École Polytechnique Fédérale de Lausanne

Hardening and Testing Privileged Code through Binary Rewriting

Master Thesis

by Matteo Rizzo

Chemin de la Praz 9 — CH-1052 Le Mont-sur-Lausanne

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer

Thesis Advisor

Julien Voisin

External Expert

EPFL IC IINFCOM HEXHIVE

BC 160 (Bâtiment BC) - Station 14 - CH-1015 Lausanne

January 16, 2020

La vita non va dietro alle carte. Le carte vanno dietro alla vita.

— Anonymous

Dedicated to my father, Michele, my mother, Roberta, and my sister, Marianna, who have followed and helped me immensely during the last three years. This work would have not been possible without their support.

Acknowledgments

I want to thank my advisor, Prof. Mathias Payer, for his support, close guidance, and for all the feedback I received. I enjoyed the four months I spent at HexHive working with him and I thank him for pushing me to strive for more. He is a brilliant researcher and I wish him the best in his career. I also want to thank Prof. George Candea and Prof. Katerina Argyraki for introducing me to the world of systems research and for supporting me throughout my second year at EPFL.

I want to thank my family for their ever-present support and for believing in me and letting me be here. Getting through my Master's studies has been one of my greatest challenges, and I would not have made it without their emotional support.

I want to thank the members of HexHive for welcoming me during my Master Project and offering their guidance and feedback on my research and presentations. I learned a lot about research and about security in general during our conversations in these four months. Good luck to everyone for your PhDs and careers!

Finally, I want to thank the members of polygl0ts, EPFL's Capture The Flag (CTF) team, for accompanying me in my journey into systems security. Playing Capture The Flag with them has not only been great fun but it has also taught me much about the applied side of security. The knowledge that I gained by playing CTF has been instrumental in my research work. I'm looking forward to playing even more CTFs with you!

Lausanne, January 16, 2020 Matteo Rizzo

Abstract

Kernel code contains memory corruption vulnerabilities and is an appealing target for attackers, who can abuse its flaws to escalate privileges and bypass sandboxing. Kernel memory corruption accounted for more than 20% of CVEs in the Android Platform between May 2017 and May 2018. Fuzzing is an effective strategy to discover software flaws and has found thousands of bugs in security-critical kernel components.

Instrumentation boosts the efficacy of fuzzing by enabling the fuzzer to select more interesting test cases and detect more bugs during testing. Coverage-tracking instrumentation lets the fuzzer select better inputs by providing feedback about each test case. Sanitizing instrumentation lets the fuzzer detect more bugs by actively checking for security violations.

State-of-the-art instrumentation like Address Sanitizer is implemented as a compiler pass, which only works on C source code and requires a recent compiler. Compiler-based instrumentation is unsuitable for kernel components that are closed-source, that use hand-written assembly, or that are incompatible with an instrumenting compiler. Existing methods to instrument binary kernel code have prohibitive runtime overhead (10-100x).

We show that static rewriting is a viable way to instrument binary kernel-mode code at low runtime overhead. RetroWrite shows that static rewriting is feasible for position-independent user-mode binaries. We present `kRetroWrite`, a static rewriter for kernel-mode binaries and we use it to instrument binaries for both coverage tracking and memory sanitization.

The performance impact of our binary instrumentation is similar to that of source-based instrumentation in our evaluation. `kRetroWrite` rewrites individual kernel modules, allowing us to selectively instrument the kernel components that we are targeting. Our instrumentation passes are compatible with their source-based counterparts and with off-the-shelf kernel fuzzers.

Contents

Chapter 1

Introduction

Modern operating system kernels are large and complex. Security researchers routinely find vulnerabilities in OS kernels, and Linux alone received more than 250 CVEs in 2019 [25]. Security vulnerabilities in the OS kernel are a threat to the safety of the entire system because the kernel runs at the highest privilege level, and an attacker that compromises it gains unrestricted access to the machine. The kernel is also an appealing target for attackers trying to evade userspace exploit mitigations and process sandboxing: mainstream web browsers like Chrome prevent processes that render untrusted web content from accessing kernel subsystems that have a history of vulnerabilities [16]. Even though mainstream kernels ship with mitigations against several kinds of attacks, those mitigations are often ineffective and attackers routinely bypass them.

Fuzzing is an effective way to discover bugs in complex software systems and in recent years it has gained traction in both research and industry. Large scale fuzzing campaigns such as Google's OSS-fuzz and syzbot have found thousands of security-critical bugs in mainstream software, and continue to do so. Fuzzing is easy to use even for non-experts compared to other automatic bug finding methods: it requires little human effort besides the implementation of a harness, and the selection of the initial seed corpus. It is also embarrassingly parallel because each generated test case can be executed independently of the others: it scales to large server

farms with thousands of CPU cores, further adding to its effectiveness.

Fuzzers are most effective if they leverage feedback about the generated test cases and they have a way to detect crashes: this is typically accomplished using instrumentation. Coverage-guided fuzzers use instrumentation to track the coverage of each test case. The fuzzer selects inputs that produce new coverage for further mutation which lets it discover new parts of the program without supervision. Fuzzers are also often paired with sanitizers, which instrument the target to actively check for security violations. Sanitization uses instrumentation to detect bugs that the fuzzer would miss.

Unfortunately, it is much more challenging to instrument a program when the source code is not available. In spite of the booming popularity of open source software, closed source software is still the norm in many cases. For example more than 90% of desktop PCs and laptops run Windows or macOS, both proprietary OSs [43]. Even when source code for the majority of the system is available, many devices require closed source drivers: for example this is the case for GPUs and Android phones. The main difficulty in instrumenting a binary is that much of the information that exists in the source code such as typing, control flow, and the distinction between code and data is discarded during the compilation process. Particularly, the compiler translates all references between code and static data to hard-coded constant offsets: this means that simply inserting new instructions into the target makes it malfunction because it breaks all references. There are broadly two approaches to instrumenting binaries: static rewriting and dynamic rewriting.

Existing binary instrumentation toolkits for the kernel use dynamic rewriting, which has poor performance, and therefore they are poorly suited for fuzzing. Dynamic rewriters run the rewriting engine side-by-side with the target, and translate and instrument the target's code as needed. Dynamic rewriting does not rely on heuristics because it can gather information about the target's behavior at runtime. Runtime information allows dynamic rewriting to deal with arbitrary code and to scale well to large and complex targets. Unfortunately dynamically translating the target introduces overhead. Furthermore dynamic rewriters cannot afford to run complex

analysis and optimization passes because they would further increase this overhead, making the rewritten code less than optimal: AFL-QEMU introduces a 10-100x runtime overhead [12] to instrument a binary for coverage tracking compared to compiler-based instrumentation. Only a fraction of the test cases generated by a fuzzer trigger bugs in the target, and therefore a faster target directly translates into more bugs found in the same amount of time.

Static rewriting instruments the target ahead of time and has lower overhead than dynamic rewriting, but scales poorly because it relies on complex analysis [49, 50]. A static rewriter does not need to run side-by-side with the target, and can do more expensive optimizations offline: therefore it can have lower overhead than a dynamic rewriter. However it cannot take advantage of runtime information and is forced to rely on static analysis. The main difficulty for static rewriters is distinguishing between references and scalars, which is known to be undecidable statically. Static rewriters are forced to use unsound heuristics which increase complexity and do not scale to large targets.

Recent work [12] presented RetroWrite, a zero-cost principled static rewriter, and efficient static instrumentation for binaries. RetroWrite leverages PC-relative addressing to disambiguate references from other constants in a binary. The authors observe that disambiguating scalars from references is possible in practice for *position-independent code* (PIC) when the target ISA supports PC-relative addressing. This covers a broad range of targets, including shared libraries and most programs on the x86_64 architecture. The authors of RetroWrite use it to instrument binaries for coverage tracking and sanitization and achieve compiler-level performance.

RetroWrite and its instrumentation passes only support userspace binaries and do not work on the kernel. RetroWrite's symbolizer only handles userspace PIE binaries, which have a different format from kernel modules. Furthermore, RetroWrite's coverage and binary ASan instrumentation passes are designed for user-space binaries and do not work in the kernel. To address this limitation we develop `kRetroWrite`, a static rewriter that can instrument kernel code at high performance for fuzzing. `kRetroWrite` takes as input a binary kernel module and produces a *symbolized assembly* listing that can be reassembled into a working module. Symbolized assem-

bly uses labels rather than hard-coded offsets to refer to static data and code: instrumentation passes can freely insert and remove instructions in the symbolized listing without breaking the code. We base our system on RetroWrite and change the symbolizer to support the executable format used by Linux modules. As a proof of concept we implement two binary instrumentation passes for Linux modules: (1) A binary kcov pass that provides coverage information for the instrumented code to a userspace agent. (2) A binary *Kernel Address Sanitizer* (KASan) [46] pass that detects memory errors such as buffer overflows in the instrumented module and produces a bug report. Both passes seamlessly integrate with their source-based counterparts and can be used with off-the-shelf kernel fuzzers. To the best of our knowledge, no publicly available memory sanitizer works with binary Linux modules. kmemcheck [17] marks kernel pages as not present to trap on every memory access, and it can detect use-after-free and uses of uninitialized memory. kmemcheck was removed from the kernel in 2017 [27] and replaced with KASan, which is 6x faster [23] and supports multiple CPUs.

`kRetroWrite` lets users target specific kernel components with instrumentation, and works even when compiler-based approaches are not applicable. Modern kernels build only essential components into the main kernel executable, and place other components in *kernel modules* which are loaded on demand. When fuzzing a specific kernel components, users often want to only apply instrumentation to that component: targeted instrumentation improves performance and guides the fuzzer towards the component under test. `kRetroWrite` instruments individual modules rather than the entire kernel and therefore it lets users choose the kernel components where they would like to pay for instrumentation overhead. `kRetroWrite`'s utility is not limited to closed-source modules: compiler-based instrumentation does not work on hand-written assembly and on code that requires a legacy toolchain. Our instrumentation works at the assembly level and therefore it can cover such cases.

Our experiments demonstrate that `kRetroWrite` instrumentation is competitive with compiler-based instrumentation in both runtime performance and coverage. We are only about 8% slower at fuzzing Btrfs [37] while achieving only 6% less coverage than source-based instrumentation. `kRetroWrite` scales to large modules and instruments a 32MiB kernel module in

about 1 minute.

In the rest of this report, we give an overview of the background, then describe our three contributions:

- `kRetroWrite`, a static rewriter for kernel code that can instrument binary kernel modules. To our knowledge this is the first system for sound static rewriting of kernel code. `kRetroWrite` shows that static rewriting is practical for kernel modules and that the RetroWrite approach is applicable to the kernel,
- An instrumentation pass that allows userspace agents to collect coverage information about a binary Linux module,
- An instrumentation pass that adds KASan checks to binary Linux modules to detect memory safety violations.

We then evaluate our `kRetroWrite` prototype, discuss its limitations, discuss related work, and finally present our conclusions.

Chapter 2

Background

In this chapter we provide an overview of fuzzing, sanitization, and binary rewriting. We first discuss the core ideas behind fuzzing, and why it is more difficult to apply it to kernel code. Next we discuss the challenges of binary rewriting and discuss RetroWrite's approach. Finally we discuss coverage tracking and sanitization.

2.1 Fuzzing

Fuzzing is a method to test programs automatically that uses randomly-generated test cases. Modern software systems are large and complex, and the software running on a modern desktop computer comprises hundreds of millions of lines of code. Unfortunately, the scale of these systems makes it hard to comprehensively reason about their correctness. Therefore most software contains bugs, some of which are security bugs that allow an attacker to compromise the system. The security community uncovers thousands of vulnerabilities every year, and malicious actors routinely exploit security holes to their advantage. A fundamental problem in securing complex software is that manual testing and auditing are time-consuming and do not scale. Fuzzing addresses this problem by generating test cases automatically. Fuzzing consists in automatically generating random inputs, feeding them to the target, and monitoring it for

crashes or other unexpected behavior. Fuzzing does not require any human input besides the initial setup and therefore it is scalable. Furthermore it is embarrassingly parallel and can be run on an arbitrary number of cores. Fuzzing regularly finds thousands of bugs in popular software packages: Google's OSS-fuzz project continuously fuzzes open-source software and reported more than 1000 security bugs in 2019 [34].

Fuzzing kernels presents additional challenges compared to userspace applications.

2.1.1 Userspace fuzzing

A crashing userspace fuzzing target is terminated by the kernel, which then notifies the fuzzer. Userspace processes run in an isolated context and do not have direct access to the hardware. A misbehaving userspace process can not crash the entire machine unless the kernel is also buggy. Userspace programs are mostly deterministic, unless they use multiple threads and synchronization or a random number generator. Therefore it is easier to reproduce a crashing test case in userspace. There are many off-the-shelf tools to fuzz userspace applications, and many resources online on how to use them. Userspace fuzzers are increasingly being used as part of the regular development cycle of applications. Attackers are also more likely to focus on attacking userspace applications rather than the kernel because vulnerabilities are more stable and easier to exploit.

2.1.2 Kernel fuzzing

By contrast, fuzzing kernel code is harder than fuzzing in userspace. Firstly, when the kernel crashes the entire machine goes down. This requires that the fuzz target is either run in a virtual machine or on a dedicated machine to limit the impact of a system crash. The VM also needs to be rebooted every time the system crashes which impacts performance. Secondly the flow of execution in the kernel is non-deterministic, even on a single-CPU machine, because of interrupts. This makes reproducing bugs and the execution flows of system calls more complicated. There

are also fewer tools to fuzz kernels and they require more complicated setups such as compiling a custom kernel and using special configurations.

2.2 Binary rewriting

Instrumenting a program consists in inserting short snippets of code to monitor its behavior. Instrumentation enhances the effectiveness of a fuzzer by providing feedback about the generated inputs and detecting more bugs. Instrumentation is typically added at compile time using a compiler pass, but this technique cannot be used in many situations. For example, it cannot be used when the target is only available as a binary, or when the target requires a compiler that does not support adding instrumentation. Furthermore passes for C/C++ compilers do not instrument hand-written assembly.

The most straightforward approach to instrumenting a binary is to simply insert new instructions into the target. Unfortunately this approach does not work except for trivial programs that contain no references to code or static data. When the compiler translates source code to machine code it replaces such references with hardcoded constants. Inserting new instructions in a compiled program invalidates the values of the constants, and breaks it. We say that compiled binaries have a *rigid* structure because moving any code or data breaks the references. A binary rewriter must ensure that all references remain correct after inserting the instrumentation. There are two approaches to rewriting binaries: static rewriting and dynamic rewriting.

2.2.1 Dynamic rewriting

Dynamic rewriting rewrites the target's code while the target itself is executing. The target runs side-by-side with a rewriting engine which translates its code on demand. Dynamic rewriters have access to dynamic information such as the path that the execution has taken. They can leverage this information to recognize and adjust references in the target. Dynamic rewriters do not need to analyze the entire target program, but only the part that is executing. Dynamic

```
mov [rax + rbx*8], rdi
dec rbx
jnz -7
```

Listing 2.1: Original code

```
mov [rax + rbx*8], rdi
inc rcx
dec rbx
jnz -7
```

Listing 2.2: Instrumented code

Figure 2.1: These assembly code snippets show why simply inserting instructions in a binary causes it to break. The original code jumps back to the beginning of the loop. The instrumented code jumps to the inserted instrumentation and breaks. Adjusting the jump offset to account for the instrumentation solves the problem. A binary rewriter must find and adjust all references. In general, this problem is undecidable for a static rewriter.

information allows such rewriters to scale to large targets, but comes at the cost of performance: the rewriter has to run side-by-side with the target and consumes execution resources. Moreover binary rewriters cannot perform complex analysis on the target because it would further increase the runtime overhead. The most efficient dynamic rewriters, like PIN [28] and DynamoRIO [5] have between 10% and 20% rewriting overhead, with no instrumentation.

Existing work shows that dynamic rewriters such as QEMU [3] can instrument both userspace binaries [1] and kernels [35] for fuzzing. In both cases dynamic rewriting introduces significant overhead, which is undesirable when fuzzing.

2.2.2 Static rewriting

Static rewriters process the target statically and emit an instrumented binary. They could have no runtime overhead in principle because they do not have to execute at the same time as the target and can perform expensive optimizations offline. However static rewriters suffer from the lack of runtime information and have to resort to static analysis to identify references in the target. Static analysis is complex and introduces imprecision, which makes it challenging to scale these rewriters to large targets. Distinguishing references from scalars and structure offsets statically is known to be an undecidable problem in the general case, and static rewriters have to resort to heuristics.

In RetroWrite, Dinesh et al. [12] observe that distinguishing references from scalars and structure offsets is possible for position-independent executables. Position-independent executables are the norm on modern platforms because they allow the operating system to load each binary at a random address as a mitigation against software exploits (*Address Space Layout Randomization*, or ASLR). Modern CPU architectures include architectural support for PIE in the form of PC-relative addressing. Binaries compiled for these architectures refer to other code or data using an offset from the current value of the PC, called *RIP-relative addressing* on x86_64. PC-relative addressing lets static analysis identify references to other code and static data because they always use PC-relative addressing while other references never use it. This makes it possible to disambiguate references from scalars and structure offsets statically. RetroWrite lifts the target binary to assembly and replaces all hardcoded constant references with symbolic references to labels. Data-to-code and data-to-data references always use relocations in PIE binaries and can be symbolized by resolving the relocation. The resulting symbolized assembly can be instrumented by inserting new instructions into the listing. RetroWrite shows that static rewriting scales to complex targets in the common case of position-independent code.

2.3 Coverage-guided fuzzing

Coverage-guided fuzzing addresses the inability of a fuzzer to cover new parts of the target by leveraging coverage feedback. One of the main issues with fuzzing is that the fuzzer tends to get stuck without achieving high coverage of the program under test (*coverage wall*). This limits the effectiveness of the fuzzer because the test cases can only trigger bugs if they cause the program to execute the buggy code. One of the reasons why fuzzers run into the coverage wall is that fuzz targets often accept highly structured input and it is unlikely that randomly generated data will satisfy the constraints of the input format. The program under test rejects such invalid inputs early in the parsing stage and the fuzzer fails to trigger deep bugs. *Smart fuzzing* mitigates this by providing the fuzzer with a description of the input format, but creating such descriptions is time-consuming.

A coverage-guided fuzzer learns the input format by using coverage feedback. The test case generator collects coverage from the target and attempts to provide inputs that discover new paths. Such inputs are deemed *interesting* and added to a corpus. The fuzzer generates new inputs by randomly mutating an input from the corpus. AFL [51] popularized coverage-guided fuzzing and showed that it can find real-world vulnerabilities [52]. AFL tracks branch coverage by instrumenting the target at compile time. Other fuzzers collect coverage with dynamic rewriting (e.g., AFL-QEMU [1]), or hardware-assisted feedback (e.g., kAFL [38]).

kcov [21] is an implementation of coverage tracking built into the Linux kernel. It uses a compiler pass to insert a call to a coverage collection function at the beginning of every basic block. kcov exposes coverage information to userspace programs through a special file. kcov was added to Linux to support fuzzing the kernel from user space. It does not instrument some parts of the kernel such as interrupt handling and the scheduler to ensure that the collected coverage is as close as possible to being deterministic as a function of system call inputs.

2.4 Address Sanitizer

Sanitization helps a fuzzer find more bugs by actively checking for security violations. The most straightforward way to detect bugs in a target is to look for crashes. This is simple but imperfect because many exploitable bugs do not crash the target, or do so only under specific circumstances.

Sanitization augments fuzzing by using instrumentation enforce a security policy, such as memory safety. This lets the fuzzer detect buggy executions earlier and more reliably. *Address Sanitizer* (ASan) [41] detects memory corruption and is the most widely used sanitizer. Memory corruption is endemic in C/C++ code bases and is arguably the most dangerous class of security bugs. It accounted for more than 60% of CVEs in the Android platform between May 2017 and May 2018 [40]. ASan uses a special region of memory called *shadow memory* to track valid addresses and instruments every memory access to check the shadow. ASan adds *red*

zones around every variable that trigger a security violation when accessed. Red zones help detect buffer overflows more reliably. Memory belonging to a red zone is said to be *poisoned*. ASan instruments the memory allocator to add red zones around heap-allocated objects, and instruments function prologues and epilogues to poison and unpoison memory around stack-allocated objects. ASan introduces an average slowdown of 73% and an average memory usage increase of 3.37x on the SPEC CPU2006 benchmark suite [41]. Despite the performance overhead, it is widely used for fuzzing and has founds thousands of bugs in large software systems like Chromium. ASan has no false positives.

Kernel Address Sanitizer (KASan) [46] is a version of ASan that works on kernel code. Kernel code is more difficult to instrument because it uses its own memory allocator and resides in high memory. Furthermore some parts of the kernel are timing-critical or expected to access memory out-of-bounds (e.g., when traversing the stack) and must not be instrumented. Userspace ASan is a generic compiler pass and library that can be used on most userspace applications by switching on a compiler flag, but KASan is tightly integrated into the kernel's memory subsystem.

Chapter 3

Design

We now provide an overview of `kRetroWrite`: our design goals, an overview of the system, the design of the symbolizer, and the design of the instrumentation passes.

3.1 Goals

Our goal is to provide a platform to write static instrumentation passes for binary kernel modules. `kRetroWrite` should not restrict users to the passes we implement but instead it should provide a foundation for our users to build their binary instrumentation. `kRetroWrite`'s core should take care of the binary analysis and rewriting part: that is, it should identify and fix up references, and provide the instrumentation with the result of the analysis. Analyses performed by the core should scale to real-world kernel components and not rely on source code at all. Furthermore the rewriting process itself should introduce no runtime overhead. No existing tools satisfy these requirements: they either require source code (e.g., kcov [21], KASan [46]), or use dynamic rewriting, which has high rewriting overhead (e.g., TriforceAFL [35]).

We also set the following goals for our instrumentation passes: they should provide similar capabilities to source-based passes and integrate with existing compiler-based instrumentation.

We do not want to force our users to write custom kernel fuzzers or make our binary instrumentation mutually exclusive with its source-based counterpart. Modules instrumented with our passes should work side-by-side with source-instrumented modules and support existing kernel fuzzers. This would allow `kRetroWrite` users to tap into a rich ecosystem of existing instrumentation without being constrained to source-available modules compatible with modern compilers.

3.2 System overview

`kRetroWrite` is a static rewriter for kernel code, based on RetroWrite. Static rewriting fulfills our goal of low runtime overhead, but generally suffers from poor scalability due to the requirement for heavyweight static analysis. RetroWrite [12] shows that scalable and sound static rewriting is indeed possible in the common case of PIC binaries. PIC is widely used in modern systems because it is required by shared libraries and exploit mitigations such as ASLR. Major Linux distributions enable PIC by default [48] [15] [31] and recent versions of Android forbid non-PIC binaries [39]. Modern OSs implement ASLR both for user applications and for the kernel, and therefore modern kernels are also position-independent. We show that RetroWrite's approach is not just applicable to user applications, but to the kernel as well. RetroWrite fulfills our goals of scalable binary instrumentation at low runtime overhead.

Unfortunately we cannot simply apply RetroWrite to the kernel, as it is designed to rewrite user applications. Kernel executables use a different format and different ABI that RetroWrite's symbolizer does not support. RetroWrite's instrumentation passes are also not designed to work in the kernel: AFL-RetroWrite communicates with a fuzzer process which would not work in a kernel; RetroWrite's binary ASan links with `libASan` which does not exist in the kernel and uses userspace addresses rather than kernel addresses for the shadow memory. Therefore we have to adapt RetroWrite's symbolizer and instrumentation passes to the kernel.

Like RetroWrite, `kRetroWrite` is structured in multiple parts:

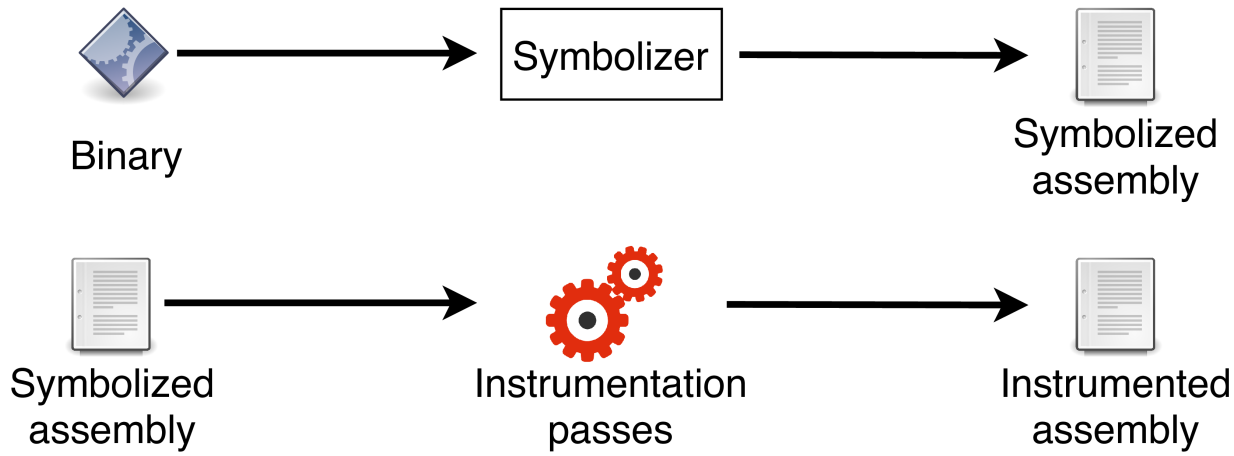

Figure 3.1: The `kRetroWrite` pipeline. The symbolizer produces symbolized assembly which can be reassembled into a binary. Instrumentation passes add instrumentation to the symbolized assembly.

- The *symbolizer* lifts a binary to *symbolized assembly*, where all hard-coded offsets are replaced by symbolic references to labels;
- A set of analysis passes recover information about the code, such as an approximate control flow graph and register liveness;
- A set of instrumentation passes instrument the symbolized assembly file for fuzzing and sanitization.

A symbolized assembly file can be turned back into a binary using an off-the-shelf assembler. This modular architecture allows us to add new instrumentation passes with little effort.

Our prototype rewrites individual Linux kernel modules rather than an entire kernel. There are three reasons for this decision:

- Rewriting individual modules allows us to target specific kernel components with instrumentation. Instrumentation introduces runtime overhead and fuzzing campaigns often target a specific component, not the whole kernel. `kRetroWrite` users can minimize runtime overhead by only instrumenting the components that their fuzzing campaign is

targeting

- In Linux's case, proprietary components are not compiled into the kernel due to licensing constraints. Instead they are distributed as proprietary modules that are loaded into the kernel. The source for the kernel itself is available and can be instrumented at compile time.
- We can compile the main kernel with support for kcov and KASan. This compiles the infrastructure needed to support the instrumentation into the main kernel. For example it replaces the allocator with one that inserts red zones and allocates memory for the KASan shadow. This is analogous to linking with `libASan` in userspace.

In short, instrumenting individual modules allows us to target kernel components more precisely, and at the same time minimizes the complexity of our tool by reusing Linux's existing instrumentation infrastructure.

3.3 Symbolizer

The symbolizer lifts a binary to symbolized assembly. It finds all references to code and static data and replaces them with references to labels. Symbolization enables rewriting because it removes hard-coded offsets to other parts of the binary. We identify four issues with RetroWrite's symbolizer that prevent it from working on kernel modules: the presence of relocations in code sections, the presence of self-modifying code in the kernel, the memory layout of kernel modules, and the presence of multiple code sections.

3.3.1 Code relocations

Unlike userspace binaries, kernel modules use relocations that overwrite instructions in the code section to import symbols from other modules or from the main kernel. The kernel computes

```
lea rax, [rip + 0x1234]
call 0x100
dec rcx
jnz -15
```

Listing 3.1: Original assembly code

```
loop1:
        lea rax, [data1]
        call func1
        dec rcx
        jnz loop1
```

Listing 3.2: Symbolized assembly

Figure 3.2: These code snippets show the difference between regular assembly and symbolized assembly. Regular assembly contains hard-coded offsets to the beginning of the loop, to the beginning of a function and to a data variable. Symbolized assembly uses labels instead.

the offset between an imported symbol and the instruction that uses it at load time and writes it directly to, e.g., the offset field of a memory access. This is normally not done in userspace because it prevents multiple processes that use the same binary from sharing the code section in memory. This is not an issue for kernel modules because it is not possible to load the same module in memory multiple times. Code relocations are a challenge for the rewriter because they do not specify which field of the instruction they are writing to, but merely state how to compute the value of the relocation and the address that this value should be written to. RetroWrite does not support relocations that write to code because it does not know what part of the instruction's operands should be symbolized or which syntax it should use for the symbolized instruction. We add support for these relocations to `kRetroWrite`'s symbolizer by leveraging additional information provided by the disassembler.

3.3.2 Self-modifying code

Linux uses runtime patching (i.e., self-modifying code) to ensure that it is using an optimal set of machine instructions at runtime. Self-modifying code is a challenge for RetroWrite's rewriter because it treats code as data. RetroWrite lets instrumentation passes insert new instructions anywhere in the symbolized binary because it assumes that the value of the bytes in the code section does not matter as long as the references are preserved. Self-modifying code breaks this and needs special handling, therefore RetroWrite does not support it. We observe that while arbitrary self-modifying code would be intractable for a static rewriter, the kernel supports only

limited transformations. Linux uses a set of tables that specify the address of the instructions to patch and the new code that should be written to that address. We can support runtime self-patching by finding these tables and ensuring that the entries point to the original instructions during symbolization.

3.3.3 Structure of kernel modules

The structure of a kernel module differs from that of a userspace binary. The memory layout of a userspace binary's code and static data is chosen at link time and known statically. This is not true for kernel modules, whose structure is less rigid. The sections of a kernel module can be loaded at an arbitrary distance from each other, which means that the full memory layout of a kernel module is not known until load time. Only the offset of a byte from the start of the section that contains it is known statically. RetroWrite's symbolizer assumes that the layout of the binary is known, and kernel modules break this assumption. We solve this by using (section, offset from the start of the section) as addresses for static data and code.

3.3.4 Multiple code sections

Userspace binaries only use one section for code, one section for writable static data, and one section for read-only static data. Kernel modules can have multiple sections for each. RetroWrite misses code in kernel modules because it expects all code to reside in a section called `.text`. Kernel modules contain several other code sections, such as `.init.text`, which contains initialization routines. We account for this by using section attributes instead of section names to identify code sections.

3.4 Binary kcov instrumentation

We implement a binary instrumentation pass that enables coverage-guided fuzzing of binary Linux modules. Our requirements are to support existing kernel fuzzers and to integrate with existing source-based instrumentation.

RetroWrite reuses AFL's [51] instrumentation pass for coverage tracking. AFL's instrumentation is designed for userspace applications and does not support fuzzing kernels out-of-the box. While other work [35] [38] adapted the instrumentation to the kernel, they are one-off implementations and require a custom fuzzer. One of our design goals is to integrate with existing kernel compiler passes and existing infrastructure. Porting AFL's instrumentation would go against that goal.

kcov [21] is the Linux kernel's coverage collection framework. kcov provides basic block coverage to a fuzzer running in userspace through a special file in debugfs. kcov is tightly integrated into the kernel and it is already used by existing fuzzers such as syzkaller [18]. Therefore it satisfies our requirement of being compatible with existing kernel fuzzers. kcov inserts a call to a tracing function at the beginning of every basic block. Our binary kcov pass uses RetroWrite's *control flow graph* (CFG) analysis pass to recover the start of all basic blocks and instruments recovered blocks. We rely on the main kernel to provide the infrastructure for kcov, such as the tracing function. kcov is not enabled by default for performance reasons but it can be switched on at compile time. Userspace fuzzers can collect coverage for both modules instrumented at compile time and modules instrumented with `kRetroWrite` at the same time. Therefore our binary kcov pass satisfies our other requirement of integrating with existing compiler-based passes.

3.5 Binary KASan instrumentation

The *Kernel Address Sanitizer* (KASan) is a version of ASan that targets the kernel. KAsan relies on compile-time instrumentation and thus does not work on binaries, or inline assembly. We use `kRetroWrite` to implement an instrumentation pass that adds the same checks as KASan to binary Linux modules. To the best of our knowledge we are the first to implement a memory checker that supports binary modules and offers precise detection of out-of-bounds memory accesses.

ASan and KASan consists of a compiler instrumentation pass and a runtime. The compiler pass instruments every memory access with a sanity check and adds red zones around stack variables and globals. The runtime instruments the allocator to insert red zones around heap allocations, manages the shadow memory, and displays error messages when a check fails. The userspace runtime is implemented by `libASan` and instrumented programs link to it. KASan's runtime is part of the kernel itself and is controlled by a compile-time switch. While we can rely on the kernel to provide the runtime, we still need to instrument modules to add checks and insert red zones on the stack.

RetroWrite provides a binary ASan pass for userspace. This binary ASan is about 65% slower than source ASan and achieves the same precision on the heap. KASan's instrumentation is largely similar to ASan's, so we can apply their instrumentation pass to kernel modules with only minor modifications. The main difference between ASan and KASan instrumentation is the location of the shadow metadata table: the ASan shadow is in the low half of the address space while the KASan shadow is in the higher half. We adapt binary ASan to use the KASan shadow.

We also change binary ASan to not spill registers above the stack pointer. Linux's userspace ABI allows saving data above the stack pointer but this is prohibited in the kernel. Binary ASan saves data above the stack pointer which violates the kernel ABI and makes rewritten modules crash. We solve this by making the instrumentation adjust the stack pointer.

Chapter 4

Implementation

In this chapter we cover the implementation of `kRetroWrite`. We cover the implementation of the rewriter, of our binary KASan instrumentation and of our binary kcov instrumentation.

4.1 Symbolizer

Kernel modules use relocations in the code sections to import external symbols. Userspace binaries do not use relocations in the data and code sections directly but instead import all external symbols through the *Global Offset Table* (GOT). This lets all processes that map the same userspace binary share the backing physical memory for all read-only sections because the contents of those sections are always the same. The only section that changes between the different copies is the GOT, which contains pointers to all imported symbols. Kernel modules use relocations in the code section directly because this removes the overhead of accessing the GOT. In order to handle these relocations, our symbolizer must discover which field in the instruction the relocation is writing to and symbolize it accordingly. RetroWrite's symbolizer does not handle relocations that write to the bytes of an instruction because userspace binaries do not use them. We solve this problem by recovering the encoding of the instruction from the disassembler. When `kRetroWrite` loads a binary it disassembles every function, and stores

instruction metadata provided by the disassembler. This includes the encoding of the instruction, which specifies the position of every field in the instruction bytes. Code relocations can write to one of two fields in an x86_64 instruction: a displacement in a memory operand or an immediate. Each instruction can have at most one immediate and one memory operand. We compute the position of the relocation inside the encoded instruction and compare it with the offset of the displacement and the memory operand that we extracted from the disassembly. This way we can disambiguate between a relocation that writes to the immediate and one that writes to the offset of a memory operand.

4.2 Self-modifying code

The kernel subsystem that handles runtime patching is called *SMP Alternatives*. Every kernel module contains a special section called `.altinstructions`, which has references to each instruction to patch and to its replacement. `.altinstructions` entries reference the instruction that they patch using a relocation that points into a code section. Our symbolizer handles runtime patching by ensuring that all such relocations always point to the original instruction and not to instrumentation inserted by a pass. We implement this by inserting labels between the instrumentation and the original instructions and by updating all relocations in `.altinstructions` to point to those labels.

4.3 Binary KASan

Our binary KASan instrumentation pass instruments all memory accesses with a snippet that checks shadow memory and calls KASan's error reporting function if it detects a violation. We reuse most of RetroWrite's binary ASan pass but make some modifications to make it compatible with KASan. The two main differences that we have to account for are that the KASan shadow is located at a different address from the ASan shadow and that the kernel ABI does not use a stack red zone.

4.3.1 Kernel shadow address computation

ASan and KASan use a region of shadow memory to store validity information for memory regions. Each byte of shadow memory corresponds to 8 bytes of real memory. Before a memory access, ASan instrumentation computes the address of the corresponding shadow byte as

$$\text{shadow byte address} = \text{shadow base} + (\text{address} \gg_L 3)$$

where $\gg_L$ is a logical right shift. The instrumentation loads the shadow byte and checks if it indicates that the target is not inside a red zone. If the check fails, the instrumentation calls a reporting function that crashes the program and prints a stack trace. The reporting function takes the address of the faulty memory access as its first argument; the address is used to produce the bug report. The address of the shadow byte is computed in the same way in both ASan and KASan but the base address of the shadow is different. This is because x86_64 splits the address space in two: high memory (above `0xffff800000000000`) and low memory (below `0x7fffffffffff`). Other addresses are not valid and trigger a general protection fault. Linux uses high memory for itself and low memory for userspace. The same rule applies to the ASan and KASan shadows. The authors of ASan chose the base address of the ASan shadow so that it fits into a 32-bit integer. This allows the compiler to encode the base address in a 32-bit immediate, which saves one instruction in the instrumentation. This optimization is not applicable to KASan because the base of the shadow is in high memory, and no high addresses fit into 32 bits. KASan instrumentation has to use an extra instruction to load the shadow base into a register.

RetroWrite's binary ASan pass has to clobber two registers to compute the address of the shadow byte and check its value. It tries to avoid clobbering live registers because spilling and restoring to the stack is expensive. RetroWrite identifies register spills as one of the reasons why the overhead of binary ASan is worse than that of source ASan. Our KASan instrumentation would have to use an extra register to store the shadow base, which would further increase its overhead over compiler instrumentation. We solve this by using a different algorithm to compute the address of a shadow byte. Our algorithm uses two registers, just like RetroWrite's, and keeps

the target address in a register so that we can pass it to the error reporting function without recomputing it. Our instrumentation computes the address of the shadow byte as

$$\text{shadow byte address} = ((\text{shadow base} \ll 3) + \text{address}) \gg_A 3$$

where $\gg_A$ is an arithmetic right shift, $\ll$ is a left shift, and all additions are modulo 2^{64}. It is easy to show that this is equivalent to the ASan algorithm for every address in high memory. We do not need to support low memory addresses because accessing user memory from the kernel directly (i.e., without using `copy_from_user` or `copy_to_user`) is a bug. Accesses to user memory raise a page fault which crashes the kernel and alerts the fuzzer. Source-based KASan also handles accesses to user memory in the same way.

4.3.2 Lack of red zones in the kernel

The System V ABI reserves 128 bytes above the stack pointer as scratch space for the current function. The specification calls this a *red zone*, confusingly the same term that ASan uses for poisoned memory. RetroWrite's binary ASan spill registers to the red zone to avoid moving the stack pointer. However the kernel uses a different ABI without red zones because the CPU uses the space above the stack pointer to push exception frames. If an interrupt fires while the instrumentation is executing, the exception frame overwrites the saved register and makes the kernel crash. The simplest solution is to move the stack pointer above the address where we saved the registers. Unfortunately this fix breaks address calculations for SP-relative memory accesses because the memory access uses the old value of `rsp` as base, but the instrumentation uses the adjusted value. We address this second issue by adjusting the address calculation for SP-relative accesses in our instrumentation when we spill registers. Instead of using `rsp` + orig_offset to look up the shadow byte, we use `rsp` + orig_offset + $2 \times r_s$ where r_s is the number of registers that we spilled.

4.4 Binary kcov

kcov uses a compiler pass to insert a call to a tracing function (`__sanitizer_cov_trace_pc`) at the start of every basic block. The tracing function is implemented in the main kernel and imported by modules. The implementation reads the address of the caller from the stack and does not take any arguments, unlike the ASan error reporting functions. Our binary kcov pass needs to discover basic block starts and instrument them with a call to the tracing function. We use an approximate control flow analysis to discover basic blocks: the analysis pass marks every function start and jump target as basic block start during symbolization. RetroWrite uses the same technique for their fuzzing instrumentation so we reuse their implementation. Like RetroWrite, we only analyze direct jumps and do not rely on heavyweight analysis to infer indirect jump targets.

Binary kcov also saves and restores caller-saved registers when calling the tracing function. The System V ABI specifies that callees are free to overwrite `rax`, `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`, `r10`, `r11`, and `rflags`: calling the tracing function may clobber any of those registers and our instrumentation must preserve them. We use the register analysis pass to infer which of those registers are live at the call site and only spill as many as necessary to limit runtime overhead.

Chapter 5

Evaluation

In this section we validate the claims that we made earlier by performing experiments. We show that `kRetroWrite` combines the scalability of dynamic rewriting with the performance of source-based instrumentation. Furthermore we show that our coverage instrumentation successfully guides a coverage-guided fuzzer deep into its target. We formulate these claims into the following research questions:

RQ1 Does `kRetroWrite` scale to real-world targets?

RQ2 Is our instrumentation competitive to compiler-based memory checking and coverage tracking in terms of performance?

RQ3 Is our binary coverage instrumentation competitive with source-based coverage instrumentation on the above targets? Is it as effective as source-based instrumentation in guiding fuzzers?

We answer our research questions by performing the following experiments:

1. Rewrite Btrfs [37], Ext4 [13], and `isofs` [26], three real-world kernel modules.

2. Evaluate fuzzing throughput on the above modules, comparing (i) compiler-based KASan and kcov and (ii) binary KASan and kcov.

3. Evaluate fuzzing coverage on the above targets, comparing the basic block coverage reached when fuzzing with (i) source-based kcov and (ii) compiler-based kcov.

5.1 Setup and hardware

We run our fuzzing campaigns with syzkaller [18] compiled from git commit `3a75be00f50996031dd301d44b009d56db3485f0` and Linux `5.5-rc1`. syzkaller is a state-of-the-art coverage-guided kernel fuzzer that found thousands of bugs in Linux and other mainstream OSs [45]. syzkaller collects coverage with kcov and uses KASan to detect memory errors, and so it is compatible with our binary instrumentation. We build Linux with `defconfig` + `kvmconfig`, which are needed to run inside a VM, and enable debug info, kcov, and KASan, which are required to fuzz with Syzkaller. We build Ext4, Btrfs, `isofs`, and their dependencies as modules and compile everything else into the main kernel image. We build the target modules twice: once with source-based instrumentation and once with no instrumentation. We rely on the KASan and kcov runtimes built into the main kernel image rather than adding a copy of the runtime to the instrumented modules. This allows our instrumentation to interoperate with source-instrumented modules and reduces complexity (see section 3.2). We use an *initial ramdisk* (initramfs) to boot the machine and load the modules, then hand control over to a Debian Stretch userspace image.

There is no way to target syzkaller to a single module: by default syzkaller fuzzes every system call that it knows about. To target our fuzzing campaign we whitelist only system calls that interact with the file system, such as `open`, `read`, `write`, `ioctl`. This is not a perfect solution because those same system calls can also be used to interact with drivers through, e.g., character devices, but it restricts the scope of the campaign by disallowing large subsystems such as network I/O. We build two such whitelists: the first one includes all system calls that interact

with the file system from userspace, and the second one includes only a small set of seldom-used system calls. We run the target kernel in QEMU VMs with 2 CPUs and 2 GiB of physical memory each. Each VM runs 8 userspace agents concurrently. We store the VMs' disks in `tmpfs` to reduce the overhead of disk writes and minimize storage wear. We use QEMU in KVM mode to minimize the overhead of virtualization.

As we have a limited amount of time to conduct our evaluation, we run our fuzzing campaigns on two machines in parallel: the first machine has an AMD Ryzen 7 PRO 1700 (8 cores, 16 threads) CPU and 32 GiB of DRAM and runs Ubuntu 19.04; the second one has an Intel Core i7-8700 (6 cores, 12 threads) CPU and 16 GiB of DRAM and runs Ubuntu 19.10. We use the first machine to fuzz with the full system call list and 8 VMs, and the second machine to fuzz with the reduced system call list and 6 VMs. We build Linux using `gcc` 9.2.1.

We begin each 24h fuzzing trial with an empty corpus and save the corpus at the end of the run. Syzkaller uses API descriptions for system call inputs and therefore it is able to discover interesting paths quickly even with an empty corpus.

5.2 Scalability

We use `kRetroWrite` to rewrite and instrument the kernel modules shown in Table 5.1. They are part of mainline Linux and each implements a file system. Ext4 is the default file system for many major Linux distributions such as Debian [10] and Fedora Workstation [14]. Btrfs is also supported by major distros and is the default file system for openSUSE and SLES [6]. `isofs` implements the ISO 9660 file system, the standard file systems for CD-ROMs [20].

Name	Size (debug)	Size (stripped)	Symbolization (avg)	Instrumentation (avg)
Btrfs	32M	1.2M	48s	+14s
Ext4	16M	616K	21s	+10s
`isofs`	1.8M	32K	4.3s	+0.6s

Table 5.1: The Linux modules that we instrument with `kRetroWrite` and fuzz. We instrument the modules with both binary kcov and binary KASan and report `kRetroWrite`'s run time.

We symbolize each module in the table using `kRetroWrite` and instrument it with binary KASan and binary kcov. We run `kRetroWrite` on an Intel Core i7-8700 CPU @ 3.20GHz with PyPy3 7.1.1. The modules are built from Linux `5.5-rc1`, the latest development version at the time of our evaluation. `kRetroWrite` is fast enough that it could be run automatically as part of a fuzzing campaign, e.g., when a vendor publishes an updated version of a binary module. Symbolization and instrumentation are embarrassingly parallel tasks because each function can be processed independently, but `kRetroWrite` runs on a single thread. Implementing parallel symbolization and instrumentation would reduce the run time further and would only require a minor engineering effort.

As Table 5.1 shows, we successfully rewrote and instrumented large, real world kernel modules. Dinesh et al. evaluate RetroWrite by rewriting binaries with a maximum size of 12MB, and conclude that RetroWrite scales to real world binaries. The largest kernel module in our evaluation is 32M (*2.6x* larger) and all our modules are larger than one megabyte. Therefore we are confident that `kRetroWrite` scales to arbitrarily large kernel modules (**RQ1**).

5.3 Instrumentation Performance

We evaluate the performance of our instrumentation by fuzzing Ext4, Btrfs, and `isofs` with syzkaller for 24 hours and measuring the number of executions per second. The number of executions per second is an important metric for fuzzers because covering more of the target directly translates to a higher probability of finding bugs. We also compare coverage in section 5.4 because a fuzzer could easily maximize its throughput by only exploring shallow paths in the target. Such a fuzzer would achieve high throughput but not find any bugs.

5.3.1 Results

Our evaluation shows that our performance is similar to source-based kcov and KASan (**RQ2**). Figure 5.1 shows syzkaller's throughput in the fuzzing trials that we ran. We are 8.3% slower

on average on Btrfs, 2x faster on `isofs` and 26% slower on Ext4 with the full system call list. With the reduced system call list, we are 1% faster on Btrfs and 72% faster on Ext4. We were not able to run more trials due to time constraints, so our results are affected by random variance. Unlike RetroWrite's binary ASan, which consistently showed a slowdown, in many cases our instrumentation does not seem to have a large performance impact compare to source-based instrumentation. We explain this by observing that RetroWrite evaluated the performance of binary ASan on SPEC CPU2006. SPEC CPU2006 is a CPU benchmark which involves few context switches and all the code was instrumented with binary ASan or ASan in their evaluation. We only use our instrumentation on the target module and always use compiler-based instrumentation on the rest of the kernel. Furthermore fuzzing the kernel incurs many context switches as each test case issues multiple system calls, which cause two context switches each. Disk I/O causes further context switches because every disk access is handled by the hypervisor. Finally the fuzzer processes have to communicate with the manager process which runs on the host. We omit the results for `isofs` with the reduced system call list because syzkaller did not generate any test cases covering the module in that setting. We speculate that this happens because we cannot use a CD-ROM image as the root filesystem for fuzzing and therefore file operations do not hit any `isofs` filesystems.

We do not modify syzkaller, the kernel, the compiler, the hypervisor, or any of the modules to run any of our experiments. This shows that `kRetroWrite` integrates seamlessly into an off-the-shelf kernel fuzzing setup.

5.4 Instrumentation Coverage

We evaluate the effectiveness of our coverage tracking instrumentation by measuring the coverage of the target module when fuzzing with source kcov and binary kcov. We save the fuzzing corpus after each campaign and re-run all test cases on a kernel instrumented with source kcov. We compute the number of basic blocks in the module and use it to compute the fraction of basic blocks covered by each corpus.

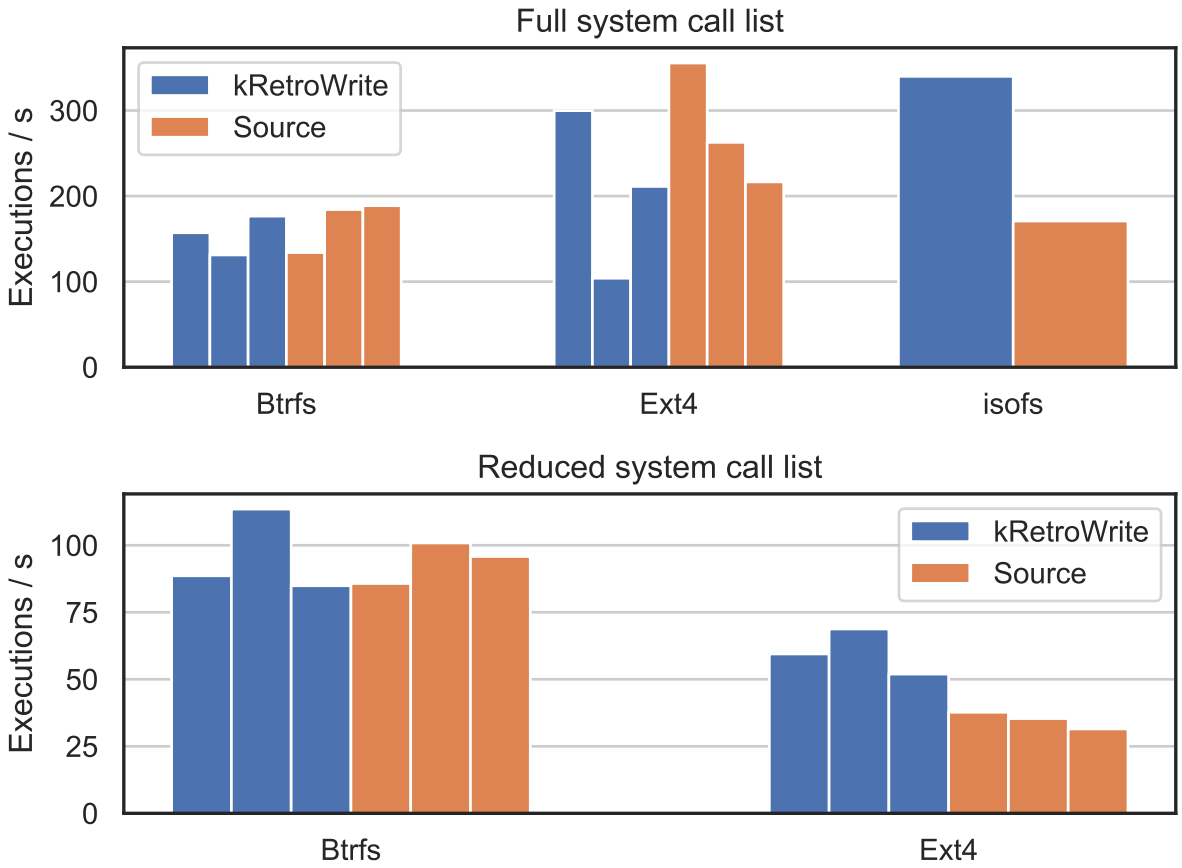

Figure 5.1: Plot of fuzzing executions per second on real-world kernel modules. Each fuzzing trial ran for 24 hours. We repeated the trials 3 times for Btrfs and Ext4, and only once for `isofs` due to time constraints. Higher numbers indicate better performance.

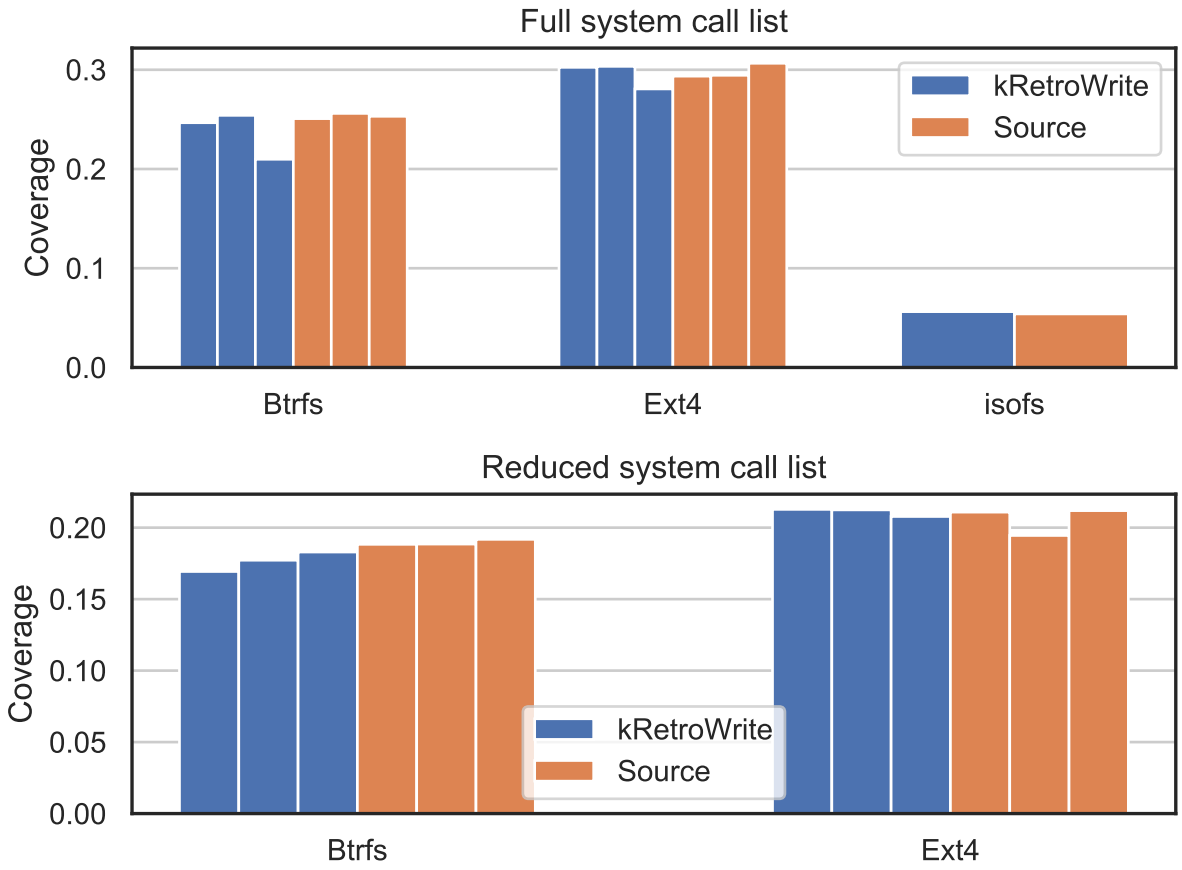

Figure 5.2: Plot of basic block coverage for our fuzzing campaigns. Coverage is given as a fraction of the total number of basic blocks in the module under test. We repeated the trials 3 times for Btrfs and Ext4, and only once for `isofs` due to time constraints. Higher numbers indicate better performance.

Our experiments show that the coverage obtained by syzkaller with our binary instrumentation is similar to the one obtained with source-based instrumentation (**RQ3**). Figure 5.2 shows that syzkaller covers about 6% less basic blocks when using our instrumentation on Btrfs, whereas we are within 2% of the source-based instrumentation on Ext4 and `isofs`. We hypothesize that the missing coverage is due to the imprecise control flow recovery missing basic blocks. Our evaluation therefore shows that our instrumentation is a viable alternative to source-based coverage instrumentation for fuzzing binary kernel modules.

We also show that when fuzzing with our binary instrumentation, syzkaller hits many of the basic blocks it hit with source-based instrumentation with similar frequency. We rerun all test cases from the Ext4 corpora (full system call list) on a kernel instrumented with source kcov, and count how many times the corpora hit each basic block in Ext4. Figure 5.3 shows that most basic blocks in Ext4 are covered by a similar number of test cases in the source corpora and in the binary corpora. Moreover it shows that the basic blocks that syzkaller found with our binary kcov are not a subset of the blocks that it found with source-based kcov. This means that our binary kcov did not only guide syzkaller through the module effectively but it also made it discover modules that weren't found by the source-based instrumentation.

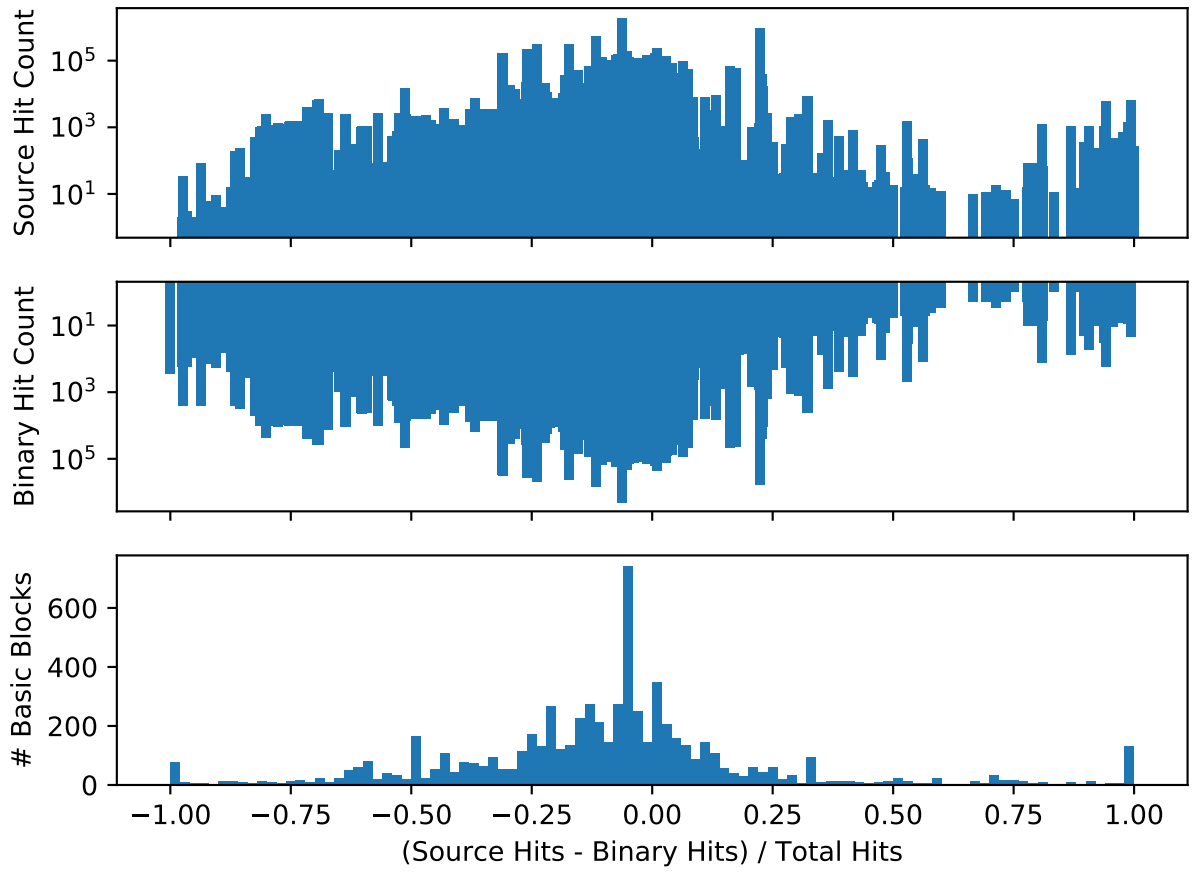

Figure 5.3: This plot shows how many times the corpora accumulated by syzkaller when fuzzing Ext4 with the full system call list covered each basic block in Ext4. The basic blocks are sorted on the horizontal axis according to the relative difference in popularity. The bottom plot is a histogram of the distribution of the basic blocks. The top two plots have a log-scale vertical axis.

Chapter 6

Related Work

6.1 Blackbox fuzzing

Blackbox fuzzers are the oldest and simplest class of fuzzers. They are not aware of the structure of the target at all but instead they entirely rely on random mutation to trigger crashes in the target. The fundamental problem for black box fuzzer is to find interesting paths through the target that lead to bugs. Because most real-world targets expect structured input, purely random test cases are unlikely to be effective. Prior work attempted to address this in various ways. One approach is to use random mutation on a starting corpus with no coverage feedback (e.g., Radamsa [36]). This class of fuzzers relies on a high-quality starting corpus and cannot discover the input format on its own. Another approach is to use grammars or API descriptions to generate valid inputs or call sequences. Trinity [47] is an example of a blackbox Linux system call fuzzer that uses system call descriptions. This approach is more effective than executing random call sequences but requires manual effort to create API descriptions. IMF [19] addresses this problem by inferring API models for the kernel automatically. IMF infers API models (sets of ordering dependences and value dependences) from the execution trace of a userspace program and uses it to fuzz the kernel. IMF can fuzz binaries but relies on crashes to detect bugs, and is likely that it will miss bugs such as subtle memory corruption. Furthermore it increases coverage by

observing userspace consumers of kernel APIs and not by observing the kernel itself, unlike a coverage guided fuzzer. DIFUZE [8] infers system call models automatically like IMF but does so by analyzing source code. Their approach is not applicable to binaries.

6.2 Symbolic execution and whitebox fuzzing

Symbolic execution is an alternative to fuzzing for finding bugs. Symbolic execution consists in executing the program under test with a symbolic, rather than concrete, input and collecting path constraints as the execution proceeds. Unlike concrete execution, which only takes one of the possible paths, symbolic execution explores all paths. When the symbolic executor encounters an error condition it can use a constraint solver to synthesize an input that triggers the error from the path constraints. S^2E [7] can symbolically execute binaries and does not require source code. It can also symbolically execute an entire system, including the kernel. Symbolic execution suffers from similar problems to fuzzing, such as low coverage, but for different reasons. Fuzzing struggles to achieve high coverage of the target because random inputs stop exploring new paths through the target. Symbolic execution suffers from exponential path explosion in all but the simplest targets, and complex constraints strain the solver. CAB-Fuzz [22] limits path explosion when fuzzing the kernel by focusing symbolic execution on states that are more likely to lead to vulnerabilities, such as boundary conditions. Tools like Driller [44] combine fuzzing and symbolic execution to mitigate the weaknesses of both techniques. Driller overcomes the path explosion problem by using the fuzzer to guide exploration of the target and only switching to symbolic execution when the fuzzer gets stuck trying to bypass a complex check.

Whitebox fuzzers can find deeper bugs in their target but are often significantly harder to implement and scale to large systems. Scaling whitebox testing to full kernels is an open area of research.

6.3 Hardware-assisted greybox fuzzing

kAFL [38] uses *Intel Processor Trace* (IPT) to collect coverage information about kernel code at low overhead. IPT is a feature of recent Intel CPUs that records the execution flow of code executing on the CPU, including the outcome of conditional jumps and the target of direct jumps. kAFL uses customized versions of QEMU and KVM that use IPT to record the execution trace of a guest operating system running in a VM. Their approach relies solely on hardware support and does not use any instrumentation, and therefore it has almost no performance overhead. kAFL is mostly OS-independent because it uses the hypervisor to collect coverage information and only contains a small amount (usually less than 150 lines) of OS-specific code.

Unlike kAFL, our approach uses static instrumentation which is more general because it can be used not only to collect coverage but also to implement sanitization. Furthermore kAFL relies on hardware support and cannot be ported to other architectures which lack IPT, or to older Intel CPUs. kAFL does not integrate with existing coverage tracking frameworks such as kcov and therefore it cannot be integrated with off-the-shelf fuzzers like syzkaller without additional engineering effort.

6.4 Rewriting-based kernel instrumentation

DBT-based fuzzers use a dynamic rewriter to instrument the target kernel at runtime. The target kernel is virtualized using dynamic binary translation and the DBT hypervisor instruments the target's code on the fly. An example of such a fuzzer is TriforceAFL [35] which uses QEMU in system emulation mode to collect coverage information about the guest OS. TriforceAFL can fuzz any target that runs inside QEMU and can run entirely in a RAM disk, and therefore porting it to new target only requires little engineering effort. TriforceAFL uses heavyweight dynamic binary instrumentation which slows down execution and prevents the fuzzer from finding interesting test cases. Schumilo et al. found that TriforceAFL is up to 54 times slower than kAFL [38]. Unicorefuzz [29] uses DBT to target arbitrary kernel functions that are not accessible

from userspace. It uses Unicorn [32], a fork of QEMU, to target parsers in the kernel. Unicorefuzz can expose bugs in parts of the kernel that are normally difficult to reach from userspace and works on binaries. However the authors' evaluation shows that Unicorefuzz achieves around 46% of the throughput of AFL-QEMU [1]. Valgrind memcheck implements memory sanitization for userspace programs using the Valgrind [30] DBT toolkit. Dinesh et al. [12] show that Valgrind memcheck is 3x slower than RetroWrite's Binary ASan and detects fewer bugs. To the best of our knowledge there exist no DBT-based systems that can perform memory sanitization for kernel code. Despite the name, the Linux kernel's kmemcheck is unrelated to Valgrind memcheck and does not use DBT.

DBT-based approaches have good scalability and are widely used to fuzz real-world targets, but they impose high overhead which precludes their use for high-performance instrumentation.

RetroWrite [12] rewrites userspace binaries at low cost without heuristics and achieves both scalability and low overhead, but does not work on the kernel. Static rewriters before RetroWrite [50] [49] [33] [4] [11] [42] [2] were limited to small binaries, had other limitations such as high memory runtime overhead, or could only apply some types of instrumentation.

Chapter 7

Conclusion

7.1 Future work

Support for other operating systems Our prototype implementation of `kRetroWrite` can only rewrite Linux modules. Although closed source modules for Linux do exist, they are far more common in proprietary operating systems such as Windows and macOS. `kRetroWrite` makes no assumption about the target operating system except for the loader, which must support the executable format used on the target platform. Porting `kRetroWrite` to those platforms is an engineering effort which we leave to future work.

Support for other architectures `kRetroWrite` only supports x86_64. While x86_64 is the most popular architecture on desktops and servers, it only has a small market share in other fields such as mobile devices and IoT. Supporting other architectures would require extending the symbolizer to handle more relocation types, and porting the assembly snippets which we use in our instrumentation passes. ARM is the most popular architecture for Android phones, which are based on Linux and often ship with proprietary device drivers. Porting `kRetroWrite` to ARM would allow us to fuzz closed-source modules that ship with Android phones.

Mitigations Our instrumentation passes target fuzzing but it is also possible to use `kRetroWrite`

to retrofit security mitigations. For example it could be used to add a shadow stack by instrumenting all call and return instructions. We hypothesize that Hardware-assisted mitigations, such as Intel CET [9] and ARM pointer authentication [24] can be retrofitted on binaries without heavyweight static analysis.

7.2 Conclusion

To summarize we develop `kRetroWrite`, a scalable static rewriter for Linux modules. We present two instrumentation passes implemented on top of `kRetroWrite`: (i) binary kcov to collect coverage for greybox fuzzing, and (ii) KASan memory checking to precisely detect memory corruption during fuzzing. We show that our instrumentation passes are competitive with source-based instrumentation in both performance and coverage, and at the same time remain compatible. `kRetroWrite` instruments binary kernel modules, enabling targeted application of our instrumentation even where compiler instrumentation is not an option, such as proprietary modules, inline assembly, and code that requires an old compiler. Our work shows that RetroWrite's approach is not only applicable to userspace but also to the kernel.

Bibliography

[1] *AFL QEMU mode*. [Online, accessed 4-Jan-2020]. URL: https://github.com/google/AFL/blob/master/qemu_mode/README.qemu.

[2] Erick Bauman, Zhiqiang Lin, and Kevin W Hamlen. "Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics." In: *Network and Distributed System Security Symposium*. 2018.

[3] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *Usenix Annual Technical Conference*. 2005.

[4] Andrew R. Bernat and Barton P. Miller. "Anywhere, Any-Time Binary Instrumentation". In: *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*. 2011.

[5] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. "An Infrastructure for Adaptive Dynamic Optimization". In: *International Symposium on Code Generation and Optimization*. 2003.

[6] *Butter bei die Fische!* [Online, accessed 8-Jan-2020]. URL: https://www.suse.com/c/butter-bei-die-fische/.

[7] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "S2E: A Platform for In-vivo Multi-path Analysis of Software Systems". In: *International Conference on Architectural Support for Programming Languages and Operating Systems*. 2011.

[8] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. “DIFUZE: Interface aware fuzzing for kernel drivers”. In: *ACM Conference on Computer and Communication Security*. 2017.

[9] Intel Corporation. *Control-flow Enforcement Technology Specification*. [Online, accessed 9-Jan-2020]. URL: https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[10] *Debian - Ext4*. [Online, accessed 8-Jan-2020]. URL: https://wiki.debian.org/Ext4.

[11] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. “Bistro: Binary component extraction and embedding for software security applications”. In: *European Symposium on Research in Computer Security*. 2013.

[12] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. “RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization”. In: *IEEE International Symposium on Security and Privacy*. 2020.

[13] *Ext4*. [Online, accessed 8-Jan-2020]. URL: https://www.kernel.org/doc/Documentation/filesystems/ext4.txt.

[14] *Ext4 in Fedora 11*. [Online, accessed 8-Jan-2020]. URL: https://fedoraproject.org/wiki/Ext4_in_Fedora_11.

[15] *Fedora - Harden All Packages*. [Online, accessed 7-Jan-2020]. URL: https://fedoraproject.org/wiki/Changes/Harden_All_Packages.

[16] James Forshaw. *Breaking the Chain*. [Online; accessed 25-December-2019]. URL: https://googleprojectzero.blogspot.com/2016/11/breaking-chain.html.

[17] *Getting started with kmemcheck*. [Online, accessed 3-Jan-2020]. URL: https://www.kernel.org/doc/html/v4.14/dev-tools/kmemcheck.html.

[18] Google. *syzkaller*. [Online, accessed 5-Jan-2020]. URL: https://github.com/google/syzkaller.

[19] HyungSeok Han and Sang Kil Cha. “IMF: Inferred model-based fuzzer”. In: *ACM Conference on Computer and Communication Security*. 2017.

[20] *Information processing — Volume and file structure of CD-ROM for information interchange*. Standard. International Organization for Standardization, Apr. 1988.

[21] *kcov: code coverage for fuzzing*. [Online, accessed 4-Jan-2020]. URL: https://www.kernel.org/doc/html/v5.5-rc1/dev-tools/kcov.html.

[22] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, and Taesoo Kim. "CAB-Fuzz: Practical Concolic Testing Techniques for {COTS} Operating Systems". In: *Usenix Annual Technical Conference*. 2017.

[23] Andrey Konovalov and Dmitry Vyukov. *KernelAddressSanitizer (KASan) — a fast memory error detector for the Linux kernel*. [Online, accessed 6-Jan-2020]. URL: https://events.static.linuxfound.org/sites/events/files/slides/LinuxCon%20North%20America%202015%20KernelAddressSanitizer.pdf.

[24] Arm Limited. *Arm® Architecture Reference Manual - Armv8, for Armv8-A architecture profile*. [Online, accessed 9-Jan-2020]. URL: https://static.docs.arm.com/ddi0487/ea/DDI0487E_a_armv8_arm.pdf.

[25] *Linux Kernel CVEs*. [Online; accessed 25-December-2019]. URL: https://www.linuxkernelcves.com/cves.

[26] *Linux Kernel Documentation - isofs*. [Online, accessed 8-Jan-2020]. URL: https://www.kernel.org/doc/Documentation/filesystems/isofs.txt.

[27] *LKML archive: kmemcheck: kill kmemcheck*. [Online, accessed 3-Jan-2020]. URL: %7Bhttps://lore.kernel.org/lkml/20171007030159.22241-1-alexander.levin@verizon.com/%7D.

[28] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: *ACM International Conference on Programming Language Design and Implementation*. 2005.

[29] Dominik Maier, Benedikt Radtke, and Bastian Harren. "Unicorefuzz: On the Viability of Emulation for Kernelspace Fuzzing". In: *Usenix Workshop on Offensive Technologies*. 2019.

[30] Nicholas Nethercote and Julian Seward. “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation”. In: *ACM International Conference on Programming Language Design and Implementation*. 2007.

[31] *New 17.0 profiles in the Gentoo repository*. [Online, accessed 7-Jan-2020]. URL: https://www.gentoo.org/support/news-items/2017-11-30-new-17-profiles.html.

[32] Anh Quynh Nguyen and Hoang Vu Dang. *Unicorn: Next Generation CPU Emulator Framework*. [Online, accessed 9-Jan-2020]. URL: https://www.unicorn-engine.org/BHUSA2015-unicorn.pdf.

[33] Pádraig O’sullivan, Kapil Anand, Aparna Kotha, Matthew Smithson, Rajeev Barua, and Angelos D Keromytis. “Retrofitting security in cots software with binary rewriting”. In: *IFIP International Information Security Conference*. Springer. 2011, pp. 154–172.

[34] *OSS-Fuzz issue tracker*. [Online, accessed 4-Jan-2020]. URL: https://bugs.chromium.org/p/oss-fuzz/issues/list?can=1&q=Type%3DBug-Security&colspec=ID%20Status%20Proj%20Reported%20Summary&sort=-id&num=1000.

[35] *Project Triforce: Run AFL on Everything!* [Online; accessed 19-December-2019]. 2016. URL: https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/.

[36] *Radamsa — a general purpose fuzzer*. [Online, accessed 9-Jan-2020]. URL: https://gitlab.com/akihe/radamsa.

[37] Ohad Rodeh, Josef Bacik, and Chris Mason. “Btrfs: The linux b-tree filesystem”. In: *ACM Transactions on Storage (TOS)* (2013).

[38] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. “kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels”. In: *Usenix Security Symposium*. 2017.

[39] *Security Enhancements in Android 5.0*. [Online, accessed 7-Jan-2020]. URL: https://source.android.com/security/enhancements/enhancements50.html.

[40] Konstantin Serebryany. *Hardware Memory Tagging to make C/C++ memory safe(r)*. [Online, accessed 4-Jan-2020]. URL: https://github.com/google/sanitizers/blob/master/hwaddress-sanitizer/Hardware%20Memory%20Tagging%20to%20make%20C_C++%20memory%20safe(r)%20-%20iSecCon%202018.pdf.

[41] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. "AddressSanitizer: A fast address sanity checker". In: *Usenix Annual Technical Conference*. 2012.

[42] Matthew Smithson, Khaled ElWazeer, Kapil Anand, Aparna Kotha, and Rajeev Barua. "Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness". In: *2013 20th Working Conference on Reverse Engineering (WCRE)*. 2013.

[43] StatCounter. *Desktop Operating System Market Share Worldwide*. [Online, accessed 3-Jan-2020]. URL: https://gs.statcounter.com/os-market-share/desktop/worldwide.

[44] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." In: *Network and Distributed System Security Symposium*. 2016.

[45] *Syzbot fixed bugs*. [Online, accessed 8-Jan-2020]. URL: https://syzkaller.appspot.com/upstream/fixed.

[46] *The Kernel Address Sanitizer (KASAN)*. [Online, accessed 3-Jan-2020]. URL: https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[47] *Trinity*. [Online; accessed 25-December-2019]. URL: https://github.com/kernelslacker/trinity.

[48] *Ubuntu Foundations Team - Weekly Newsletter, 21017-06-15*. [Online, accessed 7-Jan-2020]. URL: https://lists.ubuntu.com/archives/ubuntu-devel/2017-June/039816.html.

[49] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. “Ramblr: Making Reassembly Great Again.” In: *Network and Distributed System Security Symposium*. 2017.

[50] Shuai Wang, Pei Wang, and Dinghao Wu. “Reassembleable disassembling”. In: *Usenix Security Symposium*. 2015.

[51] Michał Zalewski. *American Fuzzy Lop*. [Online, accessed 4-Jan-2020]. URL: `http://lcamtuf.coredump.cx/afl/`.

[52] Michał Zalewski. *The bug-o-rama trophy case*. [Online, accessed 7-Jan-2020]. URL: `http://lcamtuf.coredump.cx/afl/#bugs`.