tool that detects undefined behavior that was not covered so far [101]. The undefined behaviors UBSan detects include signed integer overflows, floating point or integer division by zero, invalid bitwise shift operations, floating point overflows caused by casting (e.g., casting a large double-precision floating point number to a single-precision float), uses of misaligned pointers, performing an arithmetic operation that overflows a pointer, dereferencing a null pointer, and reaching the end of a value-returning function without returning a value. Most of UBSan's detection features are stateless, so they can be turned on collectively without interfering with each other. UBSan can also detect several kinds of well-defined but likely unintended behavior. For example, the language standard dictates that unsigned integers wrap around when they overflow. This well-defined behavior is often unexpected and a frequent source of bugs, however, so UBSan can optionally detect these unsigned integer wrap-arounds.

2.3 Test Input Generation

Dynamic analysis tools, often called *sanitizers,* only detect bugs on code paths that execute while executing test inputs. Increasing code coverage therefore increases vulnerability finding opportunities. Program execution can be driven by unit or integration test suites, automated test case generators, alpha and beta testers, or any combination thereof.

Unit testing and integration testing are already best practices in software engineering. Writing these tests has traditionally been a manual process. While indispensable in general, using hand-written tests does have some drawbacks when used to find bugs in a program. First, manually written tests often focus on positive testing using *valid* inputs to check expected behavior. Security bugs are typically exploited by feeding the program *invalid* inputs, however. Second, manually written tests hardly ever cover *all* code paths.

Developers can use automated test case generators to alleviate these problems. One option is to use symbolic execution, which systematically explores all possible execution paths to generate concrete program inputs [148, 61, 30, 29]. These inputs can then be fed into dynamic analysis tools to find bugs in a program. However, this approach in general does not scale due to the path explosion problem and the cost of constraint solving. A more scalable option is to run a fuzzer on the program under dynamic analysis [104]. Fuzzers are testing tools that run programs on automatically generated inputs, typically using light-weight dynamic program analysis such as coverage feedback. Fuzzers perform negative testing, because they tend to provide invalid inputs to the program, and can find security bugs relatively quickly, especially if the bugs are triggered on code paths that are easily accessible.

Finally, developers can ship sanitization-enabled programs to beta testers and to collect and transmit any sanitizer output back to the developer. The main advantage here is that beta testers can distribute the testing load, therefore allowing developers to locate bugs more quickly. One disadvantage is that beta testers will inadvertently focus on testing the program's main usage scenarios. Another disadvantage is that sanitizers can slow down the program to the point where it becomes unusable, thus reducing the chance that beta testers will thoroughly test the programs. Lettner et al. [96] demonstrated that *partitioned sanitization*, where sanitization is turned on and off at run time based on criteria such as coverage and execution speed, can alleviate this concern to the extent that sanitizers compose.

2.4 Finding OS Kernel Vulnerabilities

Many contemporary OSs have their kernels written in C, and, therefore, the low-level vulnerabilities discussed in Section 2.1 may be present in their kernel code. This also means that both the vulnerability detection and input generation techniques discussed in Section 2.2 and Section 2.3, respectively, can also be used to find OS kernel vulnerabilities. There are, however, some constraints or practices that arise in the kernel environment, and do not exist in a typical user-space environment. These differences may require the techniques presented earlier in this chapter be adapted for them to apply in kernel space. This section provides a general discussion of why and how finding vulnerabilities in OS kernels differs from doing so for user-space programs.

2.4.1 Vulnerability Detection

Kernel API. The C and C++ programming language standard defines a set of *library functions* that developers can use when writing user-space programs. Kernel APIs, however, do not necessarily provide the same set of library functions defined in the language standard. A kernel API may not provide some of the C/C++ library functions that are not frequently used in kernel code; some of the library functions may be provided under a different name; for some functionalities frequently used in the kernel environment, such as concurrency control or memory allocation primitives, a kernel API may provide functions tailored for different purposes, even though they do not correspond to any of the C/C++ library functions.

In general, it is straightforward to take into account these API differences when applying analysis techniques developed for user space to kernel code. For example, KernelAddressSanitizer (KASAN) [64] and KernelMemorySanitizer (KMSAN) [65], the kernel-space equivalent of AddressSanitizer (ASan) and MemorySanitizer (MSan), respectively, use the same core techniques and high-level bug detection policies as their user-space equivalent; like ASan, to detect spatial memory safety errors in the Linux kernel, KASAN creates red-zones between objects when they are allocated, and guards every memory access with a check that ensures the access goes to valid memory and not to the red-zones. Although different OS kernels define their own set of *memory allocators* [43, 109] that are different from the C/C++ library functions, this only requires a different set of allocators be instrumented. That is, as long as the memory allocators in kernel space can be identified and instrumented, the same technique can detect OS kernel vulnerabilities.

Kernel-specific Language Semantics. For some dynamic analysis techniques developed for user space to apply in kernel space, however, a significant change in their high-level bug detection algorithms or policies may be required. The C and C++ language standard is sometimes not followed by the kernel, due to performance or hardware constraints that arise in the kernel programming environment. For example, the Linux memory model that describes the ordering requirements between memory accesses [107] slightly deviates from what is specified by the language standard. A possible implication of this is that different bug detection policies may need to be designed specifically for kernel code, because a code snippet that constitutes a bug in a user-space program may not constitute a bug in kernel code. To apply user-space bug detection techniques to kernel code, if the techniques make assumptions about the memory model, one needs to take into account the kernel's memory model. For example, for data race bug detection, the difference between existing Linux kernel code following the Linux kernel memory model and the language standard may require a change either in the kernel code or bug detection techniques [174].

2.4.2 Test Input Generation

Different methods of generating test inputs described in Section 2.3 can be applied to both user and kernel space. For instance, developers can still manually write test inputs for OS kernels, and testers (or users) can help generate test inputs by actually using the system deploying the OS kernel under test. There are, however, aspects of the OS kernels that need to be considered for high-quality test input generation, which are introduced and summarized in this section.

Multi-Dimensional Input Space. The input space of OS kernels can broadly be classified into two: the system call interface and the peripheral interface, as described in Section 1.1. The existence of *multiple* input spaces may complicate test input generation. When it comes to an adversarial analysis of OS kernels, in particular, one may need to consider and evaluate threat models that involve multiple input spaces, since an attacker may have access to multiple input spaces. Depending on the threat model which defines the set of input spaces that potential adversaries have access to, multi-dimensional test inputs—test inputs that can exercise the code paths of multiple input spaces—may have to be generated for an adversarial analysis of OS kernels.

Event-driven, Stateful Execution. OS kernels can be viewed as an event-driven, reactive system; the execution of an OS kernel is driven by *events* such as system calls (or, more generally, events received from the system call interface) and interrupts (or events received from the peripheral input space). In reaction to the events received, the internal state of the kernel transitions from one to another. The kernel's reaction to a given event is determined by the event as well as its current state. This means that, depending on the current state, the kernel may react differently to the same given event.

This *statefulness* of OS kernels makes several key differences when it comes to high-quality input generation. To exercise deeper code paths of a stateful OS kernel, test inputs need to contain *sequences of events*, taking into account the dependencies between the events. For instance, a system call `write` may only exercise the actual file writing code paths, when its first argument is a valid file descriptor opened (and not closed afterwards) by the process using a system call `open` that precedes the `write` system call. Several automated test input generation algorithms that are oriented towards generating random sequences of bytes may not be effective at triggering deeper code paths in OS kernels.

Chapter 3

Dynamic Analysis of Peripheral Input Space

This chapter presents a technique that facilitates the exploration of the peripheral input space exposed by device drivers running in an OS kernel. Among different hardware-OS interaction mechanisms that is used to implement the peripheral input space, the framework presented in this chapter focuses on the Peripheral Component Interconnect (PCI), and, more specifically, the peripheral input space exposed by the following two mechanisms: *memory-mapped I/O* (MMIO) and *direct memory access* (DMA).

The key idea is to monitor MMIO or DMA mappings set up by the driver, and then dynamically *trap* the driver's accesses to such memory regions. Section 3.3 presents a generic analysis framework designed based on this idea, which enables various kinds of analysis of the peripheral input space. More specifically, the framework is designed to allow its users to register their own hooks that are called upon each trapped access. This effectively enables them to conduct a *fine-grained*, thus precise analysis of the peripheral input space. For example, one can implement hooks that record and/or mutate hardware-OS interactions in support of reverse engineering, record-and-replay, fuzzing, etc.

To showcase versatility of this analysis framework, a kernel driver fuzzing framework is designed on top of the analysis framework, which is presented in Section 3.4. This fuzzing framework can be used to find vulnerabilities that can be exploited in OS kernel attacks originating in untrustworthy peripherals. The evaluation presented in Section 3.6 shows that using this fuzzing framework, different classes of vulnerabilities, e.g., memory safety violations, double-fetch bugs, and kernel pointer disclosure, can be found in real-world device drivers.

3.1 Motivation

Peripheral Attacks. The trust boundary that separates peripheral subsystems from kernel drivers has drawn a great deal of interest from security researchers in recent years. This is because peripheral devices may provide a remote attack vector (e.g., network devices may receive malicious packets over the air), and they typically lack basic defense mechanisms. Consequently, peripheral devices have frequently fallen victim to *remote* exploitation [55, 176, 7, 16, 18, 31]. Moreover, several recently published attacks demonstrated that peripheral compromise can be turned into full system compromise (i.e., remote kernel code execution) by coaxing a compromised device into generating specific outputs, which in turn trigger a vulnerability when processed as an input in a device driver [19, 17].

Double-fetch Bugs. Vulnerabilities occurring along the peripheral attack surface can be subtle. With a compromised device, any value read from an I/O mapping should be considered to be under an attacker's control. If values read from an I/O mapping are not properly sanitized, it can lead to various low-level vulnerabilities including memory safety violations. In addition to these traditional vulnerabilities, a more subtle class of bugs may exist; the attacker can freely modify the content of an I/O mapping at any time, even in between the driver's reads. If the driver reads the same memory location multiple times (i.e., overlapping fetches [178]) while the data can still be modified by the device, double-fetch bugs may be present [150, 84].

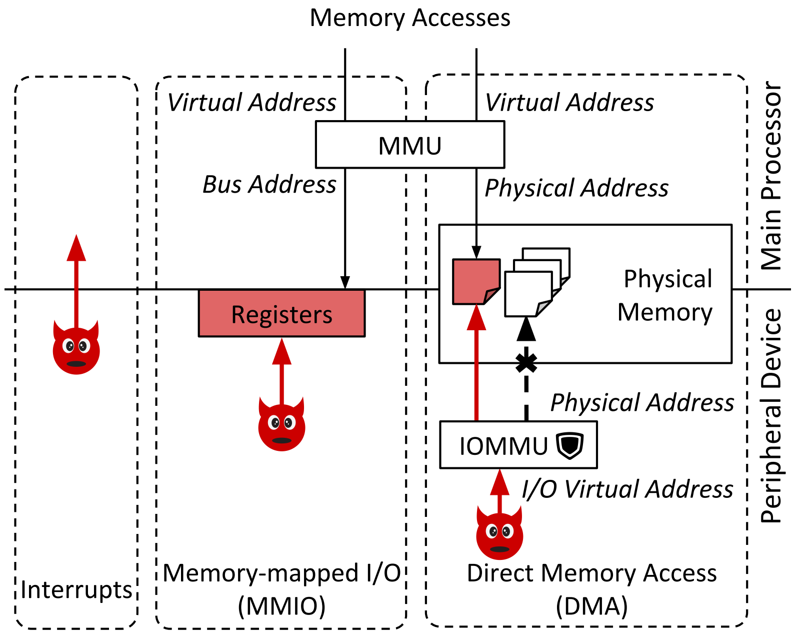


Figure 3.1: Hardware-OS interaction mechanisms.

3.2 Background

3.2.1 Hardware-OS Interaction Mechanisms

Figure 3.1 illustrates the various ways in which devices can interact with the OS and the device driver. Although the following description assumes that the device driver runs on a Linux system with an ARMv8-A/AArch64 CPU, the following discussion generally applies to other platforms as well.

Interrupts. A device can send a signal to the CPU by raising an interrupt request on one of the CPU's interrupt lines. Upon receiving an interrupt request, ARMv8-A CPUs first mask the interrupt line so that another interrupt request cannot be raised on the same line while the first request is being handled. Then, the CPU transfers control to the interrupt handler registered by the OS for that interrupt line. Interrupt handlers can be configured at any time, though the OS typically configures them at boot time.

Processing Interrupts. To maximize the responsiveness and concurrency of the system, the OS attempts to defer interrupt processing so that the interrupt handler can return control to the CPU as soon as possible. Typically, interrupt handlers only process interrupts in full if they were caused by time-sensitive events or by events that require immediate attention. All other events are processed at a later time, outside of the interrupt context. This mechanism is referred to as top-half and bottom-half interrupt processing in Linux lingo.

In Linux, after performing minimal amount of work in the hardware interrupt context (`hardirq`), the device driver schedules the work to be run in either software interrupt context (`softirq`), kernel worker threads, or the device driver's own kernel threads, based on its priority. For higher priority work, a device driver can register its own `tasklet`, a deferred action to be executed under the software interrupt context, which also ensures serialized execution. Lower priority work can further be deferred either to kernel worker threads (using the workqueue API) or to the device driver's own kernel threads.

Memory-Mapped I/O. Analogous to peripherals using interrupts to signal the OS and the device driver, the CPU uses memory-mapped I/O (MMIO) to signal peripherals. MMIO maps a range of kernel-space virtual addresses to the hardware registers of peripheral devices. This allows the CPU to use normal memory access instructions (as opposed to special I/O instructions) to communicate with the peripheral device. The CPU observes such memory accesses and redirects them to the corresponding hardware. In Linux, device drivers call `ioremap` to establish an MMIO mapping, and `iounmap` to remove it.

Direct Memory Access. Direct memory access (DMA) allows peripheral devices to access physical memory directly. Typically, the device transfers data using DMA, and then signals the CPU using an interrupt. There are two kinds of DMA buffers: coherent and streaming.

Coherent DMA buffers (also known as consistent DMA buffers) are usually allocated and mapped only once at the time of driver initialization. Writes to coherent DMA buffers are usually uncached, so that values written by either the peripheral processor or the CPU are immediately visible to the other side.

Streaming DMA buffers are backed by the CPU's cache, and have an explicit owner. They can either be owned by the CPU itself, or by one of the peripheral processors. Certain kernel-space memory buffers can be *mapped* as streaming DMA buffers. However, once a streaming DMA buffer is mapped, the peripheral devices automatically acquires ownership over it, and the kernel can no longer write to the buffer. Unmapping a streaming DMA buffer revokes its ownership from the peripheral device, and allows the CPU to access the buffer's contents. Streaming DMA buffers are typically short-lived, and are often used for a single data transfer operation.

3.2.2 Input/Output Memory Management Unit

Since DMA allows peripherals to access physical memory directly, its use can be detrimental to the overall stability of the system if a peripheral device misbehaves. Modern systems therefore deploy an input output memory management unit (IOMMU) (also known as system memory management unit, or SMMU, on the ARMv8-A/AArch64 architecture) to limit which regions of the physical memory each device can access. Similar to the CPU's memory management unit (MMU), the IOMMU translates device-visible virtual addresses (i.e., I/O addresses) to physical addresses. The IOMMU uses translation tables, which are configured by the OS prior to initiating a DMA transfer. Device-initiated accesses that fall outside of the translation table range will trigger faults that are visible to the OS.

3.3 I/O Monitoring Framework

This section presents the design of PeriScope, a dynamic analysis framework that can be used to examine bi-directional communication between devices and their drivers over MMIO and DMA. Contrary to earlier work on analyzing device-driver communication on the device side, PeriScope analyzes this communication on the driver side, by intercepting the driver's accesses to communication channels. PeriScope does this by hooking into the kernel's page fault handling mechanism. This design choice makes PeriScope driver-agnostic; PeriScope can analyze drivers with relative ease, regardless of whether the underlying device is virtual or real, and regardless of the type of the peripheral device.

At a high level, PeriScope works as follows. First, PeriScope automatically detects when the target device driver creates a MMIO or DMA memory mapping, and registers it. Then, the user of the framework selects the registered mappings that he/she wishes to monitor. PeriScope marks the pages backing these selected monitored mappings as not present in the kernel page tables. Any CPU access to those marked pages therefore triggers a page fault, even though the data on these pages is present in physical memory.

When a kernel page fault occurs, PeriScope first marks the faulting page as present in the page table (❶ in Figure 3.2). Then, it determines if the faulting address is part of any of the monitored regions (❷). If it is not, PeriScope re-executes the faulting instruction (❺), which will execute without triggering a page fault this time. Afterwards, PeriScope marks the page as not present again (❼), and resumes the normal execution of the faulting code.

If the faulting address does belong to a monitored region, PeriScope invokes a pre-instruction hook function registered by the user of the framework, passing information about the faulting instruction (❹). Then, PeriScope re-executes the faulting instruction (❺). Finally, PeriScope invokes the post-instruction hook registered by the driver (❻), marks the faulting page as not present again (❼), and resumes the execution of the faulting code.

Tracking Allocations. PeriScope hooks the kernel APIs used to allocate and deallocate DMA and MMIO regions[1]. PeriScope uses these hooks to maintain a list of all DMA/MMIO allocation contexts and their active mappings. PeriScope assigns an identifier to every context in which a mapping is allocated, and presents the list of all allocation contexts as well as their active mappings to privileged user-space programs through the `debugfs` file system.

[1]Establishing DMA and MMIO mappings is a highly platform-dependent process, so device drivers are obliged to use the official kernel APIs to do so.

Enabling Monitoring. PeriScope exposes a privileged user-space API that enables monitoring of DMA/MMIO regions on a per-allocation-context basis. Once monitoring is enabled for a specific allocation context, PeriScope will ensure that accesses to all current and future regions allocated in that context trigger page faults.

Clearing Page Presence. PeriScope marks all pages containing monitored regions as not present in the kernel's page tables to force accesses to such pages to trigger page faults. One complication that can arise here is that modern architectures, including x86-64 and AArch64, can support multiple page sizes within the same page table. On AArch64 platforms, a single page table entry can serve physical memory regions of 4KB, 16KB, or 64KB, for example. If a single (large) page table entry serves both a monitored and a non-monitored region, then we split that entry prior to marking the region as not present. This is done to avoid unnecessary page faults for non-monitored regions. Note that, even after splitting page table entries, PeriScope cannot rule out spurious page faults completely, as some devices support DMA/MMIO regions that are smaller than the smallest page size supported by the CPU.

Trapping Page Faults. PeriScope hooks the kernel's default kernel page fault handler to monitor page faults. Inside the hook function, we first check if the fault originated from a page that contains one of the monitored regions. If the fault originated from some other page, we immediately return from the hook function with an error code and defer the fault handling to the default page fault handler. If the fault did originate from a page containing a registered buffer, PeriScope marks that page as present (❶), and then checks if the faulting address falls within a monitored region (❷). If the faulting address is outside a monitored region, we simply single-step the faulting instruction (❺), mark the faulting page as not present again (❼), and resume normal execution of the faulting code. If the faulting address does fall within a monitored region, however, we proceed to the instruction decoding step (❸).

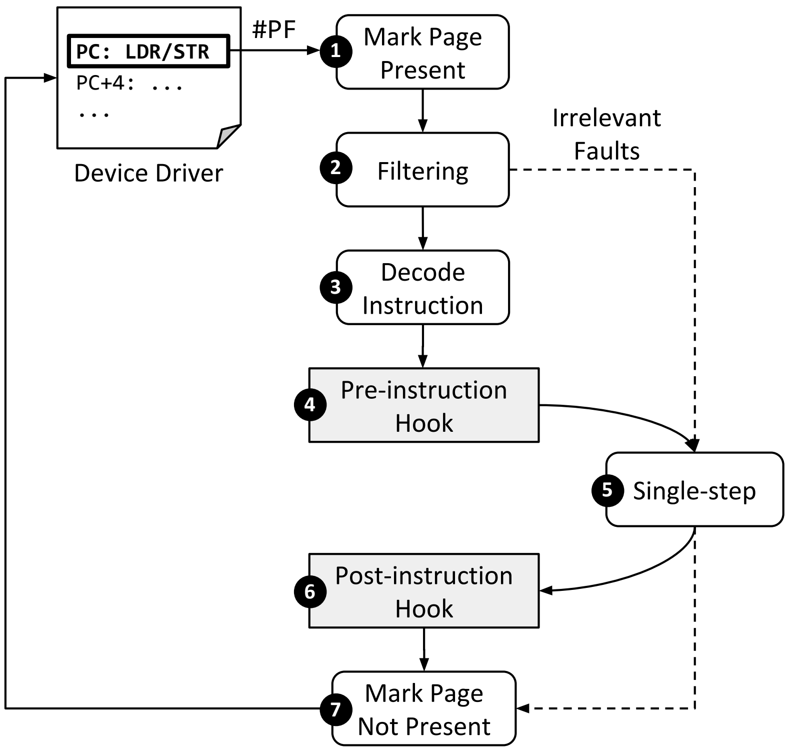


Figure 3.2: PeriScope fault handling.

Instruction Decoding. In order to accurately monitor and (potentially) manipulate the communication between the hardware/firmware and the device driver, we need to extract the source register, the destination register and the access width of the faulting instruction (❸ in Figure 3.2). We implemented a simple AArch64 instruction decoder, which provides this information for all load and store instructions. PeriScope carries this information along the rest of its fault handling pipeline.

Pre-instruction Hook. After decoding the instruction, PeriScope calls the pre-instruction hook that the user of the framework can register (❹). We pass the address of the faulting instruction, the memory region type (MMIO or DMA coherent/streaming), the instruction type (load or store), the destination/source register, and the access register width to this hook function. The pre-instruction hook function can return two values: a default value and a skip-single-step value. If the function returns the latter, PeriScope proceeds immediately to step ❻. Otherwise, PeriScope proceeds to step ❺.

PeriScope provides a default pre-instruction hook which logs all memory stores before the value in the source register is stored to memory. We maintain this log in a kernel-space circular buffer that can later be read from the file system using `tracefs`.

Single-stepping. When execution returns from the pre-instruction hook, and the hook function did not return the skip-single-step value, we re-execute the faulting instruction, which can now access the page without faulting. We use the processor's single-stepping support to ensure that only the faulting instruction executes, but none of its successors do (❺).

Post-instruction Hook. When PeriScope regains control after single-stepping, it first clears the page present flag for the faulting page again so that future accesses to the faulting page once again trigger a page fault. Then, it calls the post-instruction handler, which, similarly to the pre-instruction handler, has a default implementation that can be overridden through an API (❻). The default handler logs all memory loads by examining and logging the value that is now stored in the destination register.

3.4 I/O Fuzzing Framework

This section describes PeriFuzz, an I/O fuzzing framework that is designed as a client module of PeriScope. PeriFuzz enables an adversarial analysis of device drivers, as it provides a means to feed inputs to device drivers through their peripheral input space. The design goal of PeriFuzz is to uncover vulnerabilities that could potentially be exploited by a compromised peripheral device. More specifically, PeriFuzz's design assumes the following threat model.

3.4.1 Threat Model

Peripheral Compromise. PeriFuzz operates under the assumption that an attacker can compromise a peripheral device, which, in turn, can send arbitrary data to its device driver. Compromising a peripheral device is feasible because such devices rarely deploy hardware protection mechanisms or software mitigations. As a result, silent memory corruptions occur frequently [113], which significantly lowers the bar to mount an attack. That peripherals can turn malicious after being attacked was demonstrated by successful *remote* compromises of several network devices such as Ethernet adapters [55], GSM baseband processors [176, 31], and Wi-Fi processors [7, 16, 18].

IOMMU/SMMU Protection. For many years, a strict hardware-OS security boundary existed in theory, but it was not enforced in practice. Most device drivers *trusted* that the peripheral was benign, and gave the device access to the entire physical memory (provided that the device was DMA-capable), thus opening the door to DMA-based attacks and rootkits [9, 161]. This situation has changed for the better with the now widespread deployment of IOMMU units (or SMMU for AArch64). IOMMUs can prevent the device from accessing physical memory regions that were not explicitly mapped by the IOMMU, and they prevent peripherals from accessing streaming DMA buffers while these are mapped for CPU access. The latter restriction can be imposed by invalidating IOMMU mappings, or by copying the contents of a streaming DMA buffer to a temporary buffer (which the peripheral cannot access) before the CPU uses them [105, 106]. PeriFuzz's threat model assumes that such an IOMMU is in place, and that is being used correctly.

Summary. In this threat model, the attacker can (i) compromise a peripheral device such as a Wi-Fi chipset over the air by abusing an existing bug in the peripheral's firmware, (ii) *exercise control* over the compromised peripheral to send arbitrary data to the device driver, and, (iii) not access the main physical memory, except for memory regions used for communicating with the device driver.

3.4.2 Framework Overview

PeriFuzz is composed of three key components, as depicted in Figure 3.3. The design of PeriFuzz is fully modular, so each component can be swapped out for an alternative implementation

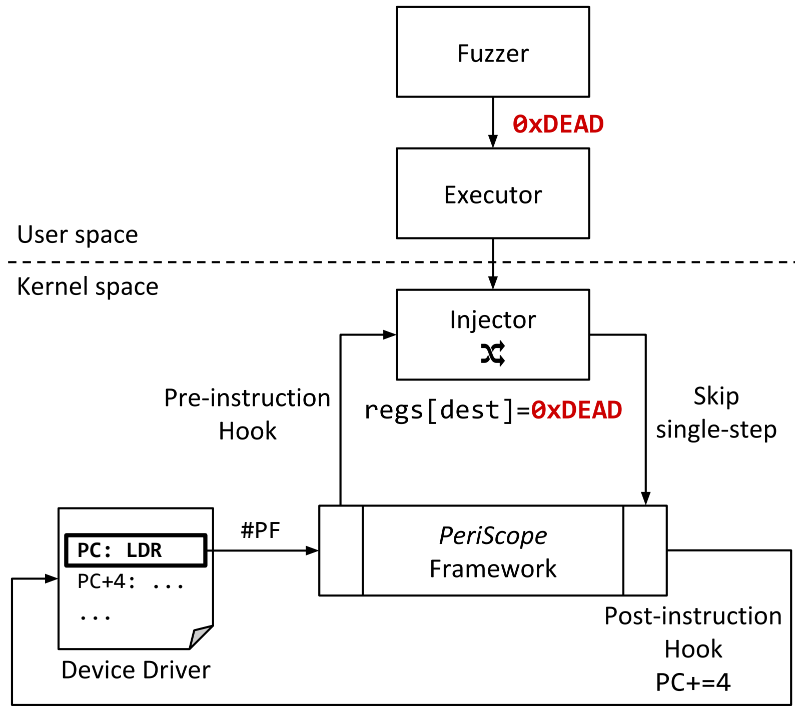


Figure 3.3: PeriFuzz overview.

that exposes the same interface.

Fuzzer. This component runs in user space of the main processor and is responsible for generating inputs for the device driver and processing execution feedback. Thanks to the modular design of PeriFuzz, any fuzzer capable of fuzzing user-space programs can be used. The prototype implementation of PeriFuzz, which is described in detail in Section 3.5, currently uses AFL [187], as several previous works that focus on fuzzing kernel subsystems did [146, 73, 124].

Executor. The executor is a user-space-resident bridge between the fuzzer (or any input provider) and the injector component described below. The executor takes an input file as an argument, and sends the file content to the injector via a shared memory region mapped into both the executor's and the injector's address spaces. The executor then notifies the injector that the input is ready for injection, and periodically checks if the provided input has been consumed. PeriFuzz launches an instance of the executor for every input the fuzzer generates.

The executor is also used to reproduce a crash by providing the last input observed before the crash.

Injector. The injector is a kernel-space module that interfaces with the PeriScope framework. The injector registers a pre-instruction hook with PeriScope, which allows the injector to monitor and manipulate all data the device driver receives from the device. At every page fault, the injector first checks if fuzzing is currently enabled, and if there is a fuzzer/executor-provided input that has not been consumed yet. If both conditions are met, the injector overwrites the destination register with the input generated by the fuzzer.

Note that PeriFuzz manipulates only the values device drivers *read* from MMIO and DMA mappings, but not the values they write. PeriFuzz, in other words, models compromised devices, but not compromised drivers.

3.4.3 Fuzzer Input Consumption

Each fuzzer-generated input is treated as a serialized sequence of memory accesses. In other words, the injector always consumes and injects the first non-consumed inputs found in the input buffer shared between the executor and injector. This fuzzer input consumption model allows for *overlapping fetch fuzzing* as it automatically provides different values for multiple accesses to the same offsets within a target mapping (i.e., overlapping fetches [178]). Providing different values for overlapping fetches enables finding double-fetch bugs [150, 84], if triggering such bugs leads to visible side-effects such as a driver crash. PeriFuzz also keeps track of the values returned for overlapping fetches, and can output this information when a driver crashes, which helps narrow down the cause of the crash. In fact, the double-fetch bugs that were identified using PeriFuzz would not have been found without this information (see Section 3.6).

Since it is assumed in the threat model of PeriFuzz that the attacker cannot access streaming DMA buffers while they are mapped for CPU access (see Section 3.4.1), overlapping fetch

Algorithm 1 Fuzzer input consumption at each driver read.

```
1:  global variables                                    ▷ Initialized when switching fuzzer input
2:      $Input \leftarrow [...]$
3:      $InputOffset \leftarrow 0$
4:      $PrevReads \leftarrow \{\}$
5:      $OverlappingFetches \leftarrow \{\}$
6:  end global variables
7:  function FuzzDriverRead($Address, Width, Type$)
8:      $Value \leftarrow Input[range(InputOffset, Width)]$
9:      for all $Prev$ in $PrevReads$ do
10:         $Overlap \leftarrow Prev.range \cap range(Address, Width)$
11:         if $Overlap$ is not empty then
12:             if $Type$ is DMA Streaming then
13:                 $Value[Overlap] \leftarrow Prev.value(Overlap)$
14:             else
15:                 $OverlappingFetches \leftarrow OverlappingFetches \cup \{(Overlap, Value)\}$
16:             end if
17:         end if
18:     end for
19:     $InputOffset \leftarrow InputOffset + Width$
20:     $PrevReads \leftarrow PrevReads \cup \{(Address, Width, Value)\}$
21:     return $Value$
22: end function
```

fuzzing for streaming DMA buffers must be disabled. To this end, a history of the driver's read accesses to each I/O mapping is maintained, which is consulted, when there is a new access, in order to determine if the access overlaps with any previous access. If they overlap, the same value that was returned for the previous access is returned, and no bytes from the fuzzer input is consumed. Algorithm 1 more precisely describes what value is injected at each driver read from an MMIO or DMA mapping.

An additional benefit of the presented fuzzer input consumption model is that it helps to keep the input size small, because the fuzzer has to generate only the bytes for the driver's read accesses that actually happen and not for the entire I/O mapping being fuzzed, which may contain bytes that are never read by the driver.

3.4.4 Register Value Injection

PeriScope provides the destination register and the access width when it calls into PeriFuzz's pre-instruction hook handler. The fuzzer input is consumed for that exact access width, and then injected into the destination register. The pre-instruction hook function returns the skip-single-step value to PeriScope, as the faulting load instruction was effectively emulated by writing a fuzzed value into its destination register. The post-instruction hook function increments the program counter, so the execution of the driver resumes from the instruction that follows the fuzzed instruction.

3.4.5 Fuzzing Loop

Each iteration of the fuzzing loop, which consumes a single fuzzer-generated input, is aligned to the software interrupt handler, i.e., `do_softirq`. The two hooks inserted before and after the software interrupt handler[2] demarcate a single iteration of the fuzzing loop, in which PeriFuzz consecutively consumes bytes in a single fuzzer input. The design decision of adding hooks to the generic interrupt handler allows PeriFuzz to remain device-agnostic, but device driver developers could provide an alternative device-specific definition of an iteration by inserting those two hooks in their drivers. Several low priority tasks are often deferred to the device driver's own kernel threads, and the fuzzing loop can be aligned to the task processing loop inside those threads.

3.4.6 Interfacing with AFL

The prototype implementation of PeriFuzz uses AFL [187], a well-known coverage-guided fuzzer, as the fuzzing front-end. This is in line with previous work on fuzzing various kernel subsystems [146, 73, 124]. To fully leverage AFL's coverage-guidance, kernel coverage and seed generation support were added in PeriFuzz, as described below.

[2]The hardware interrupt handler is not hooked, since work is barely done in the hardware interrupt context.

Coverage-guidance. A modified version of KCOV was used to provide coverage feedback while executing inputs [172]. Existing implementations of KCOV were developed for fuzzing system calls and only collect coverage information for code paths reachable from system calls. To enable device driver fuzzing along the peripheral interface, KCOV was extended with support for collecting coverage information for code paths reachable from interrupt handlers. Also applied was a patch to force KCOV to collect edge coverage information rather than basic block coverage information [33]. To collect coverage along the execution of the device driver, it is first compiled with coverage instrumentation. This instrumentation informs KCOV of hit basic blocks, which KCOV records in terms of edge coverage. The executor component retrieves the coverage feedback from kernel, once the input has been consumed. Then the executor copies this coverage information to a memory region shared with the parent AFL fuzzer process, after which KCOV is signaled so that it clears the coverage buffer preparing for the next fuzzing iteration.

Automated Seed Generation. Starting with valid test cases rather than fully random inputs improves the fuzzing efficiency, as this lowers the number of input mutations required to discover new paths. To collect an initial seed of valid test cases, the PeriScope framework was used to log all accesses to a user-selected set of buffers. A parser for the access log was implemented, which automatically turns a sequence of accesses into a seed file according to the fuzzing input consumption model (see Section 3.4.3). That said, this step is optional; one could start from any arbitrary seed, or craft test cases on their own.

3.5 Implementation

3.5.1 PeriScope

The prototype implementation of PeriScope was based on Linux kernel 4.4 for AArch64. The prototype is, for the most part, implemented as standalone kernel components that can be ported to other versions of the Linux kernel and even to vendor-modified custom kernels with relative ease.

Tracking Allocations PeriScope hooks the generic kernel APIs used to allocate/deallocate MMIO and DMA regions to maintain a list of allocation contexts. These hooks are inserted into the `dma_alloc_coherent` and `dma_free_coherent` functions to track coherent DMA mappings, into the `dma_unmap_page` function[3] and `dma_map_page` to track streaming DMA mappings, and into `ioremap` and `iounmap` to track MMIO mappings.

PeriScope assigns a context identifier to every MMIO and DMA allocation context. This context identifier is the XOR-sum of all call site addresses that are on the call stack at allocation time. The upper bits of all call site addresses are masked out to ensure that context identifiers remain the same across reboots on devices that enable kernel address space layout randomization (KASLR).

Monitoring Interface PeriScope provides a user-space interface by exposing `debugfs` and `tracefs` file system entries. Through this interface, a user can list all allocation contexts and their active mappings, enable or disable monitoring, and read the circular buffer where PeriScope logs all accesses to the monitored mappings.

As streaming DMA buffer allocations can happen in interrupt contexts, a non-blocking spinlock is used to protect access to data structures such as the list of monitored mappings. In addition, when accessing these data structures from an interruptible code path, interrupts are disabled to prevent interrupt handlers from deadlocking while trying to access the same structures.

[3]`dma_unmap_page` unmaps a streaming DMA mapping from the peripheral processor. Doing so transfers ownership of the mapping to the device driver.

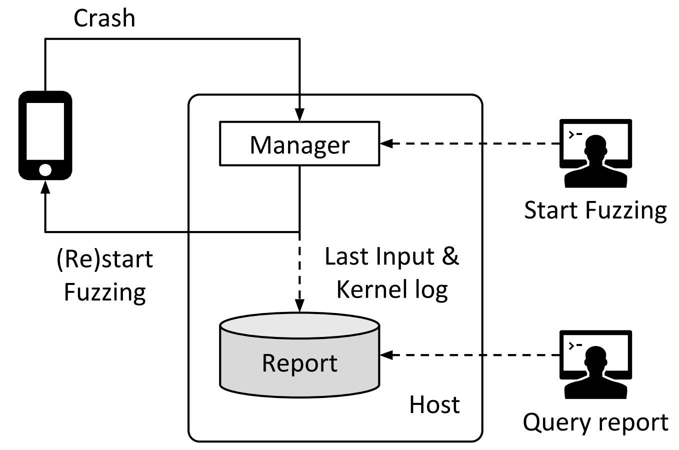


Figure 3.4: Continuous fuzzing with PeriFuzz

3.5.2 PeriFuzz

Kernel-User Interface The injector registers a device node that exposes device-specific `mmap` and `ioctl` system calls to the user-space executor. The executor can therefore create a shared memory mapping via `mmap` to the `debugfs` file exported by the injector module. Through this interface, the executor passes the fuzzer input to the injector running in the kernel space. The `ioctl` handler of the injector module allows the executor (i) to enable and disable fuzzing, and (ii) to poll the consumption status of a fuzzer input it provided. Similarly, KCOV provides the coverage feedback by exporting another `debugfs` file such that the executor can read the feedback by `mmap`ing the exported `debugfs` file.

Persisting Fuzzer Files Many fuzzers including AFL store meta-information about fuzzing and input corpus in the file system. However, these files might not persist if the kernel crashes before the data is committed to the disk. To avoid this, all the fuzzer files are made persistent by modifying AFL such that it calls `fsync` after every file write. Persisting all files that AFL writes allows the user of the framework (i) to investigate crashes using the last crashing input and (ii) to resume fuzzing with the existing corpus stored in the file system.

Fuzzing Manager The fuzzing procedure is completely automated through Python scripts that run on a host separate from the target device. The continuous fuzzing loop is driven by a Python program, as illustrated in Figure 3.4. The manager process runs in a loop in which it (i) polls the status of the fuzzing process, (ii) starts/restarts fuzzing if required, (iii) detects system reboots, (iv) downloads the kernel log and the last input generated before the crash after a reboot, and (v) examines the last kernel log to identify the issue that led to the crash.[4] The manager stores the reports and the last crashing inputs for investigation and bug reporting.

3.6 Effectiveness

PeriScope and PeriFuzz were evaluated by monitoring and fuzzing the communication between two popular Wi-Fi chipsets and their device drivers used in several Android smartphones.

3.6.1 Target Drivers

The choice of Wi-Fi drivers as the evaluation target was primarily motivated by their large attack surface, as evidenced by a recent series of fully remote exploits [7, 18]. Smartphones frequently connect to potentially untrustworthy Wi-Fi access points, and Wi-Fi drivers and peripherals implement vendor-specific, complex internal device-driver interaction protocols (e.g., for offloading tasks to the device side) that rely heavily on DMA-based communication.

The Wi-Fi peripheral chipset market for smartphones is dominated by two major vendors: Broadcom and Qualcomm. As shown in Table 3.1, two popular Android-based smartphones were tested, each of which has a Wi-Fi chipset from one of these vendors. Google Pixel 2 was tested, which runs Android 8.0.0 Oreo[5] and Qualcomm's `qcacld-3.0` Wi-Fi driver. Also tested was Samsung Galaxy S6, which runs LineageOS 14.1 and Broadcom's `bcmdhd4358` Wi-Fi driver. LineageOS 14.1 is a popular custom Android distribution that includes the exact same Broadcom driver as the official Android version for the Galaxy S6.

[4]Syzkaller's report package was used to parse the kernel log.
[5]`android-8.0.0_r0.28`

Table 3.1: Target smartphones.

	Google Pixel 2	Samsung Galaxy S6
Model Name	walleye	SM-G920F
Released	October, 2017	April, 2015
SoC	Snapdragon 835	Exynos 7420
Kernel Version	4.4	3.10
Wi-Fi Device Driver	`qcacld-3.0`	`bcmdhd4358`
Wi-Fi IOMMU Protection	Yes	No

Table 3.2: The number of MMIO and DMA allocation contexts that create attacker-accessible mappings.

Driver	MMIO	DMA Coherent	DMA Streaming
`qcacld-3.0`	1	9	5
`bcmdhd4358`	4	11	29

Note that, although the Samsung Galaxy S6 has an IOMMU, it is not being used to protect the physical memory from rogue Wi-Fi peripherals. Regardless, all experiments were conducted under the assumption that IOMMU protection *is* in place. Newer versions of the Samsung Galaxy phones do enable IOMMU protection for Wi-Fi peripherals.

3.6.2 Target Attack Surface

The code paths of a driver that are reachable from the peripheral device vary depending on the internal state of the driver (e.g., is the driver connected, not connected, scanning for networks, etc). The following evaluation assumes that the driver has reached a steady state where it has established a stable connection with a network. Only the code paths reachable in this state is considered as part of the attack surface. This attack surface can be quantified and characterized by counting (i) the number of allocation contexts that create attacker-accessible MMIO and DMA mappings and (ii) the number of driver code paths that are executed while the user is browsing the web.

Table 3.3: The number of basic blocks executed under web browsing traffic per kernel control path. A basic block could run in interrupt context (referred to as IRQ), kernel thread or worker context (Kernel Thread), or others (Others). Some basic blocks can be reached in multiple contexts.

Driver	IRQ	Kernel Thread	Others	Hit / Instrumented
`qcacld-3.0`	1633 (36.9%)	2902 (65.6%)	672 (15.2%)	4427/81637
`bcmdhd4358`	743 (68.9%)	284 (26.3%)	301 (27.9%)	1078/23404

Table 3.2 summarizes the MMIO and DMA allocation contexts in both device drivers, which create mappings that can be accessed by the attacker while the user is browsing the web. MMIO and DMA coherent mappings were established during the driver initialization, and were still mapped to both the device and the driver by the time the user browses the web; DMA streaming mappings were destroyed after their use, but regularly get recreated and mapped to the device while browsing the web. Thus, an attacker on a compromised Wi-Fi chipset can easily access these mappings, and write malicious values in them to trigger and exploit vulnerabilities in the driver.

Next, the code paths that get exercised under web browsing traffic were classified based on the context in which they are executed: interrupt context, kernel thread context, and other contexts (e.g., system call context). Table 3.3 shows the results. Of all the basic blocks executed under web browsing traffic, 36.9% and 68.9% run in interrupt context for the `qcacld-3.0` and `bcmdhd4358` drivers, respectively. Some of the code that executes in interrupt context may not be reachable from any system calls through legal control-flow paths, and therefore may not be fuzzed by system call fuzzers. This means that the dynamic analysis framework presented in this chapter complements existing efforts along the system call interface.

Table 3.4: Allocation contexts selected for fuzzing. DC stands for DMA coherent, DS for DMA streaming, and MM for memory-mapped I/O.

Driver	Alloc. Context	Alloc. Type	Alloc. Size	Used For
`qcacld-3.0`	**QC1**	DC	8200	DMA buffer mgmt.
	QC2	DC	4	DMA buffer mgmt.
	QC3	DS	2112	FW-Driver message
	QC4	DS	2112	FW-Driver message
`bcmdhd4358`	**BC1**	DC	8192	FW-Driver RX info
	BC2	DC	16384	FW-Driver TX info
	BC3	DC	1536	FW-Driver ctrl. info
	BC4	MM	4194304	Ring ctrl. info

3.6.3 Target Mappings

After investigating how each of the active mappings were used by their respective drivers, the DMA and MMIO regions that were (i) accessed frequently and (ii) used for low-level communication between the driver and the device firmware (e.g., for shared ring buffer management) were targeted in the fuzzing experiments. PeriScope was used to determine which regions the driver accesses frequently, and the driver's code was also manually investigated to determine the purpose of each region.

For `qcacld-3.0`, experiments were conducted for two allocation contexts for DMA coherent buffers and two contexts for DMA streaming buffers. For `bcmdhd4358`, experiments were conducted for three allocation contexts for DMA coherent buffers and one allocation context for an MMIO buffer. Table 3.4 summarizes the allocation contexts for which fuzzing experiments were conducted; all the mappings allocated in those contexts were fuzzed.

3.6.4 Fuzzer Seed Generation

PeriScope's default tracing facilities was used to generate initial seed input files. For each selected allocation context, all allocations of, and all read accesses to the memory mappings were first recorded, while generating web browsing traffic for five minutes. The allocation/access log was parsed to generate unique seed input files. Finally, AFL's corpus minimization tool was used to minimize the input files. This tool replays each input file to collect coverage information and uses that information to exclude redundant files.

3.6.5 Vulnerabilities Discovered

Table 3.5 summarizes the vulnerabilities discovered by using PeriFuzz. Each entry in the table is a unique vulnerability at a distinct source code location.

Table 3.5: Unique device driver vulnerabilities found by PeriFuzz.

Alloc. Context	Alloc. Type	Error Type	Analysis	Double-fetch	Status (Severity)	Impact
QC2	DC	Buffer Overflow	Unexpected RX queue index		CVE-2018-11902 (High)	Likely Exploitable
QC3	DS	Null-pointer Deref.	Unexpected message type		Confirmed (Low)[a]	DoS
QC3	DS	Buffer Overflow	Unexpected peer id		Known	Likely Exploitable
QC3	DS	Buffer Overflow	Unexpected number of flows		Known	Likely Exploitable
QC3	DS	Address Leak/Buffer Ovf.	Unexpected FW-provided pointer		CVE-2018-11947 (Med)[b]	Likely Exploitable
QC3	DS	Buffer Overflow	Unexpected TX descriptor id		Known	Likely Exploitable
QC4	DS	Reachable Assertion	Unexpected endpoint id		Known (Med)	DoS
QC4	DS	Reachable Assertion	Duplicate message		Known (Med)	DoS
QC4	DS	Reachable Assertion	Unexpected payload length		Known (Med)	DoS
BC1	DC	Buffer Overflow	Unexpected interface id	✓	CVE-2018-14852, SVE-2018-11784 (Low)	Likely Exploitable
BC2	DC	Buffer Overflow	Unexpected ring id in create rsp.	✓	CVE-2018-14856, SVE-2018-11785 (Low)	Likely Exploitable
BC2	DC	Buffer Overflow	Unexpected ring id in delete rsp.	✓	CVE-2018-14854, SVE-2018-11785 (Low)	Likely Exploitable
BC2	DC	Buffer Overflow	Unexpected ring id in flush rsp.	✓	CVE-2018-14855, SVE-2018-11785 (Low)	Likely Exploitable
BC2	DC	Null-pointer Deref.	Uninitialized flow ring state		CVE-2018-14853, SVE-2018-11783 (Low)	DoS
BC4	MM	Buffer Overflow	Unexpected flow ring pointer		CVE-2018-14745, SVE-2018-12029 (Low)	Likely Exploitable

[a]Qualcomm confirmed the vulnerability but they do not assign CVEs for low-severity ones.
[b]CVE assigned for the address leak.

Disclosure. These vulnerabilities were responsibly disclosed to the respective vendors. During this process, it was communicated by some vendors that some of the bugs had recently been reported by other external researchers or internal auditors. These bugs are marked as "Known" in Table 3.5. All the remaining bugs were previously unknown, and have been confirmed by the respective vendors. Table 3.5 includes CVE numbers assigned to the bugs that were reported. Also included are the vendor-specific, internal severity ratings for these bugs if communicated by the respective vendors during the disclosure process.

Error Type and Impact. Vulnerabilities found by PeriFuzz fall into four categories: buffer overflows, address leaks, reachable assertions, and null-pointer dereferences. Buffer overflows and address leaks are marked as potentially exploitable, and reachable assertions and null-pointer dereferences are marked as vulnerabilities that can cause a denial-of-service (DoS) attack by triggering kernel panics and device reboots.

Double-fetch Bugs. The fuzzing algorithm (see Algorithm 1) does not attempt to find double-fetch bugs in streaming DMA buffers, since PeriFuzz operates under the assumption that an IOMMU preventing such bugs is in place (see Section 3.4.1). That said, several double-fetch bugs were found in code that accesses coherent DMA buffers. These bugs can potentially be exploited, even when the system deploys an IOMMU. These bugs are described in detail in Section 3.6.7.

3.6.6 Case Study I: Design Bug in `qcacld-3.0`

One of the vulnerabilities found in `qcacld-3.0` is in code that dereferences a firmware-provided pointer. PeriFuzz fuzzed the pointer value as it was read by the device driver. The driver then dereferenced the fuzzed pointer and crashed the kernel. A closer inspection of this vulnerability revealed that it is in fact a design issue. The pointer was originally provided by the driver to the device. Line 11 in Listing 3.8 turns a kernel virtual address, which points to a kernel memory region allocated at Line 4, into a 64-bit integer called `cookie`. The driver sends this `cookie` value to the device, thereby effectively leaking a kernel address.

Listing 3.8 Kernel address disclosure in `qcacld-3.0`.

```
 1  A_STATUS ol_txrx_fw_stats_get(...)
 2  {
 3    ...
 4    non_volatile_req = qdf_mem_malloc(sizeof(*non_volatile_req));
 5    if (!non_volatile_req)
 6      return A_NO_MEMORY;
 7
 8    ...
 9
10    /* use the non-volatile request object's address as the cookie */
11    cookie = ol_txrx_stats_ptr_to_u64(non_volatile_req);
12
13    ...
14  }
```

An attacker that controls the peripheral processor can infer the kernel memory layout based on the cookie values passed by the driver. This address leak can facilitate exploitation of memory corruption vulnerabilities even if the kernel uses randomization-based mitigations such as KASLR. This bug can be fixed by passing a randomly generated cookie value rather than a pointer to the device.

3.6.7 Case Study II: Double-fetch Bugs in `bcmdhd4358`

The `bcmdhd4358` driver contains several double-fetch bugs that allow an adversarial Wi-Fi chip to bypass an integrity check in the driver. Listing 3.9 shows how the driver accesses a coherent DMA buffer that holds meta-information about network data. At Line 4 and Line 5, the driver verifies the integrity of the data in the buffer by calculating and checking an XOR checksum. The driver then repeatedly accesses this coherent DMA buffer again. The problem here is that the device, if compromised, could modify the data between the point of the initial integrity check, and the subsequent accesses by the driver.

Listing 3.9 Initial fetch and integrity check in `bcmdhd4358`.

```
static uint8 BCMFASTPATH dhd_prot_d2h_sync_xorcsum(dhd_pub_t *dhd,
 ↪  msgbuf_ring_t *ring, volatile cmn_msg_hdr_t *msg, int msglen)
{
  ...
  prot_checksum = bcm_compute_xor32((volatile uint32 *)msg,
   ↪  num_words);
  if (prot_checksum == 0U) { /* checksum is OK */
    if (msg->epoch == ring_seqnum) {
      ring->seqnum++; /* next expected sequence number */
      goto dma_completed;
    }
  }
  ...
}
```

PeriFuzz was able to trigger multiple vulnerabilities by modifying the data read from this buffer *after* the integrity check was completed. Listing 3.10 shows one buffer overflow vulnerability, which was triggered by fuzzing the `ifidx` value used at Line 4. The overlapping fetch that occurred before this buffer overflow is a double-fetch bug, because the overlapping fetch can invalidate a previously passed buffer integrity check. Thus, in addition to safeguarding the array access with a bounds check, the driver should copy the contents of the coherent DMA buffers to a location that cannot be accessed by the peripheral device, before checking the integrity of the data in the buffer. Subsequent uses of device-provided data should also read from the copy of the data, rather than the DMA buffer itself.

3.6.8 Case Study III: New Bug in `qcacld-3.0`

Listing 3.11 shows a null-pointer deference bug discovered in the `qcacld-3.0` driver. The pointer to the `netbufs_ring` array dereferenced at Line 9 is null, unless the driver is configured to explicitly allocate this array. The driver configuration used by the Google Pixel 2 did

Listing 3.10 Buffer overflow in `bcmdhd4358`.

```c
void dhd_rx_frame(dhd_pub_t *dhdp, int ifidx, void *pktbuf, int
 ↪  numpkt, uint8 chan)
{
  ...
  ifp = dhd->iflist[ifidx];
  if (ifp == NULL) {
    DHD_ERROR(("%s: ifp is NULL. drop packet\n",
               __FUNCTION__));
    PKTFREE(dhdp->osh, pktbuf, FALSE);
    continue;
  }
  ...
}
```

not contain the entry necessary to allocate the array. Although the driver never executes the vulnerable code under normal web browsing traffic, the vulnerable line *is* reachable through legal control flow paths.

Listing 3.11 Null-pointer dereference in `qcacld-3.0`.

```c
static inline qdf_nbuf_t htt_rx_netbuf_pop(htt_pdev_handle pdev)
{
  int idx;
  qdf_nbuf_t msdu;

  HTT_ASSERT1(htt_rx_ring_elems(pdev) != 0);

  idx = pdev->rx_ring.sw_rd_idx.msdu_payld;
  msdu = pdev->rx_ring.buf.netbufs_ring[idx];
  ...
}
```

It is difficult to detect this bug statically, as it requires a whole-program analysis of the device driver to determine if the `netbufs_ring` pointer is initialized whenever the vulnerable line can execute. PeriFuzz consistently triggered the bug, however. This vulnerability discov-

Table 3.6: Time consumed by PeriScope's page fault handler (measured in μ seconds).

	Mean	Minimum	Maximum
Tracing Only	117.6	99.8	194.5
Tracing + Fuzzing	227.8	182.7	379.7

ery therefore bolsters the argument that fuzzing can complement manual auditing and static analysis.

3.7 Performance

3.7.1 Page Fault

PeriScope incurs run-time overhead as it triggers a page fault for every instruction that accesses the monitored set of DMA/MMIO regions. This overhead was quantified by measuring the number of clock cycles spent inside PeriScope's page fault handler. The AArch64 counter-timer virtual count register `CNTVCT_EL0` was read when entering the handler and when exiting from the handler, and calculated the difference between the counter values, divided by the counter-timer frequency counter `CNTFRQ_EL0`. To minimize interference, hardware interrupts were disabled while executing PeriFuzz's page fault handler. Dynamic frequency and voltage scaling was also disabled.

The page fault handler was tested under two configurations. In one configuration, PeriScope calls the default pre- and post-instruction hooks that only trace and log memory accesses. In the other configuration, PeriFuzz's instruction hooks were registered to enable DMA/MMIO fuzzing. Table 3.6 shows the mean, minimum, and maximum values over samples of 500 page fault handler invocations for each configuration.

Note that the design of PeriScope and PeriFuzz deliberately trade performance for deterministic, precise monitoring of device-driver interactions, by trapping every single access to a set of monitored mappings. In fact, this design allowed us to temporally distinguish accesses to the same memory locations, which was essential to finding the double-fetch bugs. The drivers still function correctly, albeit more slowly, when executed under PeriScope, making it possible to examine device-driver interactions dynamically and enabling PeriFuzz to fuzz it.

3.7.2 Fuzzing

PeriFuzz builds on PeriScope and has additional components that interact with each other, which incur additional costs. The primary contributors to this additional cost are: (i) waiting for the peripheral to signal the driver, (ii) waiting for a software interrupt to be scheduled by the Linux scheduling subsystem, (iii) interactions with the user-space fuzzer, which involve at least two user-kernel mode switches (i.e., one for delivering fuzzer inputs and the other for polling and retrieving feedback), and (iv) other system activities.

Peak Throughput. The overall fuzzing throughput was measured to quantify the overhead incorporating all interactions between the PeriFuzz components. Only the peak throughput is reported in Table 3.7, since crashes and device driver lockups heavily impact the average fuzzing throughput (see Section 3.8). The inverse of the peak fuzzing throughput is a conservative lower bound for the execution time required to process a single fuzzer-generated input. Although PeriFuzz was not optimized for high throughput, these numbers are still in a range that makes PeriFuzz practical for dynamic analysis.

Overhead Breakdown. To illustrate how the fuzzing throughput can be optimized, a breakdown of the fuzzing overhead is now presented. Each iteration of the fuzzing loop was divided into three phases: (i) waiting for fuzzer-generated input to be made available to the kernel module of PeriFuzz, (ii) waiting for the device to raise an interrupt and for the driver to start processing it, and (iii) fuzzing the data read from monitored I/O mappings upon page faults. Once the driver has finished processing the interrupt, the next iteration begins. The execution time of each phase in each iteration was measured. To evaluate the impact of page faults on

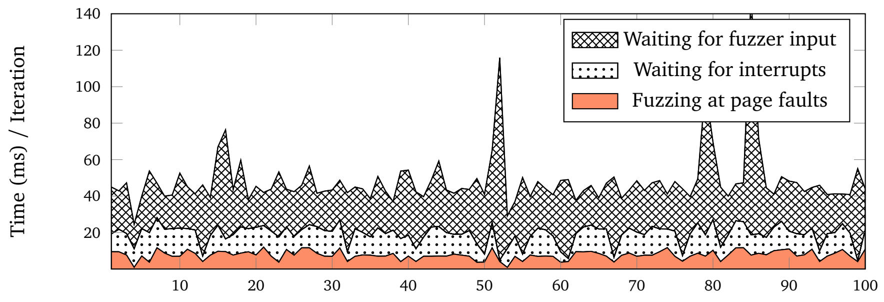
Table 3.7: Peak fuzzing throughput for each fuzzed allocation context.

Driver	Alloc. Context	Peak Throughput (# of test inputs/sec)
`qcacld-3.0`	**QC1**	23.67
	QC2	15.64
	QC3	18.77
	QC4	7.63
`bcmdhd4358`	**BC1**	9.90
	BC2	14.28
	BC3	10.49
	BC4	15.92

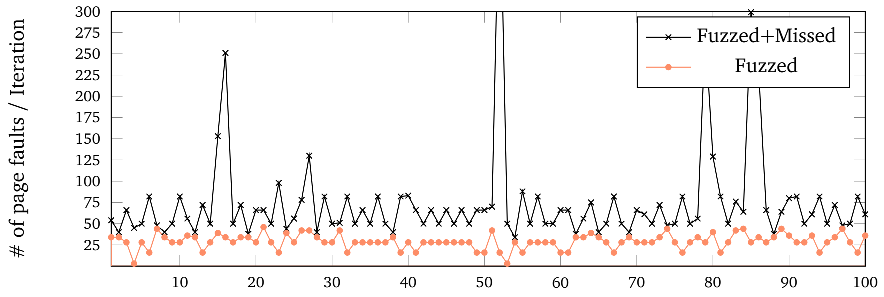
the fuzzing performance, the number of page faults triggered during each iteration was also counted.

The experiment was conducted for **QC1**, which has the highest peak throughput (see Table 3.7). Figure 3.5a shows the measurements of per-phase execution time in a stacked manner, over 100 consecutive iterations of the fuzzing loop. 60% of the total execution time is spent on waiting for the next fuzzer-generated input to become available. This delay is primarily caused by a large number of missed page faults, as hinted by Figure 3.5b. The current implementation of PeriFuzz can miss page faults, when they are triggered while PeriFuzz is preparing for the next input. This delay can be reduced by disabling page faults until the next input is ready. The delay caused by waiting for relevant interrupts, which accounts for 24.2% of the total execution time, can be reduced by forcing relevant interrupts to be raised more frequently.

The actual fuzzing at each page fault still takes 15.8% of the total execution time. One way to reduce this overhead is to trigger page faults only at first access to a monitored mapping within each iteration. At first access, the underlying page can be overwritten with the fuzzer input and then made present, so that subsequent accesses to the page within the same iteration do not trigger extra page faults. This would come, however, at the cost of precision, because it loses precise access tracing capability, effectively disabling overlapping fetch fuzzing as well



(a) The execution time of three phases



(b) The number of fuzzed and missed page faults

Figure 3.5: PeriFuzz overhead breakdown.

as detection of potential double-fetch bugs.

3.8 Limitations

This section describes problems that limit both the effectiveness and efficiency of PeriFuzz. These are well-known problems that also affect other kernel fuzzers, such as system call fuzzers.

System Crashes. The OS typically terminates user-space programs when they crash, and they can be restarted without much delay. Crashing a user-space program therefore has little impact on the throughput of fuzzing user-space programs. Crashes in kernel space, by contrast, cause a system reboot, which significantly lowers the throughput of any kernel fuzzer. This is particularly problematic if the fuzzer repeatedly hits shallow bugs, thereby choking the system without making meaningful progress. This problem was circumvented by disabling certain code paths that contain previously discovered shallow bugs. This does, however, somewhat reduce the effectiveness of PeriFuzz as it cannot traverse the subpaths rooted at these blacklisted bugs.

Note that this problem also affects other kernel fuzzers, e.g., DIFUZE and Syzkaller [44, 66].

Driver Internal States. Due to the significant latency involved in system restarts, whole-system fuzzers typically fuzz the system without restarting it between fuzzing iterations. This can limit the effectiveness of such fuzzers, because the internal states of the target system persist across iterations. Changing internal states can also lead to instability in the coverage-guidance, as the same input can exercise different code paths depending on the system state. This means that coverage-guidance may not be fully effective. Worse, when changes to the persisting states accumulate, the device driver may eventually lock itself up. For example, a problem was observed during the evaluation where, after feeding a certain number of invalid inputs to a driver, the driver decided to disconnect from the network, reaching an error state from which the driver could not recover without a system reboot. Existing device driver checkpointing and recovery mechanisms could be adapted to alleviate the problem [165, 85], because they provide mechanisms to roll drivers back to an earlier state. Such a roll back takes significantly less time than a full system reboot.

3.9 Related Work

Vulnerabilities in device drivers can lead to a compromise of the entire system, since many of these drivers run in kernel space. To detect these vulnerabilities, device driver developers can resort to dynamic analysis tools that monitor the driver's behavior and report potentially harmful actions. Doing this ideally requires insight into the communication between the driver and the device, as this communication can provide the context necessary to find the underlying cause of a vulnerability. Analyzing device-driver communication requires (i) an instance of the device, whether physical or virtual, and (ii) a monitoring mechanism to observe and/or influence device-driver communication. Existing approaches can therefore be classified based on where and how they observe (and possibly influence) device-driver interactions.

Device Adaptation. To exercise direct control over the data sent from the hardware to the driver, an analyst can adapt the firmware of real devices to include such capabilities. This can be done by reverse engineering the firmware and reflashing a modified one [145], or by using custom hardware that supports reprogramming of devices [1]. However, these frameworks are typically tailored to specific devices, and, given the heterogeneity of peripheral devices, their applicability is limited. For example, Nexmon only works for some Broadcom Wi-Fi devices [145], and Facedancer21, a custom Universal Serial Bus (USB) device, can only analyze USB device drivers [1].

Virtual Machine Monitor. A driver can be tested in conjunction with virtual devices running in a virtual environment such as QEMU [15]. The virtual machine monitor observes the behavior of its guest machines and can easily support instrumentation of the hardware-OS interface. Previous work uses existing implementations of virtual devices for testing the corresponding drivers [82, 147]. For many devices, however, an implementation of a virtual device does not exist. In this case, developers must manually implement a virtual version of their devices to interact with the device driver they wish to analyze [86]. Several frameworks alleviate the need for virtual devices by relaying I/O to real devices [186, 168], but these frameworks generally require a non-trivial porting effort for each driver and device, and/or do not support DMA.

Symbolic Execution. S2E augments QEMU with selective symbolic execution [38]. Several tools leverage S2E to analyze the interactions between OS kernel and hardware by selectively converting hardware-provided values into symbolic values [89, 37, 140, 132]. However, symbolic execution in general is prohibitively slow due to the cost associated with the path explosion and constraint solving problem. Moreover, without constraint solving, symbolic execution itself does not reveal vulnerabilities, but rather generates a set of constraints that must be analyzed by separate model checkers. Writing such a model checker is not trivial, especially when the bug requires modeling memory states. Most of the checkers supported by SymDrive, for example, target stateless bugs such as kernel API misuses, but ignore memory corruption bugs [140].

3.10 Conclusions

The interactions between peripherals and drivers can be complex, and hence writing correct device driver software is hard. Unfortunately, as has been recently demonstrated, vulnerabilities in wireless communication peripherals and corresponding drivers can be exploited to achieve remote kernel code execution without invoking a single system call. Nonetheless, no versatile framework has existed until now that analyzes the interactions between peripherals and drivers.

PeriScope, presented in this chapter, is a generic I/O monitoring framework that addresses the specific analysis needs of the two peripheral interface mechanisms MMIO and DMA. The fuzzing framework PeriFuzz builds upon this monitoring framework and can help the end user find bugs in device drivers reachable from a compromised device; uniquely, PeriFuzz can expose double-fetch bugs by fuzzing overlapping fetches, and by warning about overlapping fetches that occurred before a driver crash. Using these tools, 15 unique vulnerabilities in the Wi-Fi drivers of two flagship Android smartphones were found, including 9 previously unknown ones.

Chapter 4

Accelerating Dynamic Analysis of OS Kernels

Despite these recent developments in OS kernel fuzzing, *high-speed OS kernel fuzzing* still remains challenging. One reason is that an OS kernel's execution can easily be prolonged; low-priority, time-consuming tasks (e.g., waiting for I/O requests to complete) in kernel space are are typically processed asynchronously and in a deferred manner. This means that the time it takes for the kernel to process an input in full tends to increase, which in turn decreases the overall efficiency of fuzzing, or dynamic analysis in general.

Another reason relates to the seemingly conflicting goals of high-speed and clean-state fuzzing. Processing each input generated by a fuzzer may change the OS kernel's internal state, which, in turn, can negatively influence the kernel's subsequent test input processing. When it comes to fuzzing kernel-mode device drivers, this influence can result in the driver locking itself up [44, 155], or unstable system state in general, when, for example, a memory corruption bug corrupts a wider system state. Unloading and reloading the driver after processing each test input generated by a fuzzer, and rebooting the OS after hitting a bug, can prevent the interference between test inputs, but doing so results in a significant reduction in fuzzing speed.

As an alternative, prior work used a system snapshot created at system startup to always restore a clean state of the system for each test input, skipping time-consuming reboots. However, snapshot techniques at the virtual machine level without optimizations can be too costly (e.g., QEMU's VM snapshot [2]), and user-mode system snapshot techniques either suffer from similar performance problems [3] or require extensive driver porting efforts when a user-mode kernel is used [181].

This chapter presents a new primitive—*dynamic virtual machine checkpointing*—to address the aforementioned challenges and enable high-speed, clean-state kernel driver fuzzing. The core idea is to continuously create checkpoints during a fuzzing run in order *to skip previously observed, and checkpointed operations* that a kernel driver fuzzer performs. Test cases generated by fuzzers often have a substantial amount of similarities between them, leading to a repeated traversal of identical target driver states. Virtual machine checkpoints, strategically created by the checkpoint management policies, can be used to directly restore the virtual machine state established by time-consuming operations without repeatedly executing them.

This dynamic virtual checkpointing primitive reduces the average test case execution time and, by design, ensures that no residual states remain after executing a test case; even if the test case causes a kernel panic, a known virtual machine state can be quickly restored from an existing checkpoint. The evaluation presented in Section 4.8 shows that this primitive can improve the performance of kernel fuzzers without modifying their underlying input generation algorithm.

4.1 Motivation

4.1.1 Why Use Snapshots?

Prior work used different snapshot techniques for fuzzing OS kernel subsystems [3] and user-space programs [187, 180]. The basic idea is to snapshot the target program before it starts processing input and run the program from that snapshot for each test input. This means that every test input executes on the same, clean state of the target program. No residual state remains, by construction, after each iteration of the fuzzing loop. Test inputs do not interfere with each other, increasing the reproducibility of bugs [181]. Even when a test input corrupts the program state by hitting bugs, a fresh target program state can always be restored from an existing snapshot, which effectively provides crash resilience. Test inputs after a crash can execute without re-executing time-consuming initial bootstrap operations (e.g., system reboot in kernel fuzzing). Fuzzers for user-space programs typically achieve this using `fork()`. A new, fresh child process is forked from a single parent process for each test input, the performance of which is optimized via the copy-on-write mechanism. Several kernel fuzzers also use different forms of snapshots for a reboot-free and reproducible fuzzing [3, 181].

4.1.2 Why Not Use Snapshots?

Although snapshot techniques ensure clean-state fuzzing, the snapshot operations themselves may pose a non-negligible overhead. In particular, system-wide snapshot techniques, e.g., using an emulated, user-mode virtual machine with a `fork`-based snapshot technique [3], or using a hardware-accelerated virtual machine with a full memory snapshot technique, can be expensive. Several fuzzing tools do not use snapshot techniques at all [100, 66, 146], due in part to the overhead. For example, LibFuzzer [100], an in-process user-space fuzzer, and Syzkaller [66], a state-of-the-art kernel fuzzer, execute each test case on the same running instance of the program, and cleaning the program state is left to the user. The user must write cleanup routines to clean up global states that may persist across fuzzing loop iterations. To reduce the overhead associated with virtual machine snapshots, a library OS approach was proposed [181]. This approach, however, lacks compatibility with kernel-mode drivers; it requires manual efforts (or a sophisticated tool [25]) to port device drivers into user-mode ones.

Table 4.1: Comparison between kernel fuzzing approaches.

		Clean State	Compatibility*	High Speed
No Snapshot [66, 155, 44]			✓	✓
User-mode Snapshot	(LibOS) [181]	✓		✓
	(Emulation) [3]	✓	✓	
VM Snapshot		✓	✓	
VM Snapshot with Agamotto		✔	✔	✔

* Compatible with kernel-mode drivers.

4.2 Dynamic Checkpointing

The key idea presented in this chapter is to *dynamically* create checkpoints during a fuzzing run, and use these checkpoints to skip time-consuming parts in the execution of test cases. Recurring sequences of operations that test cases perform need not be executed many times; instead, the state of a virtual machine established by such operations, once checkpointed, can be directly restored from a checkpoint. This idea underpins the design of Agamotto.

Agamotto addresses the shortcomings of prior work, as described in Table 4.1. It uses virtual machine snapshots (or *checkpoints*) and thus inherits all of its advantages—clean-state, reboot-free fuzzing. In contrast to prior snapshot-based approaches, which used a single snapshot created at a fixed point in time (usually at program startup), however, Agamotto creates *multiple* checkpoints automatically at strategic points during a fuzzing run. These checkpoints allow Agamotto to skip initial parts of many test cases, improving the overall fuzzing performance. In addition, we heavily optimized individual virtual machine checkpointing primitives for an efficient multi-path exploration, which limits the performance impact of the primitives themselves.

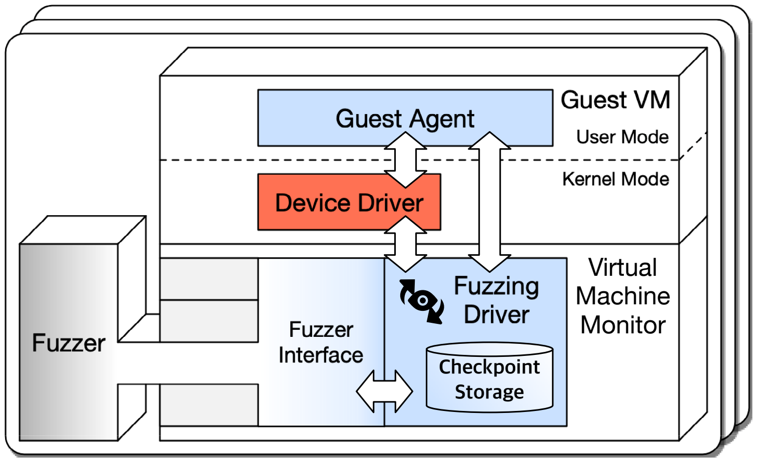


Figure 4.1: Agamotto overview.

4.2.1 Overview

Figure 4.1 shows a high-level overview of Agamotto. The architecture of Agamotto takes the form of a typical virtual machine introspection infrastructure. A full operating system including the kernel-mode **device driver**—the fuzzing target—runs within a **guest virtual machine**. Unlike prior work [3], Agamotto does not impose any constraint on the mode of execution; the guest virtual machine can execute *natively*, using hardware support (e.g., Intel's Virtual Machine Extensions [75]) when available.

The **fuzzer**, whose primary task is to generate test cases and process their execution feedback, is placed *outside* this virtual machine, running alongside the **virtual machine monitor**. Some kernel fuzzers such as Syzkaller place the fuzzer inside the guest virtual machine. This architecture is not suitable when using virtual machine checkpointing, because, as the virtual machine is being restored from a checkpoint, the fuzzer's internal states about the fuzzing progress would also get restored and thus lost. By placing the fuzzer outside the virtual machine, the fuzzer survives virtual machine restorations. Moreover, the fuzzer is shielded against guest kernel crashes and subsequent virtual machine reboots, limiting their impact on the fuzzing progress.

The **fuzzer interface** is a fuzzer abstraction layer that hides details about individual fuzzers from other components. A new fuzzer can be added by implementing various callbacks defined in this interface. These callbacks are invoked by the **fuzzing driver**, the core component of Agamotto placed inside the virtual machine monitor, which (i) drives the fuzzing loop interacting with both the fuzzer as well as the guest virtual machine, and (ii) creates and manages virtual machine checkpoints. The **guest agent**, running inside the guest virtual machine, provides the fuzzing driver with finer-grained virtual machine introspection capabilities. For example, as the guest agent starts at boot, it notifies the fuzzing driver of the boot event, so that it can start the fuzzing loop.

4.2.2 Fuzzing Loop

The fuzzing driver component of Agamotto drives the main fuzzing loop. In each iteration of the fuzzing loop, a fuzzer generates a single test case, executes it, and processes the result of its execution as feedback. In fuzzing event-driven systems such as OS kernels, each test case generated by the fuzzer can be defined as the sequence of actions it performs on the target system. Formally, let $\mathbb{S} = \{S_0, S_1, ..., S_N\}$ be the set of states of the fuzzing target, and $\mathbb{T}$ be a fuzzer-generated test case, which comprises a sequence of N actions, denoted by an ordered set $\{a_1, a_2, ..., a_N\}$. An execution of $\mathbb{T}$, denoted by a function $exec(\mathbb{T})$, is a sequential execution of actions in $\mathbb{T}$ on the fuzzing target. Each action $a_i \in \mathbb{T}$ (for $i \in \{1, ..., N\}$) moves the state of the fuzzing target from S_{i-1} to S_i.[1] The target state observed by the fuzzer (e.g., coverage) is denoted by $\mathbb{R} = \{R_1, R_2, ..., R_N\}$, where each element $R_i \subset S_i$ is the fuzzer-observed state of the fuzzing target after executing a_i. This notation is used throughout this chapter.

Figure 4.2 depicts Agamotto's fuzzing loop in comparison with Syzkaller's fuzzing loop using the above notation. The differences are (i) the added flows into checkpoint and restore and (ii) the removed flows into cleanup and reboot. Virtual machine restoration is initiated after generating, but before executing, a given test case. A checkpoint request is issued and evaluated after each action of a test case. Agamotto skips both cleanup and reboot, since a

[1]This chapter uses a transition-relation style of specifying concurrent, reactive programs (e.g, an OS kernel) to incorporate non-determinism [90, 137]. In other words, a_i is a relation between S_{i-1} and S_i, not a function.

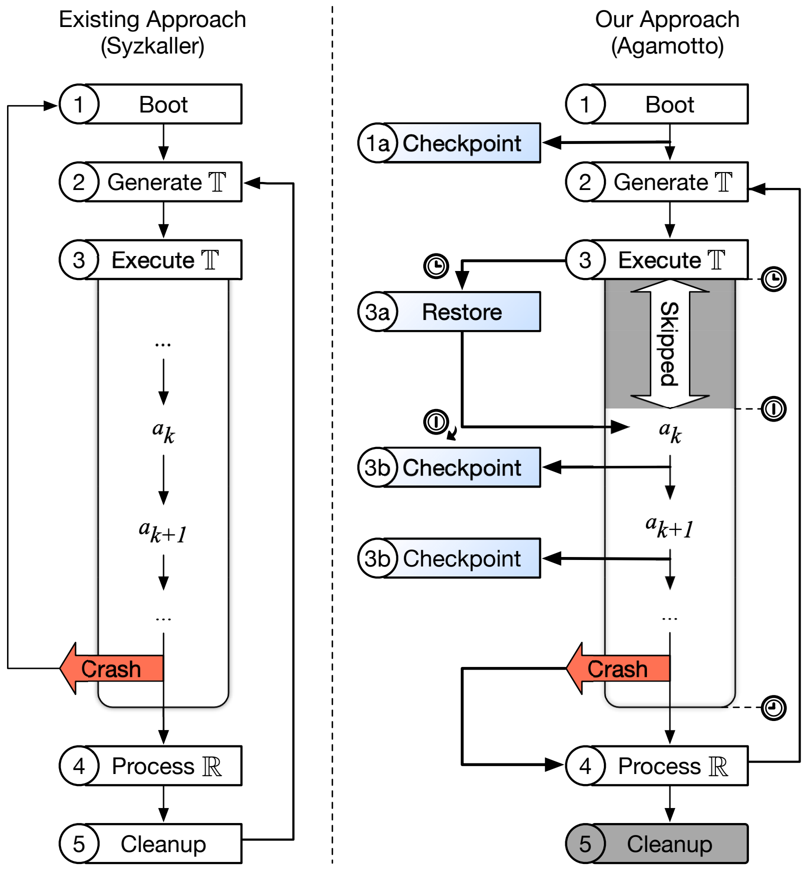
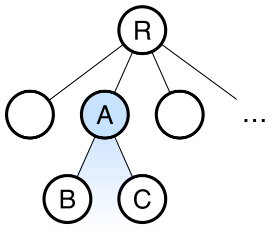


Figure 4.2: Fuzzing loop comparison.

consistent virtual machine state is always restored from a checkpoint without requiring manual cleanup, even after a crash.

After the guest virtual machine boots, but before it starts executing any test case (① in Figure 4.2), the first checkpoint, or the *root checkpoint*, is created ((1a)). Then, the fuzzer generates a test case (②) and starts executing it (③). Based on (i) the test case just generated and (ii) available checkpoints, the fuzzer decides what checkpoint the test case can start executing from and restores the virtual machine from the chosen checkpoint ((3a)). Initial parts of the test case, the result of which is already contained in the checkpoint, are skipped.

During the execution of a test case, secondary checkpoints are requested and created according to a configurable checkpoint policy. After executing each action, the test case executing inside the guest virtual machine sends a checkpoint request to the fuzzer ((3b)). Then Agamotto's checkpoint policy decides whether to checkpoint the virtual machine or not.



Node	Label
R	$\{\}$
A	$\{a_1, a_2\}$
B	$\{a_1, a_2, a_3, a_4\}$
C	$\{a_1, a_2, a_5, a_6, a_7\}$

Figure 4.3: An example checkpoint tree.

Once a test case has been executed, either successfully, with a failure (e.g., timeout), or with a system failure (e.g., kernel crash), the execution result (e.g., coverage) is sent to and subsequently processed by the fuzzer (④). If a test case did not execute in full, but only until kth action, a_k, due to timeouts or system failures, the result for only the executed parts of the test case, $\{R_1, R_2, ..., R_k\}$, will be sent to the fuzzer.

Since restoring the virtual machine entails a full system cleanup, Agamotto skips an explicit cleanup process, if any (❺). To avoid influence between iterations, existing kernel driver fuzzers either perform an explicit cleanup [66] or simply ignore the issue [44, 155]. Agamotto uses virtual machine restoration, which does not allow any internal system state, even corrupted or inconsistent ones created by kernel bugs or panics, to transfer between iterations, without requiring manually-written cleanup routines.

A bug may occur during the cleanup process that is skipped. However, potential bugs that arise in the cleanup process can be found by actively fuzzing the cleanup routines. This way, a cleanup routine can be tested more thoroughly, fully leveraging whatever smart fuzzing capabilities that the fuzzer provides. For example, a fuzzer may generate a corner test case that calls, the cleanup routine multiple times in between other actions, which may trigger more interesting and potentially more dangerous behavior of the driver under test.

4.3 Checkpoint Store and Search

While the fuzzing loop is running, multiple checkpoints are created, which are stored in Agamotto's checkpoint storage. To reduce the overhead induced by processing QEMU's snapshot format we manually manage the (re)storing of guest and device memory pages and use memory-backed volatile storage to capture the remaining virtual machine state.

The volatile state of a virtual machine comprises its CPU and memory state, and any book-keeping information about the virtual machine such as device states kept by the virtual machine monitor. A virtual machine checkpoint must contain all the volatile information to be able to fully restore the state of a virtual machine at a later point in time.

The state of a virtual machine upon each checkpoint request can be attributed to the executed part of the test case. Therefore, each newly created checkpoint is labeled as the *prefix* of a test case that represents only the executed part of a test case. That is, given a test case, $\mathbb{T} = \{a_1, a_2, ..., a_N\}$, the checkpoint created after executing kth action is labeled as $\mathbb{T}_{1..k} = \{a_1, a_2, ..., a_k\}$.

Since the root checkpoint is requested when no part of any test case has executed, it is labeled as an empty test case. Checkpoints subsequently created are marked as a non-empty test case. Checkpoints are stored in a prefix tree, called a *checkpoint tree*. Each node in this tree represents a checkpoint and is labeled as a prefix of the test case that was executing when this checkpoint was created. An example checkpoint tree is depicted in Figure 4.3.

The checkpoint tree forms an efficient search tree of checkpoints. After generating a new test case, Agamotto searches for a checkpoint from which to restore the virtual machine. To find the checkpoint that saves the largest amount of time in executing the test case, Agamotto traverses the checkpoint tree searching for a node that has a label that matches the *longest prefix* of the given test case. In Figure 4.3, given a test case, $\mathbb{T}' = \{a_1, a_2, a_7, a_8\}$, for example, Agamotto finds the node Ⓐ, which has the label that matches the longest prefix, $\{a_1, a_2\}$. Since the checkpoint tree is a prefix tree, this longest prefix match can be performed efficiently without scanning all the checkpoints stored in the tree.

The checkpoint tree also constitutes an incremental checkpoint dependency graph when checkpoint storage is further optimized with incremental checkpoints (see Section 4.5.1).

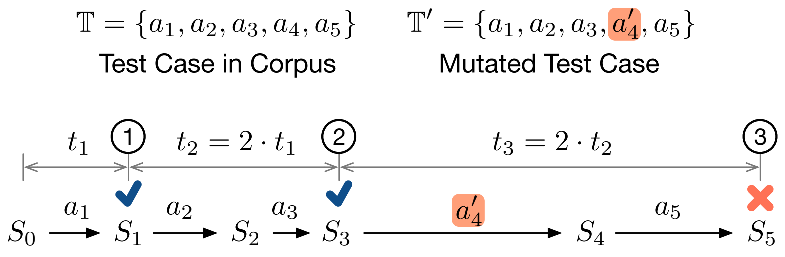

$\mathbb{T} = \{a_1, a_2, a_3, a_4, a_5\}$
Test Case in Corpus
$\mathbb{T}' = \{a_1, a_2, a_3, a_4', a_5\}$
Mutated Test Case
t_1
①
$t_2 = 2 \cdot t_1$
②
$t_3 = 2 \cdot t_2$
③
$S_0 \xrightarrow{a_1} S_1 \xrightarrow{a_2} S_2 \xrightarrow{a_3} S_3 \xrightarrow{a_4'} S_4 \xrightarrow{a_5} S_5$

Figure 4.4: Checkpoint creation policy enforcement example.

4.4 Checkpoint Management Policies

4.4.1 Checkpoint Creation Policy

Checkpointing is requested after executing each action in a test case. A checkpoint creation policy decides, upon each checkpoint request, whether to create a checkpoint or not. A checkpoint creation policy should create checkpoints frequently enough, to increase the chances of finding a checkpoint in restoring the virtual machine later, thus saving time. Checkpointing should not be too frequent, however, because (i) the checkpointing operation itself adds a runtime overhead and (ii) each newly created checkpoint adds memory pressure to the checkpoint storage. Excessive creation of checkpoints, whose expected gain is less than its cost, must be avoided. This section presents two general checkpoint creation policies, which take these two requirements into account.

Checkpointing at Increasing Intervals. This policy creates checkpoints at configurable intervals in the timeline of the guest virtual machine. Upon each checkpoint request, the policy measures the time elapsed since the last checkpoint, and, if it exceeds the configured interval, a checkpoint is created. The intervals can be configured to be constant, or dynamically determined. This policy uses an adaptive interval that increases as the level of the last checkpoint node in the checkpoint tree increases. In particular, it uses an exponentially increasing interval using two as the base; this means that the policy requires a guest execution time twice as long as the one that was required for the last checkpoint (see ① and ② in Figure 4.4). The idea is to reduce the number of checkpoints created later in time during a test case execution, thus alleviating the overhead of checkpoint creation.

Disabling Checkpointing at First Mutation. This policy targets feedback-guided mutational fuzzers, which generate new test cases by mutating parts of older test cases in the corpus. It is well-known that the great majority of mutations do not produce a new feedback signal (e.g., coverage signal [116]), which means that a new test case is more likely to be discarded than to be used for further mutation. Therefore, the expected gain of checkpointing the execution of a test case after the point of a new mutation is low. To reduce the overhead of checkpointing, this policy restricts the creation of checkpoints when executing a mutated test case. Specifically, checkpointing is disabled starting from the location of the first mutation in each test case (see ③ in Figure 4.4). allowed, however, at any point before the new mutation, because the initial part of the test case still corresponds to a prefix of some older test case in the corpus and is likely to occur again as a base for new mutations.

4.4.2 Checkpoint Eviction Policy

Since the size of the checkpoint storage is limited, we cannot store as many checkpoints as created by the checkpoint creation policy. A checkpoint eviction policy evicts an existing checkpoint to free space for a newly created checkpoint when the memory limit allocated for checkpoint storage is reached. Given a configurable checkpoint pool size, checkpoints created by the checkpoint creation policy are unconditionally stored until there is no remaining space. If there is no available space upon creation of a checkpoint, checkpoint eviction policies are consulted to find a node to evict.

The goal of a checkpoint eviction policy is to keep a high usage rate of the checkpoints in restoring a virtual machine. A checkpoint eviction policy needs to predict what checkpoints are likely to be used in the near future, to keep those candidates in the checkpoint tree, and evict others.

We use multiple checkpoint eviction policies, which are consulted sequentially. Each policy takes a set of nodes in the checkpoint tree as input and produces one or more candidate nodes as output. If a policy produces more than one candidate node, the next policy is consulted using the output nodes of the previous policy as its input. We continue consulting each policy in the pipeline until it finds a single checkpoint node to evict.

Policy-1: Non-Active. This policy is placed first in the pipeline, which prevents any active checkpoint nodes from being evicted. Active checkpoint nodes in the checkpoint tree include the node that the virtual machine is currently based on, and, recursively, the parent node of an active node. This policy selects all but the active nodes in the checkpoint tree as eviction candidates, preventing any active node from being evicted. The checkpoints that are currently active can be considered to be *spatially close* because they were created in executing a single test case—the unit of fuzzing. This policy promotes preserving the spatial locality between the active checkpoint nodes by evicting others.

Policy-2: Last-Level. This policy selects the nodes in the last level of the checkpoint tree as eviction candidates. As the depth of the checkpoint tree increases, its nodes are labeled with longer, more specialized test cases. The intuition behind selecting last-level nodes as eviction candidates is that the shorter the test case that a checkpoint node is labeled with, the more likely the label matches test cases that the fuzzer would generate in the future. By evicting last-level nodes, this policy effectively balances the checkpoint tree, letting the tree grow horizontally, rather than vertically.

Policy-3: Least-Recently-Used. The last policy in the pipeline is the Least-Recently-Used (LRU) policy, a policy widely known to be effective at managing different types of caches such as CPU data and address translation caches. The policy tracks the time each checkpoint was last used; a checkpoint is said to be used, (i) when it was created, or (ii) when the virtual machine was restored from it. The policy evicts the checkpoint used earliest in time. As widely known, an LRU policy promotes the temporal locality present in the checkpoint usage pattern. The more recently a checkpoint was used, the more likely it will be used again. Unlike previous policies, this LRU policy always determines one and only one eviction candidate, because each checkpoint is used at a unique point in time.

4.5 Lightweight Checkpoint and Restore

4.5.1 Incremental Checkpointing

QEMU's default virtual machine snapshot mechanism stores all volatile states of a virtual machine in a snapshot image. Each snapshot can introduce prohibitive space overhead, however, the memory size of the virtual machine being the dominating factor. Thus, this full snapshot mechanism is not suitable for the fuzzing use case, where a large number of virtual machines are created, and their snapshots can quickly consume all the available memory. Creating a full snapshot can also introduce a prohibitively high run-time overhead for a virtual machine with high memory requirements.

To reduce both space and run-time overheads of checkpointing, Agamotto performs *incremental checkpointing*, where only the modified (or *dirty*) memory pages are stored into each checkpoint image. The first checkpoint created by Agamotto after the first boot—the root checkpoint—would be identical to what a full snapshot mechanism would create, which contains all pages in memory. Whenever Agamotto creates a new checkpoint based on an existing one, however, only the memory pages that have been modified with respect to the base checkpoint are stored into the checkpoint image. This incremental approach greatly reduces the size of a non-root checkpoint, as well as the time it takes to create one.

The dependencies between incremental checkpoints are already expressed in the checkpoint tree data structure; that is, the virtual machine state of a given node in the checkpoint tree can be fully restored by following the path from the root to that node and incrementally applying checkpoints.

Algorithm 2 Delta checkpoint restore.

```
 1: function DeltaRestore($Src$, $Dst$)
 2:     ▷ Collect pages that need to be restored
 3:     $L \leftarrow$ LowestCommonAncestor($Src, Dst$)
 4:     $Dirty_{Src..L} \leftarrow Dirty_{Src}$
 5:     $Temp \leftarrow$ Parent($Src$)
 6:     while $Temp$ is not $L$ do
 7:         $Dirty_{Src..L} \leftarrow Dirty_{Src..L} \vee Dirty_{Temp}$
 8:         $Temp \leftarrow$ Parent($Temp$)
 9:     end while
10:     $Dirty_{Dst..L} \leftarrow Dirty_{Dst}$
11:     $Temp \leftarrow$ Parent($Dst$)
12:     while $Temp$ is not $L$ do
13:         $Dirty_{Dst..L} \leftarrow Dirty_{Dst..L} \vee Dirty_{Temp}$
14:         $Temp \leftarrow$ Parent($Temp$)
15:     end while
16:
17:     ▷ Restore pages starting from the target node
18:     $Dirty_{Delta} \leftarrow Dirty_{Src..L} \vee Dirty_{Dst..L}$
19:     $Temp \leftarrow Dst$
20:     while $Dirty_{Delta}$ is not empty do
21:         RestorePages($Dirty_{Delta} \wedge Dirty_{Temp}$)
22:         $Dirty_{Delta} \leftarrow Dirty_{Delta} \wedge \neg Dirty_{Temp}$
23:         $Temp \leftarrow$ Parent($Temp$)
24:     end while
25: end function
```

4.5.2 Delta Restore

A strawman approach to restoring a virtual machine using incremental checkpoints is to sequentially apply incremental checkpoint images starting from the root to the target node in an incremental checkpoint tree. The number of memory pages that this strawman approach should restore, however, is greater than the one that a non-incremental snapshot approach would restore; the root checkpoint in an incremental checkpoint tree already contains the full virtual machine state, and additional restorations of incremental checkpoints will add further overhead.

In the fuzzing context, high-performance restore is a requirement, because the virtual machine is restored at the beginning of every iteration of the fuzzing loop. However, since Syzkaller's default Linux kernel configuration for USB fuzzing requires at least 512MB of work-

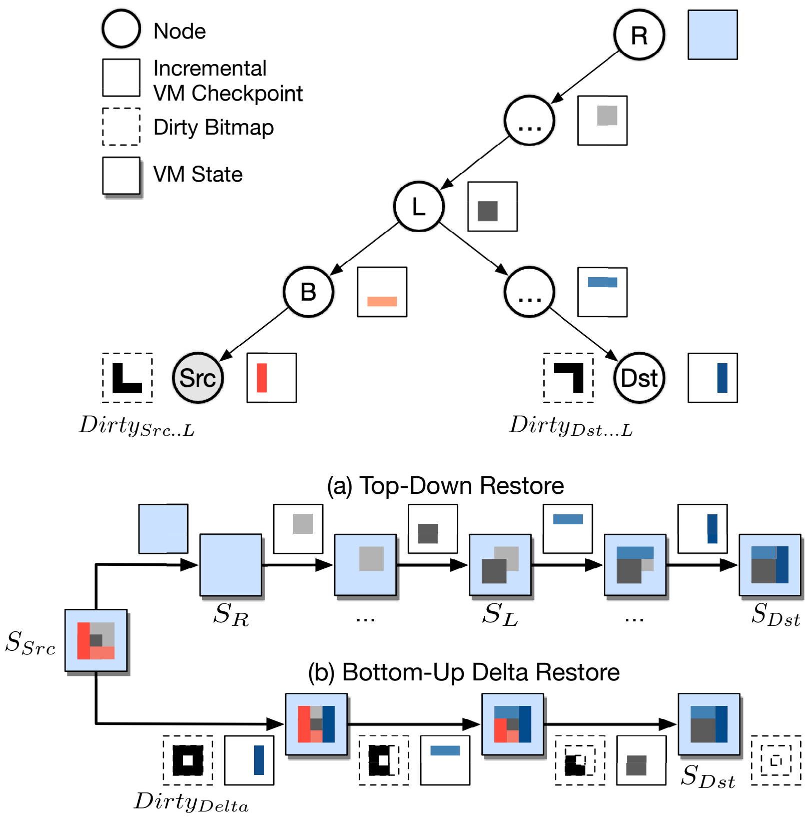


Figure 4.5: Top-down restore vs. Bottom-up delta restore.

ing memory, and Windows requires a minimum of 4GB, it would take up to several seconds for the strawman approach to restore the full virtual machine memory. We, therefore, introduce the *delta restore* algorithm, which minimizes the number of memory pages that are copied during a virtual machine restoration. The full algorithm is described in Algorithm 2. The key idea is to restore (i) only the pages that have been modified in either the current or target virtual machine state after their execution has diverged, and (ii) each modified page only once via bottom-up tree traversal. This means that the number of memory pages that are copied during a virtual machine restoration is bounded by *the number of pages modified* within the current or the target virtual machine state. Observe that, in the strawman approach, the number of copied memory pages is greater than or equal to *the number of all pages* in memory.

Figure 4.5 contrasts (a) the top-down, strawman approach with (b) our bottom-up, delta restore approach in restoring a virtual machine state. In the given checkpoint tree, the node Dst refers to the checkpoint that the system is being restored to, and the node Src is a temporary node representing the current system state from which the restoration starts. The node Ⓑ refers to the last checkpoint that the current system state is based on, and the node Ⓡ refers to the root checkpoint.

The delta restore algorithm first locates the lowest common ancestor node (node Ⓛ) of the node Src and Dst, and computes a bitmap of modified memory pages (or a dirty bitmap) of each node with respect to the node Ⓛ, denoted by $Dirty_{Src..L}$ and $Dirty_{Dst..L}$, respectively. We take the union of these two dirty bitmaps, which we call a *delta* dirty bitmap, denoted by $Dirty_{Delta}$. $Dirty_{Delta}$ contains a complete list of memory pages that need restoring. Then, starting from the node Dst, we traverse the checkpoint tree *backwards* to the root node. At each node during the traversal, we restore only the memory pages that are in $Dirty_{Delta}$ and clear their corresponding bits in $Dirty_{Delta}$ to ensure that each dirty page is restored only once. The traversal stops when $Dirty_{Delta}$ is fully cleared. The strawman approach, by contrast, restores all pages stored in incremental checkpoints starting from the node Ⓡ.

4.6 I/O Interception

Fuzzing driver code paths that can be reached through a given peripheral interface requires interception and redirection of the driver's I/O requests. We find two common models for driver I/O interception and redirection:

- **User-Space Device Emulation.** I/O requests coming from a kernel driver are redirected to a user-mode program through the system call interface. This approach typically requires kernel source code modifications for intercepting and redirecting driver I/O requests.

- **Device Virtualization.** Device virtualization techniques allow the virtual machine monitor to intercept I/O requests coming from the corresponding kernel driver.

Syzkaller's USB fuzzing mode takes the user-space device emulation approach. It adds a kernel module that intercepts and redirects USB driver I/O requests to a program running in user space via the system call interface. Since Syzkaller already contains many smart fuzzing features such as structure-awareness of USB packets, we modified Syzkaller such that Agamotto can be applied. Our key modification was moving Syzkaller's fuzzer outside of the virtual machine so that the fuzzer survives virtual machine restorations as well as kernel crashes. We also modified the communication channels between Syzkaller's components. The fuzzing algorithm and other aspects of Syzkaller were left unmodified.

For fuzzing the PCI interface, we developed our own fuzzer, which uses a device virtualization approach to intercept the driver's I/O requests at the virtual machine monitor level. A key benefit of this approach is that it does not require kernel modifications; a virtual device can be implemented within the virtual machine monitor without modifying the guest OS kernel. We created a *fake* virtual PCI device in QEMU, and plugged it into QEMU's virtual PCI bus. Our fake PCI device attached to the PCI bus gets recognized by the PCI bus driver as the guest OS kernel boots, and, once the target PCI driver gets loaded, it intercepts all memory-mapped I/O (MMIO) requests coming from the target driver. We fuzzed these MMIO requests by sending fuzzer-generated data to the driver as a response to each driver I/O request.

4.7 Implementation

Agamotto was implemented on top of QEMU 4.0.0 running in an x86 Linux environment [15]. The prototype implementation used the Linux Kernel Virtual Machine (KVM) for hardware accelerated virtualization [125]. Syzkaller[2] was used for USB fuzzing [66], and American Fuzzy Lop (AFL) version 2.52b was used for PCI fuzzing [187].

Dirty Page Logging. KVM's dirty page logging functionality was used to identify modified memory pages of the guest virtual machine, as required for the incremental checkpointing and

[2]Specifically, the commit number: ddc3e85997efdad885e208db6a98bca86e5dd52f

delta restoration techniques presented earlier in this chapter. KVM's dirty page bitmap was looked up upon a checkpoint creation request and a virtual machine restoration request. KVM's dirty page bitmap was cleared after each checkpoint creation and virtual machine restoration. Note that KVM's dirty page logging can transparently be accelerated as hardware support—e.g., Page Modification Logging in Intel x86 CPUs—becomes available. Using this dirty page logging, optimized versions of virtual machine checkpointing and restoration mechanisms were newly implemented in QEMU, since existing snapshot implementation in QEMU was found to be slower than expected.

Inter-Component Communication. A variety of commodity virtual machine introspection (VMI) mechanisms were used to implement inter-component communication channels. Control channels were implemented via hypercalls and VIRTIO pipes established between QEMU and the guest virtual machine [126]. Data channels for bulk data transfer were implemented via direct reads and writes to the guest memory or by using a separate shared memory device.

Syzkaller and AFL Support. Agamotto was designed to support multiple fuzzers, and the current prototype supports two different fuzzers. When running Agamotto with Syzkaller for fuzzing the USB interface, Syzkaller's fuzzer was used (`syz-fuzzer`) as Agamotto's fuzzer component and Syzkaller's executor (`syz-executor`) as Agamotto's guest agent. They were both modified such that they use VMI-based communication channels. When running Agamotto with AFL for fuzzing the PCI interface, an AFL fuzzer running as a thread was used as Agamotto's fuzzer component, and a shell script was used as the guest agent, which simply loads the target PCI driver.

4.8 Evaluation

All of the experiments described in this section were conducted on a machine equipped with AMD EPYC 7601 CPU and 500GB of memory. Device drivers in Linux v5.5-rc3 were targeted in

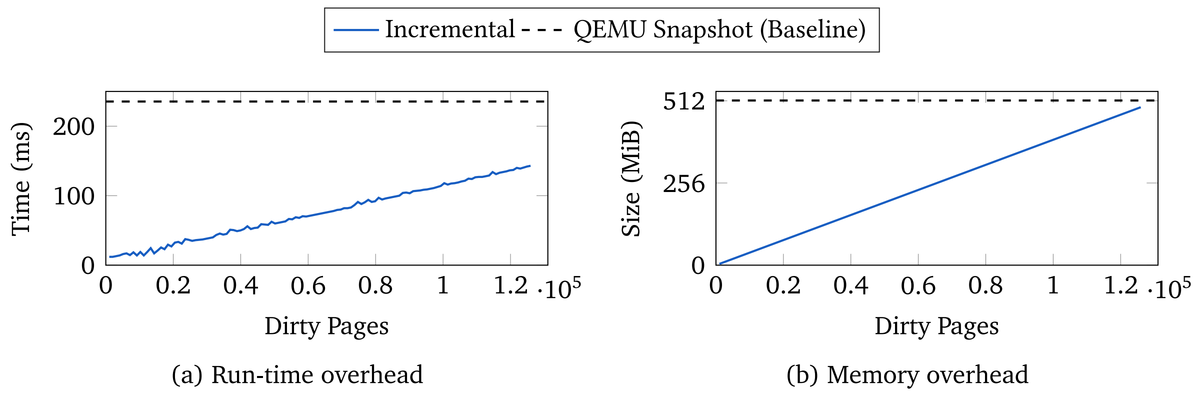


Figure 4.6: Overheads of incremental checkpointing.

evaluating Agamotto. KernelAddressSanitizer was enabled to expose more bugs [64]. This section presents an evaluation of Agamotto's individual primitives first, and then the performance evaluation of kernel driver fuzzers augmented with Agamotto in both USB and PCI fuzzing scenarios.

4.8.1 Incremental Checkpointing

Here the run-time and memory overheads of the prototype incremental checkpointing implementation are compared with the overheads of QEMU's non-incremental snapshot approach [2]. To measure the overheads conservatively, QEMU's zero page optimization—a checkpoint size reduction technique that handles a page filled with zeros by storing a fixed-size entry in the checkpoint image, instead of storing 4KiB of zeros—was disabled.

Run-Time Overhead. The run-time overhead of checkpointing primarily depends on the number of pages copied into the checkpoint image. Figure 4.6a shows the overhead of the incremental checkpointing mechanism, and that of the baseline, when checkpointing a 512MiB memory guest virtual machine. As the number of dirty pages increases, the run-time overhead of incremental checkpointing increases linearly. In contrast, the overhead of the baseline, a non-incremental approach, remains constant regardless of the number of dirty pages. In addition, QEMU's non-incremental checkpoint approach adds an additional overhead due to its

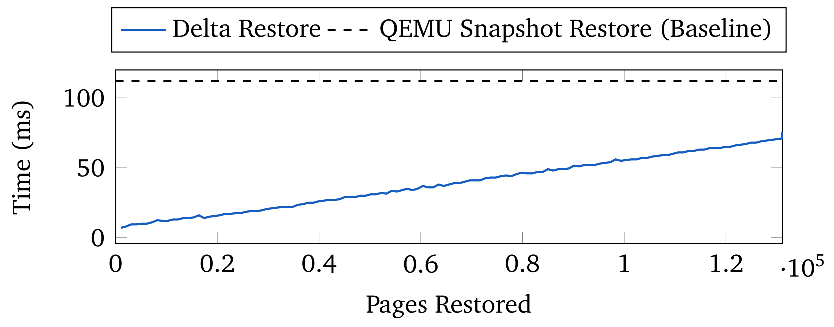


Figure 4.7: Run-time overhead of delta restore.

implementation and the full inclusion of the device memory, of which only a few pages are dirtied during fuzzing. A full restore can, therefore, take more than 200ms per checkpoint for copying all 131,072 pages, whereas incremental checkpointing, for a typical range of the number of dirty pages (see Section 4.8.3), takes less than 20ms on average as it only copies the dirty pages.

Memory Overhead. Figure 4.6b shows how the size of each checkpoint correlates to the number of dirty pages when checkpointing a 512MiB memory virtual machine. As expected, the size of an incremental checkpoint increases in proportion to the number of pages that have been modified since the last checkpoint. Given the distribution of the number of modified pages, which typically ranges from 0 to 8,000 (see Section 4.8.3), each checkpoint should take no more than 64MiB. With the zero page optimization enabled, the size of each checkpoint observed in actual fuzzing runs, on average, is less than 32MiB. This is a reduction of 90% or more in size from the baseline.

4.8.2 Delta Restore

Run-Time Overhead. Figure 4.7 shows the run-time overhead of the prototype implementation of the delta restore algorithm depending on the number of pages that are restored when restoring a 512MiB memory virtual machine. QEMU's default restoration mechanism was used as the baseline, which restores a virtual machine state from a non-incremental, full snapshot

Table 4.2: USB and PCI fuzzing targets.

Target	USB (§4.8.3)	PCI (§4.8.4)	Path (/drivers/...)
RSI	✓		net/wireless/rsi
MWIFIEX	✓		net/wireless/marvell/mwifiex
AR5523	✓		net/wireless/ath/ar5523
BTUSB	✓		bluetooth/btusb.c
PN533	✓		nfc/pn533
GO7007	✓		media/usb/go7007
SI470X	✓		media/radio/si470x
USX2Y	✓		sound/usb/usx2y
ATLANTIC		✓	net/ethernet/aquantia
RTL8139		✓	net/ethernet/realtek
STMMAC		✓	net/ethernet/stmicro
SNIC		✓	scsi/snic

image. The smaller the number of restored pages as computed by the delta restore algorithm, the less time it takes to restore a virtual machine state. The number of restored pages, as observed in actual fuzzing runs, is significantly lower than the total number of pages in memory (see Section 4.8.3). With an average number of under 8,000 restored guest and device memory pages, the prototype delta restore implementation can restore the virtual machine in 12.5ms on average, 8.9 times faster than the baseline, QEMU's implementation of the full snapshot restore approach, which takes 112ms on average.

4.8.3 Syzkaller-USB Fuzzing

Experimental Setup. USB drivers were fuzzed individually, one in each experiment. As shown in Table 4.2, 8 USB drivers were chosen in total, which include drivers (i) of 5 different classes, (ii) of different numbers of source lines of code, and (iii) from different vendors. A total of 32 fuzzing instances were used in fuzzing each driver for three hours. Each instance fuzzed the driver running in a 512MiB memory virtual machine.

All USB related functions in Syzkaller were enabled[3] so they appear in the generated test

[3] `syz_usb_connect`, `syz_usb_control_io`, `syz_usb_ep_read`, `syz_usb_ep_write`, and

inputs, and the parameters of `syz_usb_connect`—i.e., device and interface descriptors—were constrained to fuzz the drivers individually in each experiment. To minimize the effects of non-determinism in the experiments, coverage instrumentation was limited to the driver code as well as generic kernel code that drivers call into.[4]

The fuzzing algorithm of Syzkaller was not modified. Syzkaller's default five-second timeout was increased to ten seconds to encourage deeper exploration.[5] Fuzzing was started without any seed input to eliminate its impact on the results. To minimize the randomness inherent in fuzzing algorithms, different but fixed sets of PRNG seed values were used for different instances, using the equation, $\{id_{inst} + \#crashes_{inst} * 128\}$ where $inst = \{0, 1, ..., 31\}$. This equation ensures that seed values (i) are always unique across instances, and (ii) change after each kernel crash. With these adjustments, the randomness of Syzkaller's fuzzing algorithm was controlled; note, however, that the randomness originating in the target system, e.g., coverage signal, was not controlled. To account for this randomness, each experiment was run three times.

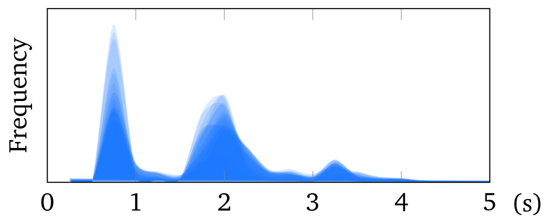
Two different versions of Agamotto were run: (i) a full-fledged Agamotto and (ii) Agamotto with only the root checkpoint enabled (referred to as Agamotto-R). By comparing Agamotto with Agamotto-R, the effectiveness of checkpoints dynamically created by Agamotto can be quantified. Syzkaller was used as a baseline, only with the aforementioned changes made for controlling timeout and randomness. Agamotto was configured with the following additional parameters: The checkpoint pool size was configured to be 12GiB per instance, and 500ms was used as the initial checkpoint creation interval.

Execution Time of Individual Test Cases. Figure 4.8 shows how much time Agamotto skips in executing each test case. By using fine-grained checkpoints created by Agamotto, the initial parts of many test cases were skipped. Each test case's execution time was measured in all

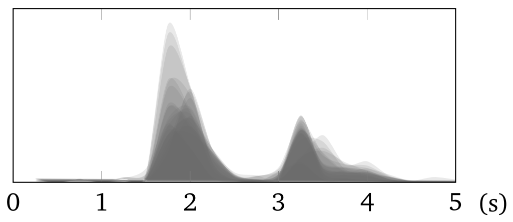
`syz_usb_disconnect` were enabled.

[4]We instrumented the source code under the following directories: drivers, sound/{usb, core}, and net/{bluetooth, nfc, wireless}.

[5]Syzkaller's default timeout model was followed, where each test case can execute for at most three seconds, but, as long as the last action has returned within last one second, it can execute up to ten seconds.

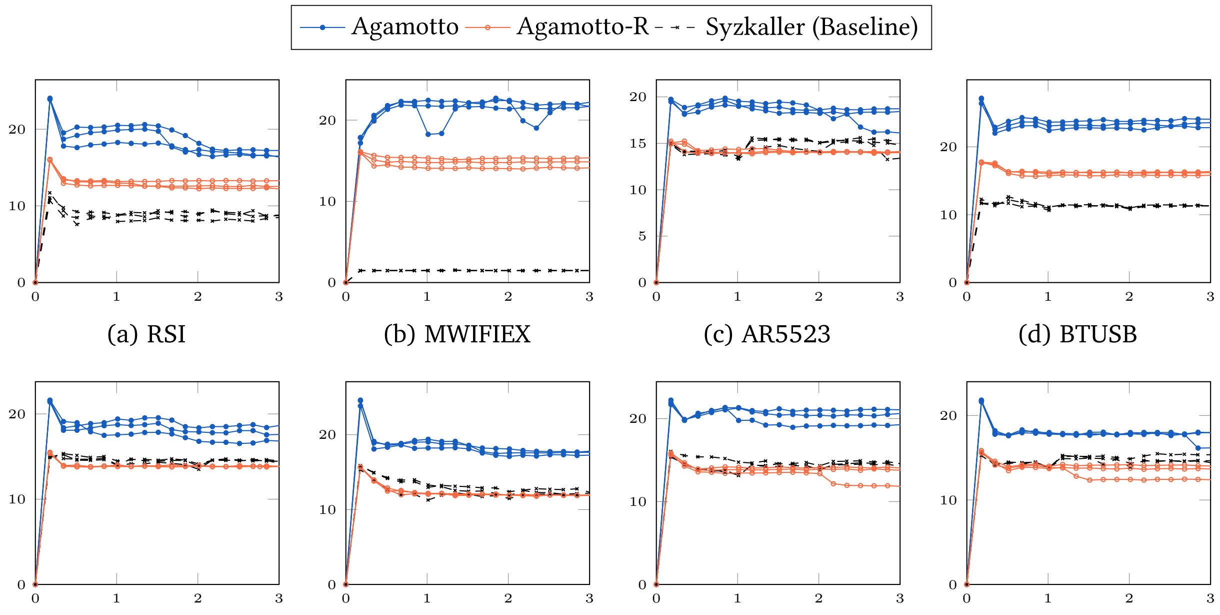


(a) Agamotto execution time



(b) Normal execution time

Figure 4.8: Distribution of the execution time per test case in Syzkaller-USB fuzzing.



(a) RSI (b) MWIFIEX (c) AR5523 (d) BTUSB

(e) PN533 (f) GO7007 (g) SI470X (h) USX2Y

Figure 4.9: Syzkaller-USB fuzzing throughput (execs/second) measured every 10 minutes for 3 hours.

experiments (Figure 4.8a) and computed each test case's normal execution time, the time each test case execution could have taken if fine-grained checkpoints were not used (Figure 4.8b). Agamotto successfully reduced the execution time of many test cases—a large portion of test cases took less than a second with Agamotto, as shown in Figure 4.8a.

Overall Fuzzing Throughput. Figure 4.9 illustrates how much Agamotto improves Syzkaller's USB fuzzing throughput. This overall fuzzing throughput includes the overhead of Agamotto itself. One common trend observed in all experiments is that Agamotto's fuzzing throughput peaks in the first 10 minutes. This is because, as fuzzing instances are started, lots of

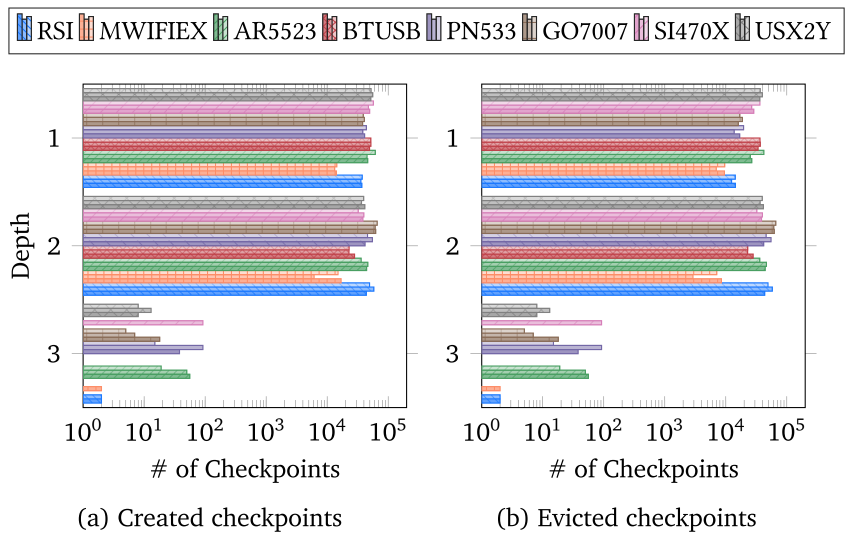
Table 4.3: Checkpoint hit and guest execution time statistics. Median values from 3 independent runs.

	# Checkpoints		# Executions		Guest Exec. Time	
	Created	Evicted	Total	Hit (Rate)	Total	Skipped (Rate*)
RSI	87k	63k	201k	120k (59%)	90.3h	42.1h (31%)
MWIFIEX	19k	9.8k	236k	60k (25%)	28.0h	18.3h (39%)
AR5523	91k	71k	201k	116k (57%)	95.0h	38.6h (28%)
BTUSB	74k	59k	254k	145k (57%)	94.7h	47.1h (33%)
PN533	89k	65k	199k	116k (58%)	95.2h	39.7h (29%)
GO7007	105k	83k	201k	126k (62%)	95.1h	44.5h (31%)
SI470X	88k	67k	223k	130k (58%)	94.9h	43.6h (31%)
USX2Y	92k	76k	195k	90k (46%)	95.0h	29.4h (23%)
Geo. Mean				51.5%		30.9%
ATLANTIC	8.4k	0.6k	191k	43k (22%)	95.2h	18.5h (22%)
RTL8139	17.9k	6.5k	272k	128k (47%)	91.5h	78.9h (46%)
STMMAC	4.8k	0.3k	160k	23k (14%)	95.2h	15.9h (14%)
SNIC	4.0k	0.2k	153k	8.3k (5.4%)	95.3h	5.35h (5.3%)
Geo. Mean				17.0%		16.7%

* Skipped/(Skipped+Total)

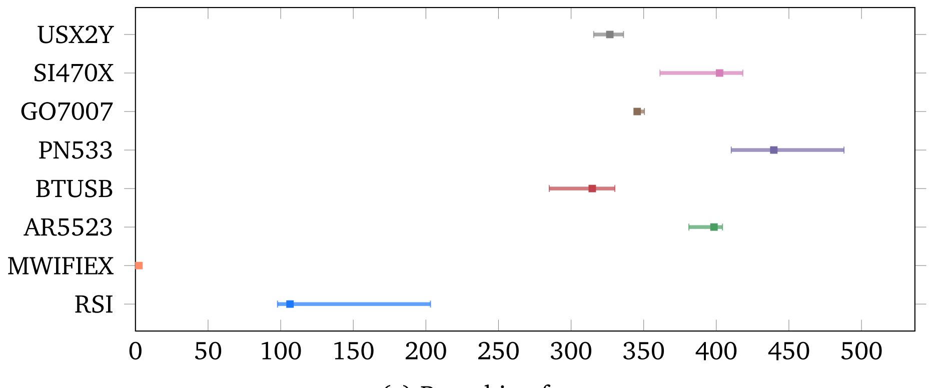
test cases producing new coverage were discovered and minimized. Each minimized test case was then mutated 100 times and executed in a row. During this period of time in which new inputs were frequently discovered, a large number of *similar* test cases were executed in a row, the throughput of which was greatly improved by Agamotto. As the fuzzing continued, coverage-increasing test cases were seldom discovered, stabilizing the throughput. Still, Agamotto's throughput was consistently higher than the baseline. Of the eight analyzed drivers only two experienced kernel crashes (MWIFIEX and RSI). The performance improvement of the remaining targets is therefore solely due to the reduced average execution time by using the checkpoints created by Agamotto.

Checkpoint Utilization and Effectiveness. Selecting a non-root checkpoint in executing a test case is referred to as *a checkpoint hit*, and selecting the root checkpoint as *a checkpoint miss*. The *hit rate* refers to the portion of executions that had a checkpoint hit among all executions.



(a) Created checkpoints

(b) Evicted checkpoints



(c) Branching factor

Figure 4.10: Distribution of the depths of all the (a) created and (b) evicted checkpoints in the checkpoint trees, as well as (c) the resulting branching factors of the trees, measured in Syzkaller-USB fuzzing.

At each checkpoint hit, a different amount of time is skipped depending on the checkpoint used. Table 4.3 summarizes the hit rates, as well as the amounts of the guest execution time skipped in each fuzzing experiment. The hit rates and time skip rates vary depending on the driver targeted in each experiment; on average, a hit rate of 51.5% was achieved, saving 30.9% in guest execution time.

To quantify the effectiveness of multiple checkpoints created by Agamotto, the throughput of Agamotto was compared against Agamotto-R; the throughput was improved by 38% on average. The shape of the checkpoint tree used to achieve this improvement is characterized in Figure 4.10. The depths of the checkpoint nodes—i.e., the number of edges from the root

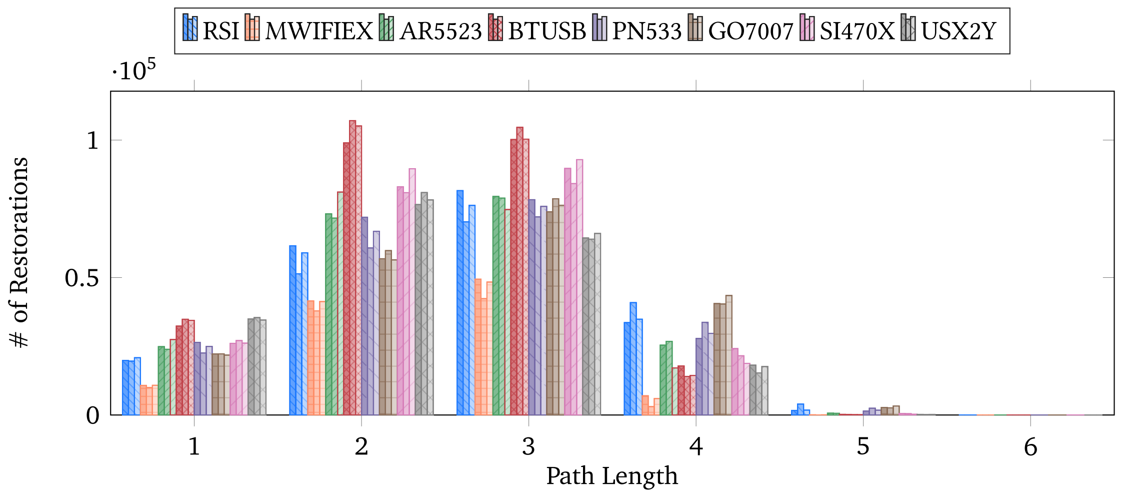


Figure 4.11: Distribution of the length of the restoration path in Syzkaller-USB fuzzing.

node—created and evicted by Agamotto ranged from 1 to 3, and the resulting checkpoint trees had an average branching factor of 175. This large branching factor reflects (i) how Syzkaller explores the input space, and (ii) that the checkpoint management policies favor checkpoint nodes of lower depths in the checkpoint tree (see Section 4.4.2). In these checkpoint trees, the length of the restoration path—i.e., the path from the node representing the dirty system state after each test case execution to the node being restored—ranged from 1 to 6, as shown in Figure 4.11. The widely ranging lengths of the restoration paths mean that different checkpoints created at various depths were actively used for virtual machine restoration, which also supports the utility of multiple checkpoints created by Agamotto.

Resilience to Kernel Panics. Agamotto found several known bugs in RSI and MWIFIEX that were already found and reported in earlier kernel versions by Syzbot [173], but left unfixed. Agamotto found one unknown bug in MWIFIEX. This bug was not found in the baseline (nor Syzbot), as it was obscured by a known, shallow bug in MWIFIEX, which repeatedly caused immediate kernel panics in the baseline. In contrast, since Agamotto puts the fuzzer outside the virtual machine, Agamotto continuously generated and ran test cases despite kernel panics, eventually getting past the known bug to discover this unknown bug. Moreover, Agamotto maintains the fuzzing throughput, even when it frequently hits these bugs. In fuzzing MWIFIEX

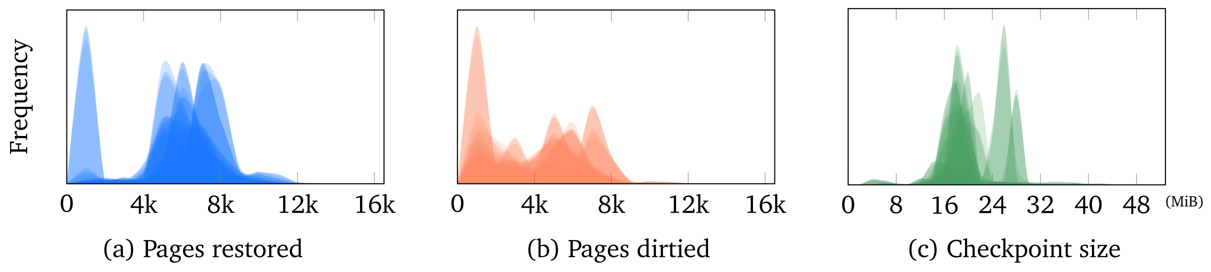


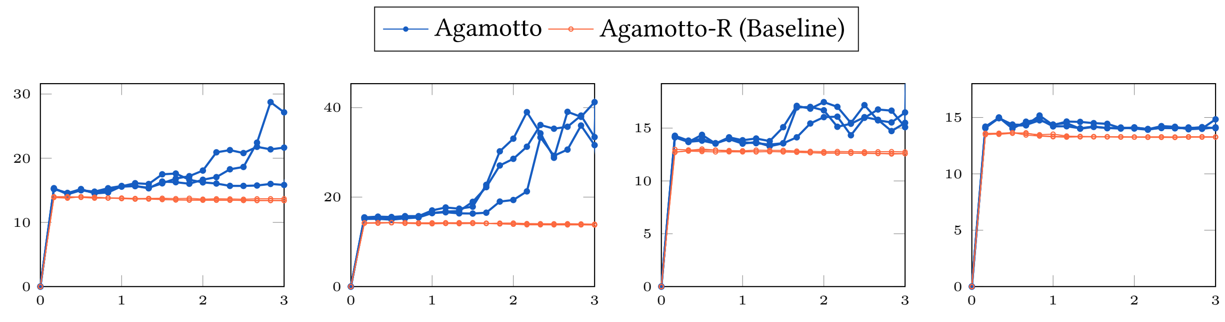
Figure 4.12: Distribution of the number of pages (a) restored and (b) dirtied per iteration, and (c) the size of checkpoints in Syzkaller-USB fuzzing.

as well as RSI, where Agamotto encountered bugs more than 6,000 and 200 times in every 10 minutes, their baseline throughput is significantly lower than the ones observed in fuzzing other drivers. Agamotto, in contrast, maintained a similar level of throughput across all experiments.

Dirty Page Statistics. To show that the incremental checkpointing and delta restore techniques are effective in practice, the number of pages that are restored and dirtied in each iteration of the fuzzing loop were counted in each experiment. The results are shown in Figure 4.12. Figure 4.12b shows that the number of pages dirtied after executing a test case has an upperbound near 8,000 pages. The number of restored pages is similarly bounded as shown in Figure 4.12a, but often exceeds this limit when the memory pages contained in the checkpoint being restored do not completely overlap with the set of dirty pages of the virtual machine at the time of restoration. This means that, as discussed in Section 4.8.1 and 4.8.2, the run-time overhead of virtual machine checkpointing and restoration was greatly reduced. Also, with the zero page optimization enabled, most of the checkpoints were found to be smaller than 32MiB, as depicted in Figure 4.12c.

4.8.4 AFL-PCI Fuzzing

Experimental Setup. To evaluate a device-virtualization-based PCI fuzzer augmented with Agamotto, four PCI drivers in Linux shown in Table 4.2 were fuzzed. AFL was used as the fuzzing engine this time, with its fuzzing algorithm unmodified again; note that AFL imple-



(a) ATLANTIC (b) RTL8139 (c) STMMAC (d) SNIC

Figure 4.13: AFL-PCI fuzzing throughput (execs/second) measured every 10 minutes for 3 hours.

ments a different input generation and scheduling algorithm than Syzkaller. The evaluation was performed against a more conservative baseline, where Agamotto was applied, but the creation of non-root checkpoints was disabled. In other words, these PCI experiments were designed to demonstrate the effectiveness of fine-grained checkpoints created by Agamotto in improving the performance of kernel driver fuzzing.

To avoid inadvertently biasing the results through the seed input, fuzzing was started with a single input as the seed, which contains an eight-byte string—"Agamotto" in the ASCII format—and without any dictionary entries. Randomness in the fuzzing algorithm was controlled the same way as in the USB experiments. Each driver was fuzzed using 32 instances for three hours. Since the driver's interactions with a PCI device were faster than what was observed in USB fuzzing, the starting checkpoint interval was reduced to 50ms. The timeout value was set to 100ms; each iteration of the fuzzing loop was terminated 100ms after the driver's last access to the I/O mappings.

Fuzzing Throughput. Although AFL uses a fuzzing algorithm different from Syzkaller's, Agamotto again improved the throughput by 21.6% on average, as shown in Figure 4.13. Note that neither AFL's nor Syzkaller's fuzzing algorithm produces a sequence of test cases that are optimal for Agamotto to accelerate. In particular, AFL's fuzzing algorithm is not tailored to fuzzing event-driven systems (e.g., it always mutates each test case in the corpus from the first byte). Still, Agamotto consistently improved the fuzzing throughput in all experiments, and

Table 4.4: Number of executions and discovered paths in AFL-PCI fuzzing. Median values from 3 independent runs.

	# Executions			# Paths Discovered		
	Agamotto-R	Agamotto	**(Increase)**	Agamotto-R	Agamotto	**(Increase)**
ATLANTIC	147k	191k	**(30.1%)**	112	142	**(18.7%)**
RTL8139	152k	259k	**(70.5%)**	71	153	**(115.4%)**
STMMAC	137k	160k	**(16.6%)**	87	121	**(50.5%)**
SNIC	144k	153k	**(6.2%)**	8	8	**(0%)**

has potential to improve it further when the checkpoint management policies are optimized together with other aspects of the fuzzing algorithm of the underlying fuzzing engine.

Path Coverage. Table 4.4 shows, in fuzzing each driver, the maximum number of code paths discovered among all fuzzing instances. Agamotto's effectiveness is far more pronounced when the underlying fuzzing engine keeps discovering new, deeper code paths; the more checkpoints created by Agamotto in deep code paths, the more time it saves. In fuzzing ATLANTIC, RTL8139, and STMMAC, Agamotto covered substantially more paths than the baseline did in the same amount of time; by executing 32.8% more test cases on average, Agamotto covered 47.8% more paths. In fuzzing SNIC, however, AFL, the underlying fuzzing engine, only discovered only a limited number of paths. Still, Agamotto did execute 6.2% more test cases than the baseline did.

4.9 Limitations

Syzkaller supports a *multi-proc* mode, which runs multiple instances of a fuzzer within a single guest OS, increasing the fuzzing throughput. Agamotto does not support this mode currently, but this mode can be supported with a finer-grained checkpointing mechanism, e.g., via finer-grained virtual machine introspection or in-kernel checkpoints with kernel modifications [85]. This direction can be explored as future work. Note, however, that other aspects of Agamotto, e.g., checkpoint management and optimization techniques, would still apply even with such finer-grained checkpointing mechanism. The design choice of checkpointing at the virtual machine level allows Agamotto to support other virtual-machine-based kernel driver fuzzers (e.g., USBFuzz [133]) as was already demonstrated with the PCI-AFL experiments.

4.10 Conclusions

This chapter presented Agamotto, a system which transparently improves the performance of kernel driver fuzzers using a highly-optimized dynamic virtual machine checkpointing primitive. During a fuzzing run, Agamotto automatically checkpoints the virtual machine at fine-grained intervals and restores the virtual machine from these checkpoints allowing it to skip reboots on kernel panics and to *fast forward* through the time-consuming parts of test cases that are repeatedly executed. The evaluation of Agamotto in various USB and PCI fuzzing scenarios with two different fuzzers demonstrated the performance benefit that Agamotto can provide, as well as its adaptability.

Chapter 5

Conclusions and Future Work

This dissertation presented techniques that can be used to improve the precision and efficiency of dynamic analysis of OS kernels. To summarize, Chapter 2 first provided the current landscape of C and C++ dynamic analysis techniques, and described how they can be applied, in particular, to systems software running in kernel space. Next, Chapter 3 presented the design and implementation of a dynamic analysis and fuzzing framework for the peripheral input space of OS kernels, which uses a page table manipulation technique to hook and fuzz interactions between device drivers and peripherals at the level of individual memory accesses. This framework enables a precise analysis of the peripheral input space of OS kernels. Chapter 4 presented a virtual machine checkpointing technique that can accelerate OS kernel fuzzing, which builds on an insight that checkpointing and checkpoint restoration primitives can be used to *fast-forward* the execution of the guest virtual machine through system crashes as well as repeatedly executed code paths of the target program. This chapter now concludes the dissertation with a summary of the presented techniques and their key results, as well as a discussion of promising lines of future work.

5.1 Dynamic Analysis of Peripheral Input Space

Unlike other kernel subsystems, kernel-mode device drivers directly interface with peripheral devices, and can thus receive malicious input from peripheral devices when compromised. Due to the low-level nature of the interactions between device drivers and their corresponding peripheral devices, this interface requires tailored techniques to enable dynamic analysis.

Chapter 3 described the design and implementation of a dynamic analysis framework for this interface, which can either passively monitor or actively influence the interactions between device drivers and their corresponding peripherals. The key enabling technique was the page table manipulation technique that allows the user of the framework to examine the main processor's *every individual access* to PCI I/O mappings, thereby enabling an *accurate* adversarial analysis of the peripheral input space of device drivers. Notably, hooking and fuzzing individual accesses to I/O mappings enables detection of double-fetch bugs in addition to traditional memory corruption vulnerabilities. Previously unknown vulnerabilities including double-fetch bugs in two Wi-Fi drivers used in two flagship smartphones were found using the prototype implementation of the fuzzing framework, which demonstrates the effectiveness of the proposed approach.

5.2 Accelerating Dynamic Analysis of OS Kernels

OS kernels can be viewed as an event-driven, stateful system; the kernel's internal state transitions from one to another based on *events*, e.g., system call exceptions, timer exceptions, device interrupts, etc. This statefulness of OS kernels often requires trade-offs be made between high-speed and clean-state fuzzing.

Chapter 4 described how virtual machine checkpointing and restoration techniques can be optimized and used to achieve high-speed *and* clean-state OS kernel fuzzing. The key insight is that OS kernel fuzzers frequently execute similar test cases in a row, and that their performance can be improved by dynamically creating multiple checkpoints while executing test cases and skipping parts of test cases using the created checkpoints. The proposed checkpointing and restoration techniques were realized in a framework called Agamotto, which can transparently accelerate existing kernel fuzzers. The evaluation showed that Agamotto boosts the performance of the state-of-the-art kernel fuzzer, Syzkaller, by 66.6% on average in fuzzing 8 USB drivers, and an AFL-based PCI fuzzer by 21.6% in fuzzing 4 PCI drivers, without modifying their underlying input generation algorithm.

5.3 Future Work

5.3.1 Peripheral Attacks using Interrupts

Attackers who aim to compromise OS kernels through the PCI interface are not confined to the data plane—MMIO and DMA—of the PCI interface. Peripherals can also signal an interrupt to the OS kernel. Interrupts effectively increases the number of kernel code paths that can be reached by an attacker on the peripheral side. This is because any kernel code running with interrupts enabled can be interrupted by attacker-signaled interrupts; when they are interrupted the control is transferred to the corresponding interrupt handler. This interrupt concurrency, as well as the data contention caused by the concurrency, has been a major source of kernel failures and vulnerabilities. This adversarial capability needs a thorough investigation to further secure OS kernels.

5.3.2 Fuzzing Deeper Code Paths

Fuzzing deeper code paths remains an open problem not just in the area of OS kernel fuzzing, but in a more general area of fuzzing. A tremendous amount of efforts have gone into making fuzzers smarter; structure-aware or grammar-based fuzzing approaches have been explored to reduce the search space for fuzzers [44, 136, 66, 182, 8, 175]; different feedback signals—

e.g., fine-grained control-flow or additional data-flow information—have been used to more effectively guide feedback-guided fuzzing [59, 58, 146]; input mutation scheduling policies that use different heuristics or machine-learning-based algorithms have been explored to better direct fuzzing towards vulnerabilities or predefined source code locations [23, 22, 35, 189]; a line of work designed smarter mutators (or mutator selection policies) to improve mutational fuzzing [103, 138, 139]; another line of work explored combining symbolic execution with fuzzing to more explore code paths protected by narrow branch conditions [160, 185, 39, 134, 188].

A promising future research direction could be derived from the general idea that fuzzing deeper code paths requires modeling and exploring the target program's code paths that handle *valid* (sequences of) inputs. The code paths that handle valid inputs, once discovered, will lead the fuzzer to find many neighborhood code paths that handle different errors. Many analytical reasoning approaches that are based on traditional dynamic and static program analysis techniques (including symbolic execution approaches) have already been heavily explored thus far. Their results are impressive but still less than ideal; they are still stuck at the exponential cost of vulnerability discovery [21]. Data-centric approaches that are based on machine learning [24, 153, 138, 189, 62, 95, 122] could be an interesting direction to investigate further.

5.3.3 Checkpoint-Aware Fuzzing

The checkpointing primitive presented in Chapter 4 introduces a new dimension in the optimization space of fuzzing OS kernels or other event-driven, reactive systems in general. The prototype implementation was conservatively evaluated without modifying the fuzzing algorithm of underlying fuzzers; that is, only spatial and temporal localities that are already present in the fuzzing algorithm of state-of-the-art fuzzers were leveraged. Thus, various aspects of the fuzzing algorithm such as input selection and mutation strategies can be revisited; checkpoint-aware or -oblivious fuzzing algorithms can be a promising line of future work.

5.3.4 Kernel Dynamic Analysis

Chapter 2 provided an overview of available dynamic analysis techniques and tools for the C programming language family. Since OS kernel subsystems are typically written in low-level programming languages such as C, one may hypothesize that the techniques would work equally well when applied to kernel space. However, more research is required to prove or disprove this hypothesis. The peculiarities of the low-level programming in kernel space often make the kernel code deviate from the language standard, which in turn may complicate the analysis policy (see Section 2.4). Although there are several tools already available which perform dynamic analysis of kernel components, other user-space tools and techniques having different precision and performance characteristics, when ported to kernel space, may improve, or complement existing but limited kernel analysis tools.

Bibliography

[1] Facedancer21. `http://goodfet.sourceforge.net/hardware/facedancer21`.

[2] QEMU system emulation user's guide.

[3] TriforceAFL: AFL/QEMU fuzzing with full-system emulation, 2016.

[4] Trinity: Linux system call fuzzer, 2019. `https://github.com/kernelslacker/trinity`.

[5] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the USENIX Security Symposium*, 2009.

[6] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 7, 1996.

[7] N. Artenstein. BroadPwn: Remotely compromising Android and iOS via a bug in Broadcom's Wi-Fi chipsets. *Black Hat USA*, 2017.

[8] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert. NAUTILUS: Fishing for deep bugs with grammars. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[9] D. Aumaitre and C. Devine. Subverting Windows 7 x64 kernel with DMA attacks. *HITBSecConf Amsterdam*, 2010.

[10] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1994.

[11] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2006.

[12] T. Ball, E. Bounimova, R. Kumar, and V. Levin. SLAM2: Static driver verification with under 4% false alarms. In *Proceedings of the International Conference on Formal Methods in Computer Aided Design (FMCAD)*, pages 35–42, 2010.

[13] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In *Proceedings of the International Conference on Integrated Formal Methods (IFM)*, pages 1–20, 2004.

[14] I. Beer. pwn4fun spring 2014 - Safari - part II, 2014. `https://googleprojectzero.blogspot.com/2014/11/pwn4fun-spring-2014-safari-part-ii.html`.

[15] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, 2005.

[16] G. Beniamini. Over the air - vol. 2, pt. 2: Exploiting the Wi-Fi stack on Apple devices, 2017. `https://googleprojectzero.blogspot.com/2017/10/over-air-vol-2-pt-2-exploiting-wi-fi.html`.

[17] G. Beniamini. Over the air - vol. 2, pt. 3: Exploiting the Wi-Fi stack on Apple devices, 2017. `https://googleprojectzero.blogspot.com/2017/10/over-air-vol-2-pt-3-exploiting-wi-fi.html`.

[18] G. Beniamini. Over the air: Exploiting Broadcom's Wi-Fi stack (part 1), 2017. `https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html`.

[19] G. Beniamini. Over the air: Exploiting Broadcom's Wi-Fi stack (part 2), 2017. `https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_11.html`.

[20] P. Biswas, A. D. Federico, S. A. Carr, P. Rajasekaran, S. Volckaert, Y. Na, M. Franz, and M. Payer. Venerable variadic vulnerabilities vanquished. In *Proceedings of the USENIX Security Symposium*, 2017.

[21] M. Böhme and B. Falk. Fuzzing: On the exponential cost of vulnerability discovery. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020.

[22] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed greybox fuzzing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[23] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[24] K. Böttinger, P. Godefroid, and R. Singh. Deep reinforcement fuzzing. In *Proceedings of the IEEE Security and Privacy Workshops (SPW)*, 2018.

[25] S. Boyd-Wickizer and N. Zeldovich. Tolerating malicious device drivers in Linux. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2010.

[26] D. Bruening and Q. Zhao. Practical memory checking with Dr. Memory. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2011.

[27] N. Burow, D. McKee, S. A. Carr, and M. Payer. CUP: Comprehensive user-space protection for C/C++. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2018.

[28] J. Caballero, G. Grieco, M. Marron, and A. Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2012.

[29] C. Cadar, D. Dunbar, and D. 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 (OSDI)*, 2008.

[30] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2006.

[31] A. Cama. A walk with Shannon: A walkthrough of a pwn2own baseband exploit. *OPCDE Kenya*, 2018.

[32] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the USENIX Security Symposium*, 2015.

[33] Q. Casasnovas. [patch] kcov: add AFL-style tracing, 2016.

[34] O. Chang. Attacking the Windows NVIDIA driver, 2017. `https://googleprojectzero.blogspot.com/2017/02/attacking-windows-nvidia-driver.html`.

[35] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[36] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the USENIX Security Symposium*, 2005.

[37] V. Chipounov and G. Candea. Reverse engineering of binary device drivers with RevNIC. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2010.

[38] V. Chipounov, V. Kuznetsov, and G. 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.

[39] M. Cho, S. Kim, and T. Kwon. Intriguer: Field-level constraint solving for hybrid fuzzing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2019.

[40] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[41] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the USENIX Security Symposium*, 2004.

[42] P. Chubb. Linux kernel infrastructure for user-level device drivers. In *Linux Conference*, 2004.

[43] J. Corbet, A. Rubini, and G. Kroah-Hartman. Chapter 8. Allocating Memory. In *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[44] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna. DIFUZE: Interface aware fuzzing for kernel drivers. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[45] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. FormatGuard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the USENIX Security Symposium*, 2001.

[46] T. H. Dang, P. Maniatis, and D. Wagner. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In *Proceedings of the USENIX Security Symposium*, 2017.

[47] A. Davis. USB – undermining security barriers. *Black Hat USA*, 2011.

[48] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. HardBound: Architectural support for spatial safety of the C programming language. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[49] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2006.

[50] D. Dhurjati and V. Adve. Efficiently detecting all dangling pointer uses in production servers. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2006.

[51] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2012.

[52] G. J. Duck, R. H. Yap, and L. Cavallaro. Stack bounds protection with low fat pointers. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[53] G. J. Duck and R. H. C. Yap. Heap bounds protection with low fat pointers. In *Proceedings of the International Conference on Compiler Construction (CC)*, 2016.

[54] G. J. Duck and R. H. C. Yap. EffectiveSan: Type and memory error detection using dynamically typed C/C++. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[55] L. Duflot, Y.-A. Perez, G. Valadon, and O. Levillain. Can you still trust your network card? *CanSecWest*, 2010.

[56] F. C. Eigler. Mudflap: Pointer use checking for C/C++. In *Annual GCC Developers' Summit*, 2003.

[57] H. Finkel. The Type Sanitizer: Free yourself from -fno-strict-aliasing. In *LLVM Developers' Meeting*, 2017.

[58] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin1, D. Wu1, and Z. Chen. GREYONE: Data flow sensitive fuzzing. In *Proceedings of the USENIX Security Symposium*, 2020.

[59] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. CollAFL: Path sensitive fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2018.

[60] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[61] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[62] P. Godefroid, H. Peleg, and R. Singh. Learn&Fuzz: Machine learning for input fuzzing. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[63] Google. Found bugs, 2020. `https://github.com/google/syzkaller/blob/master/docs/found_bugs.md`.

[64] Google. KernelAddressSanitizer (KASAN), 2020. `https://github.com/google/kasan`.

[65] Google. KernelMemorySanitizer (KMSAN), 2020. `https://github.com/google/kmsan`.

[66] Google. syzkaller - kernel fuzzer, 2020. `https://github.com/google/syzkaller`.

[67] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe. TypeSan: Practical type confusion detection. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[68] H. Han and S. K. Cha. IMF: Inferred model-based fuzzer. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[69] P. B. Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, 1970.

[70] N. Hasabnis, A. Misra, and R. Sekar. Light-weight bounds checking. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2012.

[71] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *USENIX Winter Conference*, 1992.

[72] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Fault isolation for device drivers. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2009.

[73] J. Hertz and T. Newsham. A Linux system call fuzzer using TriforceAFL, 2016. `https://github.com/nccgroup/TriforceLinuxSyscallFuzzer`.

[74] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2016.

[75] Intel. Intel 64 and IA-32 architectures software developer's manual - Chapter 23 introduction to virtual-machine extensions. `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.pdf`.

[76] Intel. Pointer checker. `https://software.intel.com/en-us/node/522702`, 2015.

[77] ISO/IEC JTC1/SC22/WG14. ISO/IEC 9899:2011, Programming Languages — C, 2011.

[78] ISO/IEC JTC1/SC22/WG14. ISO/IEC 14882:2014, Programming Language C++, 2014.

[79] Y. Jeon, P. Biswas, S. Carr, B. Lee, and M. Payer. HexType: Efficient detection of type confusion errors for C++. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[80] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[81] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2002.

[82] M. Jodeit and M. Johns. USB device drivers: A stepping stone into your kernel. In *Proceedings of the European Conference on Computer Network Defense (EC2ND)*, 2010.

[83] R. W. Jones and P. H. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *International Workshop on Automatic Debugging*, pages 13–26, 1997.

[84] M. Jurczyk and G. Coldwind. Identifying and exploiting Windows kernel race conditions via memory access patterns. 2013.

[85] A. Kadav, M. J. Renzelmann, and M. M. Swift. Fine-grained fault tolerance using device checkpoints. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[86] S. Keil and C. Kolbitsch. Stateful fuzzing of wireless device drivers in an emulated environment. *Black Hat Japan*, 2007.

[87] S. C. Kendall. Bcc: Runtime checking for C programs. In *USENIX Summer Conference*, 1983.

[88] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer. SGXBounds: Memory safety for shielded execution. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2017.

[89] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2010.

[90] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(2):190–222, 1983.

[91] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[92] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee. Preventing use-after-free with dangling pointers nullification. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[93] B. Lee, C. Song, T. Kim, and W. Lee. Type casting verification: Stopping an emerging attack vector. In *Proceedings of the USENIX Security Symposium*, 2015.

[94] J. Lee, Y. Kim, Y. Song, C.-K. Hur, S. Das, D. Majnemer, J. Regehr, and N. P. Lopes. Taming undefined behavior in LLVM. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[95] S. Lee, H. Han, S. K. Cha, and S. Son. Montage: A neural network language model-guided JavaScript engine fuzzer. In *Proceedings of the USENIX Security Symposium*, 2020.

[96] J. Lettner, D. Song, T. Park, S. Volckaert, P. Larsen, and M. Franz. PartiSan: Fast and flexible sanitization via run-time partitioning. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2018.

[97] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[98] J. Liedtke. On micro-kernel construction. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[99] LLVM Developers. Control flow integrity, 2020. `https://clang.llvm.org/docs/ControlFlowIntegrity.html`.

[100] LLVM Developers. libFuzzer – a library for coverage-guided fuzz testing, 2020. `https://llvm.org/docs/LibFuzzer.html`.

[101] LLVM Developers. UndefinedBehaviorSanitizer, 2020. `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`.

[102] A. Loginov, S. H. Yong, S. Horwitz, and T. Reps. Debugging via run-time type checking. In *International Conference on Fundamental Approaches to Software Engineering*, pages 217–232, 2001.

[103] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *Proceedings of the USENIX Security Symposium*, 2019.

[104] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.

[105] A. Markuze, A. Morrison, and D. Tsafrir. True IOMMU protection from DMA attacks: When copy is faster than zero copy. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[106] A. Markuze, I. Smolyar, A. Morrison, and D. Tsafrir. DAMN: Overhead-free IOMMU protection for networking. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[107] P. E. McKenney. Linux-kernel memory model, 2015.

[108] Microsoft Corporation. How to use Pageheap.exe in Windows XP, Windows 2000, and Windows Server 2003, 2000.

[109] Microsoft Corporation. Memory management for Windows drivers, 2017.

[110] Microsoft Corporation. Static driver verifier, 2019.

[111] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[112] M. Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. *BlueHat IL*, 2019.

[113] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[114] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[115] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. CETS: Compiler enforced temporal safety for C. In *Proceedings of the International Symposium on Memory Management (ISMM)*, 2010.

[116] S. Nagy and M. Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[117] National Vulnerability Database. NVD - CVE-2009-1897, 2009. `https://nvd.nist.gov/vuln/detail/CVE-2009-1897`.

[118] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2002.

[119] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine*, 11, 2001.

[120] N. Nethercote and J. Fitzhardinge. Bounds-checking entire programs without recompiling. In *Proceedings of the ACM SIGPLAN Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE)*, 2004.

[121] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2005.

[122] N. Nichols, M. Raugas, R. Jasper, and N. Hilliard. Faster fuzzing: Reinitialization with deep neural models. *arXiv preprint arXiv:1711.02807*, 2017.

[123] K. Nohl and J. Lell. BadUSB – on accessories that turn evil. *Black Hat USA*, 2014.

[124] V. Nossum and Q. Casasnovas. Filesystem fuzzing with American Fuzzy Lop. *Vault*, 2016.

[125] Open Virtualization Alliance. Linux kernel virtual machine. `https://www.linux-kvm.org`.

[126] Open Virtualization Alliance. Virtio. `https://www.linux-kvm.org/page/Virtio`.

[127] S. Pailoor, A. Aday, and S. Jana. Moonshine: Optimizing OS fuzzer seed selection with trace distillation. In *Proceedings of the USENIX Security Symposium*, 2018.

[128] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten years later. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[129] J. Pan, G. Yan, and X. Fan. Digtool: A virtualization-based framework for detecting kernel vulnerabilities. In *Proceedings of the USENIX Security Symposium*, 2017.

[130] Parasoft. Insure++. `https://www.parasoft.com/product/insure`, 2017.

[131] H. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software — Practice and Experience*, 27(1):87–110, 1997.

[132] J. Patrick-Evans, L. Cavallaro, and J. Kinder. POTUS: Probing off-the-shelf USB drivers with symbolic fault injection. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[133] H. Peng and M. Payer. USBFuzz: A framework for fuzzing USB drivers by device emulation. In *Proceedings of the USENIX Security Symposium*, 2020.

[134] H. Peng, Y. Shoshitaishvili, and M. Payer. T-Fuzz: Fuzzing by program transformation. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2018.

[135] B. Perens. Electric fence malloc debugger, 1993.

[136] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 2019.

[137] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Current trends in Concurrency*, pages 510–584. Springer, 1986.

[138] M. Rajpal, W. Blum, and R. Singh. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596*, 2017.

[139] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. VUzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[140] M. J. Renzelmann, A. Kadav, and M. M. Swift. SymDrive: Testing drivers without devices. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[141] M. F. Ringenburg and D. Grossman. Preventing format-string attacks via automatic and efficient dynamic checking. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2005.

[142] J. M. Rushby. Design and verification of secure systems. *ACM SIGOPS Operating Systems Review*, 15(5):12–21, 1981.

[143] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2004.

[144] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[145] M. Schulz, D. Wegemer, and M. Hollick. The Nexmon firmware analysis and modification framework: Empowering researchers to enhance Wi-Fi devices. *Computer Communications*, 2018.

[146] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In *Proceedings of the USENIX Security Symposium*, 2017.

[147] S. Schumilo, R. Spenneberg, and H. Schwartke. Don't trust your USB! how to find bugs in USB device drivers. *Black Hat Europe*, 2014.

[148] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the European Software Engineering Conference Held Jointly with ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005.

[149] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2012.

[150] F. J. Serna. MS08-061 : The case of the kernel mode double-fetch, 2008.

[151] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2005.

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

[153] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana. NEUZZ: Efficient fuzzing with neural program learning. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[154] Solar Designer. Getting around non-executable stack (and fix). Email to the Bugtraq mailing list, Aug. 1997.

[155] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz. PeriScope: An effective probing and fuzzing framework for the hardware-OS boundary. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[156] D. Song, F. Hetzelt, J. Kim, B. B. Kang, J.-P. Seifert, and M. Franz. Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In *Proceedings of the USENIX Security Symposium*, 2020.

[157] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. SoK: Sanitizing for security. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[158] J. L. Steffen. Adding run-time checking to the portable C compiler. *Software: Practice and Experience*, 22(4):305–316, 1992.

[159] E. Stepanov and K. Serebryany. MemorySanitizer: Fast detector of uninitialized memory use in C++. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2015.

[160] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[161] P. Stewin and I. Bystrov. Understanding DMA malware. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2012.

[162] J. V. Stoep and S. Tolvanen. Year in review: Android kernel security. *Linux Security Summit*, 2018.

[163] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *Proceedings of the European Workshop on System Security (EuroSec)*, 2009.

[164] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[165] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[166] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[167] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2013.

[168] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *Proceedings of the USENIX Security Symposium*, 2018.

[169] T. Tsai and N. Singh. Libsafe 2.0: Detection of format string vulnerability exploits. *White paper, Avaya Labs*, 2001.

[170] E. van der Kouwe, V. Nigade, and C. Giuffrida. DangSan: Scalable use-after-free detection. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2017.

[171] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos. Memory errors: The past, the present, and the future. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2012.

[172] D. Vyukov. kernel: add kcov code coverage, 2016.

[173] D. Vyukov. syzbot and the tale of thousand kernel bugs. *Linux Security Summit*, 2018.

[174] D. Vyukov. READ_ONCE and WRITE_ONCE, 2019.

[175] J. Wang, B. Chen, L. Wei, and Y. Liu. Skyfire: Data-driven seed generation for fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2017.

[176] R.-P. Weinmann. All your baseband are belong to us. *DeepSec*, 2010.

[177] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, 1974.

[178] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim. Precise and scalable detection of double-fetch bugs in OS kernels. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2018.

[179] W. Xu, D. C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, 2004.

[180] W. Xu, S. Kashyap, C. Min, and T. Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[181] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim. Fuzzing file systems via two-dimensional input space exploration. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[182] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang. ProFuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[183] Y. Younan. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[184] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen. PAriCheck: An efficient pointer arithmetic checker for C programs. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2010.

[185] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the USENIX Security Symposium*, 2018.

[186] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti, et al. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[187] M. Zalewski. American Fuzzy Lop, 2019. `http://lcamtuf.coredump.cx/afl`.

[188] L. Zhao, Y. Duan, H. Yin, and J. Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[189] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen. FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *Proceedings of the USENIX Security Symposium*, 2020.