Unicorefuzz:
On the Viability of Emulation for Kernelspace Fuzzing

Dominik Maier
TU Berlin

Benedikt Radtke
TU Berlin

Bastian Harren
TU Berlin

Abstract

Fuzzing uncovers an ever-growing number of critical vulnerabilities. Despite the simple concept — execute the target until it crashes — setting up fuzz tests can pose complex challenges. This is especially true for code that cannot run as part of a userland process on desktop operating systems — for example device drivers and kernel components. In this paper, we explore the use of CPU emulation to fuzz arbitrary parsers in kernelspace with coverage-based feedback. We propose and open-source *Unicorefuzz* and explain merits and pitfalls of emulation-based fuzzing approaches. The viability of the approach is evaluated against artificial Linux kernel modules, the Open vSwitch network virtualization component as well as bugs originally uncovered by syzkaller. Emulator-based fuzzing of kernel code is not very complex to set up and can even be used to fuzz operating systems and devices for which no source code is available.

1 Introduction

Fuzz testing is a powerful way to uncover a variety of bugs in an automated fashion. Companies start building up fuzzing pipelines like OSS-Fuzz, proposed by Serebryany [36]. Fuzzing user space software has been done for several decades [18], however fuzzing operating systems and kernels poses a more difficult task [34]. If arbitrary kernel components in an emulator could be fuzzed directly, security researchers would be able to scrutinize any part of any kernel and thus gain a significant edge in the everlasting battle against security vulnerabilities. A few years ago, fuzzing kernel code, for example Wi-Fi drivers, required complex setups. They had to resemble real-world scenarios, like dedicated rogue access points [6] that are hard to integrate into feedback-based fuzzing methodologies. Fuzzing kernelspace code is hard since any state change affects the whole system, the code is tucked away from userland and recovering from crashes is hardly possible without an additional layer of abstraction. The current state of the art of kernel fuzzing, *syzkaller*, solves this problem by fuzzing kernel inside Virtual Machines, triggering code from the VM-internal userland via syscalls and passing feedback back to the userland [17]. This, however, means syscalls for all tests need to exist or need to be written. It also makes security testing of binary-only kernel modules, as well as odd operating systems, hard to impossible. In this paper, we take a different approach to the problem domain. We propose *Unicorefuzz*, a novel way to fuzz parsers in the kernelspace, based on AFL-Unicorn, a CPU emulator-based fuzzer. The Unicorn emulator supports a vast range of processor architectures [31], which makes fuzzing of arbitrary kernel code, even for embedded architectures, viable. In this paper, we show that fuzzing arbitrary kernelspace functions is possible and viable.

1.1 Basic terminology

First, we define basic terms we will use throughout this paper.

Fuzzer A fuzzer successively generates inputs to the fuzz target while observing its behavior. It collects inputs that trigger behavior of interest for further analysis.

Target The fuzz target is the item being tested using a fuzzer. Depending on the methodology used this can be anything ranging from an interpreted script to a hardware device. The main requirements are that it accepts input and exhibits some resulting observable behavior.

Coverage A contiguous block of steps or instructions that are executed unconditionally is known as a *basic block*, while *branches* are conditional transfers between basic blocks. Depending on the inputs, different paths (i.e. chains of basic blocks) may be triggered. The amount of branches taken in a target affects the coverage — with more being a higher number. Normalized, coverage indicates the percentage of all possible branches taken in a single execution.

1.2 Feedback Fuzzing

Feedback fuzzing instruments the fuzz target in such a way that it receives immediate feedback about the effects of an input. According to this feedback, the fuzzer will change test cases in future iterations. Coverage guided fuzzing is the most famous example. It uses code coverage or, closely related, the number of distinct basic blocks transitions triggered, as a feedback for the fuzzer's genetic algorithm [26]. The fuzzer prefers test cases yielding higher coverage and different code paths. The goal is the maximization of code coverage in the target. Employing code coverage as a fitness function for a genetic algorithm then allows the fuzzer to iteratively build a model of the program's input grammar, which in turn improves the inputs given to the tested application. The idea to use instrumentation to learn test inputs through genetic algorithms was already proposed in 1995 by Stahmer [39]. In 2006, Jared DeMott took this approach again and proposed an evolving fuzzer [11, 12]. Simultaneously Drwery and Ormandy proposed and patented a similar method [15, 16]. The prominent fuzzers today, AFL and libFuzzer, both favour inputs that cause code paths previously not observed to be executed [27, 43]. Even with coverage guided fuzzing, using a valid input file is still recommended and expected to yield a higher code coverage early on [37, 43].

1.3 Contribution

With *Unicorefuzz*, we propose a novel approach to kernel fuzzing. We discuss the possibilities to fuzz any parser through partial emulation. The approach yields good results in artificial test cases and successfully reproduced bugs previously found by syzkaller. As a case study, it fuzzed a parser in the kernel code of *Open vSwitch*. The current setup was used to fuzz statically linked kernel modules as well loadable kernel objects on Linux. The implementations created during this research will be open sourced.

2 Related Work

Fuzzing kernel components is not as widespread as fuzzing user space programs, but some notable projects have successfully fuzzed kernel subsystems, drivers and APIs. We will not go into greater detail on publications that fuzz device drivers, such as Periscope by Song et al. [38] who focus on the hardware-OS-boundary. Instead, we discuss notable general-purpose kernel fuzzers in the following.

2.1 Trinity

While fuzzing syscalls with purely random values is simple to set up and can find some bugs, most of the tests fail immediately because of completely incorrect parameters: EINVAL (invalid argument). To reduce the percentage of wasted tests, the kernel fuzzer Trinity uses rules to execute syscalls with better-than-random input. Users need to supply annotation files for every syscall, so-called *madvise* files, including the number of arguments and argument kinds. The creator of Trinity found more than 150 bugs inside system call code, networking stack, virtual memory management and device drivers [24] in 2012.

2.2 DIFUZE

Corina et al. propose DIFUZE, an interface aware fuzzer for kernel drivers. [9]. In contrast to Trinity, DIFUZE does not need descriptions of interfaces, rather, they are inferred when building the kernel from source. DIFUZE does not offer Instrumentation options.

2.3 syzkaller

Just like Trinity, syzkaller relies on templates to reduce the percentage of wasted tests, but uses coverage based feedback-fuzzing to increase the code coverage of fuzz tests [17, 42]. The coverage data is tracked on a per-task basis and extracted from the target via a special *debugfs* entry in `/sys/kernel/debug/kcov` that outputs the current instrumentation information [42]. This kernel patch has been merged into the mainline Linux kernel by now [23]. During the first few months alone, syzkaller found more than 150 bugs in the then recent Linux kernel.

2.4 In-Kernel AFL

Nossum and Casasnovas have ported AFL to kernel space to fuzz test filesystem implementations. Like syzkaller, their unnamed tool uses coverage data supported by GCC. They expose a shared memory region to be accessible from the user space in `/dev/afl` and mmap it into AFL's memory. This way the section exists only once in physical memory. AFL can access it in the same way as it does when testing user space binaries. Since the overhead of virtualization with KVM VMs was high, they switched to User-Mode Linux which yielded a speedup of 60x. Bugs were quickly discovered in the implementations of many file system drivers [33].

2.5 TriforceAFL

TriforceAFL fuzzes virtual machines by extending AFL's QEMU mode. The host runs AFL and QEMU, the guest runs a so-called driver, a userland program that runs the target. AFL sends its generated input to the driver and QEMU sends an edge trace back to AFL [21]. The driver communicates with the TriforceAFL setup through a dedicated hypercall in QEMU. The fork system call only moves the calling thread's state into the new process and may leave threading objects like locks in an undefined state. Since QEMU uses 3 threads

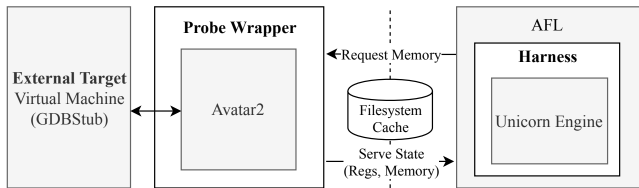

Figure 1: Architecture of Unicorefuzz

for full system emulation, TriforceAFL explicitly saves and restores the required VM state. TriforceAFL has found multiple bugs in Linux and OpenBSD, the most severe ones on Linux were named CVE-2016-4998 (out-of-bounds read of kernel memory which may lead to information disclosure and heap corruption) and CVE-2016-4997 (decrement of arbitrary kernel integers if they are positive) [21].

2.6 kAFL

The fuzzer *kAFL* leverages Intel's *Processor Trace* to recover hardware assisted code coverage. It supports OS-independent, coverage-guided kernel fuzzing and has found several bugs in Linux, macOS and Windows drivers. To provide a communication channel between the VM and kAFL, modified versions of QEMU (QEMU-PT) and KVM (KVM-PT) provide new hypercalls. After the VM has started, the agent transmits the address of the panic handler (or BugCheck on Windows) to QEMU-PT, which replaces the code at that address with a HC_SUBMIT_PANEL hypercall. If the target panics, QEMU-PT is thus notified immediately. Afterwards the agent fetches the actual user mode agent (HC_GET_PROGRAM) and starts it, transmits the value of the CR3 register (HC_SUBMIT_CR3) and declares where it expects its input (HC_SUBMIT_BUFFER). Now the main loop is ready to run. The PT data is decoded and converted to AFL's bitmaps and fed into kAFL to evaluate a test's taken branches. On every benchmarked platform, kAFL delivers more than 20 times the executions per second of TriforceAFL [34].

3 *Unicorefuzz*

This section discusses the building blocks for *Unicorefuzz* as well as the implementation itself.

Emulator Based Instrumentation For feedback fuzzing of kernel code, Unicorefuzz builds on top of *AFL-Unicorn*, which, in turn, uses a patched version of Unicorn, a fork of QEMU. As discussed in 1.2, modern fuzzers use feedback, usually code coverage or basic blocks reached, for test-case generation. Since advanced emulators, for example *QEMU*, translate a program dynamically, basic block by basic block [3], instrumentation can be added to the emulator or the translated code with ease. By instrumenting targets this way, they can be tested through feedback-fuzzing without the need to compile them from source.

Unicorn is a CPU emulator which dynamically translates CPU instructions from any supported binary instruction set into the host instruction set. In particular, given a binary for a CPU platform, QEMU, or Unicorn, for that matter, works by performing the following steps during runtime:

1. Translate a basic code block from the target platform's instruction set to the host platform's instruction set.
2. Cache the translated blocks.
3. Store a mapping from source program counter to target program counter in an address lookup table.
4. Execute the translated block.
5. Repeat for next discovered block.

The translation blocks are similar to basic block by design [3]. Leveraging this correspondence between translation blocks and basic blocks and because execution is handed back to the emulator after each run, it is possible to implement an instrumentation similar to the compile time instrumentation, using the program counter as feedback on each new basic block. AFL, a common feedback-based fuzzer, offers this instrumentation with *QEMU mode*. AFL QEMU-mode emulates fuzz targets in a patched version of QEMU that reports executed branches back to AFL. To increase speed, a forkserver resets the state to a fully loaded initial state between fuzz runs. After a new basic block is translated, the fork child's parent will also translate it and cache te result to ensure the costly translation won't be necessary again after the next fork. The control returns to QEMU after each block, which is extended with calls to afl_maybe_log, a function that passes the coverage map back to AFL via shared memory.

3.1 AFL-Unicorn

The AFL-Unicorn mode makes use of Unicorn Engine, a lightweight CPU-emulator originally forked from the aforementioned QEMU [31]. In AFL-QEMU, the emulator extends every basic block to send instrumentation information to AFL and starts AFL's forkserver after the emulator has been set up. For speed, basic blocks are cached in the parent parent process in Unicorn's AFL forkserver. AFL itself merely has to start the harness and generate inputs. The inner workings of the *Unicorefuzz* forkserver, are depicted in fig. 3.3. The figure can be read as a timeline, from left to right. Initially, AFL spawns the *Unicorefuzz* harness, which loads the initial kernel memory (in the middle). The kernel part of the harness then forks child processes, each of which run until their exit is reached. If the child kernel (bottom) dynamically requires

new memory regions or translates a new basic block, it will report back to the parent. Blocks are reported back through a pipe and then translated by the parent, so that they will already be translated for the next run, memory is stored to disk. This enables AFL-Unicorn to fuzz any binary-only machine code of all supported frontend processor architectures on every backend processor architecture [41].

3.2 avatar^2

As another piece of *Unicorefuzz*, avatar^2 is an orchestration framework, abstracting over a range of debugger and binary analysis targets [30]. As discussed below, *Unicorefuzz* makes heavy use of avatar^2 to dump process memory and registers of the fuzz target. The orchestration framework can attach to binary analysis frameworks, smart devices, any GDB session, as well as PANDA and QEMU VMs.

3.3 *Unicorefuzz*

In the following, we discuss *Unicorefuzz*, a framework able to fuzz most targets through emulation.

3.3.1 Preparation

Since AFL-Unicorn is capable of fuzz testing arbitrary binaries, it is also able to fuzz kernel components. While it interaction with hardware or emulated devices is not part of Unicorn, it can emulate almost any code. Since modelling periphery would not be an easy task in Unicorn, we focus on the emulation of parsers. These parsers usually work on buffers of untrusted input and don't require responses from periphery devices. In the preparation step, *Unicorefuzz* needs to copy over all registers, insert the input generated by AFL and run the emulation. For this, *Unicorefuzz* makes use of the avatar^2 framework to interact with a gdb stub. The user has to specify a breakpoint, as well as a memory region for AFL input in the harness. Although real devices could be fuzzed using *Unicorefuzz*, we used virtual machines with breakpoints at the target function. The user has to specify the *fuzz target address* in the wrapper, start it and make the virtual machine execute the desired function. Once the breakpoint fires, it freezes the VM, dumps all required registers and waits for dynamic mapping memory requests.

3.3.2 Dynamic Mapping

While dumping the entire memory of a target and mapping it in the harness to fuzz a function would be possible, it would defeat the purpose and cause unnecessary memory overhead. By keeping the target VM alive (in a halted state) and creating a communication channel to the fuzzer, *Unicorefuzz*'s probe wrapper will load new memory pages on demand, see 1. Whenever Unicorn accesses unmapped memory, it calls the UC_HOOK_MEM_UNMAPPED handler, which sends a memory info request to the *probe wrapper* component of *Unicorefuzz*. The request is either processed successfully (if the requested memory is valid) and mapped in the emulator, or rejected, in which case *Unicorefuzz* forces a crash. If it succeeds, the corresponding memory region is also cached to disk. Otherwise an invalid memory access was found and the process is killed with SIGSEGV — the crash will be reported to AFL.

3.3.3 Fuzz-Test Execution

Due to the caching mechanisms, no interaction with the VM is needed anymore — the mappings are pre-fork at this point. The harness of *Unicorefuzz* creates a Unicorn Engine object, maps all required and previously requested memory regions and sets the registers' values accordingly. The input has to be passed to Unicorn by calling mem_write() or reg_write (). After the user specified the addresses for fuzz input, the AFL forkserver is started by running the emulation for one instruction. When the emulation references unmapped memory again, the harness checks whether the memory region was requested in previous runs. If the probe wrapper's output directory already contains the dumped memory or a rejection, it can continue or abort immediately. Otherwise it creates an empty file in the wrapper's input directory (where the filename denotes the requested address) and polls the wrapper's output directory for the memory region dump, or the rejection. If a file in the input folder exists, it tries to read the requested memory from the virtual machine and dumps it to the output folder if the memory was successfully read, or creates a rejection file otherwise.

4 Evaluation

To evaluate viability of emulator based kernel fuzzing, the concept was tested against both synthetic and real-world targets.

4.1 broken.ko

For performance analysis and tooling tests, we created a variety of intentionally broken kernel modules, where *Unicorefuzz* is able to find bugs almost immediately. Take as example the following module:

Listing 1: Crashing Kernel Module

```
static ssize_t
write_callback(struct file *file, const char __user *buf,
                        size_t len, loff_t *offset) {
  if (buf[0] == 'A') {
    int a = 2;
    a -= 2;
    if (5/a > 0) {
      printk(KERN_INFO "this will never happen!\n");
    }
  }
  return len;
}
```

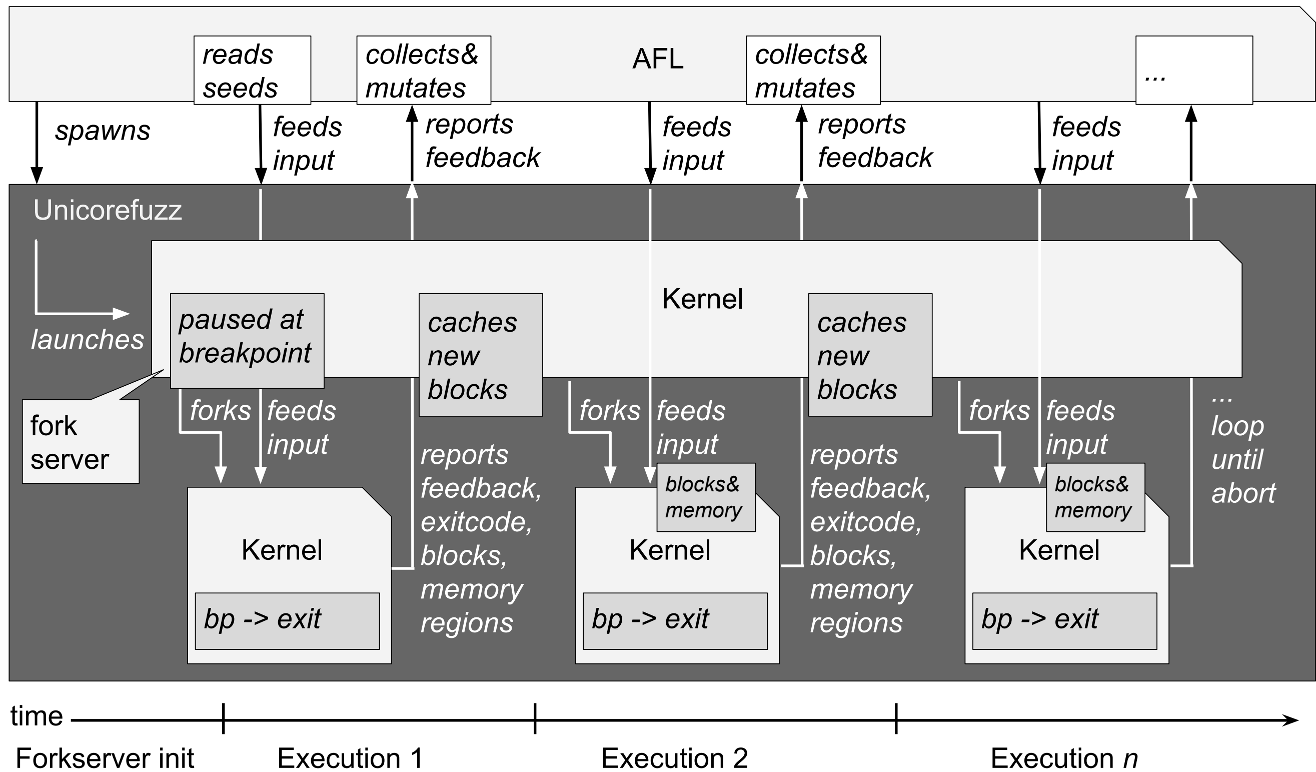

Figure 2: Simplified overview of AFL-Unicorn forkserver with *Unicorefuzz* additions.

The kernel module in listing 1 crashes when it receives a certain input over its procfs entry. Its procfs write_callback divides by zero if (and only if) the first byte of the input is 0x41 ('A'): Setting up a testcase is trivial, however still needs manual work:

1. Determine the address and return address of write_callback , add it to the *Unicorefuzz* harness.
2. Determine the input buffer address and add it to the *Unicorefuzz* probe wrapper
3. Fuzz

Even though the AFL component was started with an almost empty input directory, it was able to find a crashing input immediately with no false positives and no hangs. The compiler had replaced the division by zero with an invalid opcode, but *Unicorefuzz* catches invalid instructions and appropriately reports them as crashes. This and other test cases prove the viability as kernel fuzzer. Note that the procfs entry was just an example, the concept also works for other functions in the kernel.

4.2 Open vSwitch

Bugs in software that has to parse networking input are often security issues with dangerous impact. Since previous work has discovered bugs in Open vSwitch's user space components [40], their kernelspace counterpart, which is in the Linux source tree, poses an interesting target.

4.2.1 sk_buff

The function key_extract extracts a so-called *flow key* from an ethernet frame. It takes a pointer to an sk_buff containing the packet and a pointer to an sw_flow_key to which the flow key is written to. The probe wrapper was adapted to break on key_extract . Passing the correct parameters to key_extract needs manual analysis work. An sk_buff has multiple fields that could be exposed to the fuzzer. Figure 3 depicts a basic overview of the relevant parts in an sk_buff. Other variables other than the data buffer might be hard to fuzz, since the data structure is preprocessed by Linux. Fuzzing variables that pose as pointer would definitely yield many false-positives, even more so on internal APIs. This is why we decided on fuzzing the data part as we expected this to lead to the most promising results. Nevertheless, fuzzing metadata is an interesting subject, but it will be close to impossible to differentiate

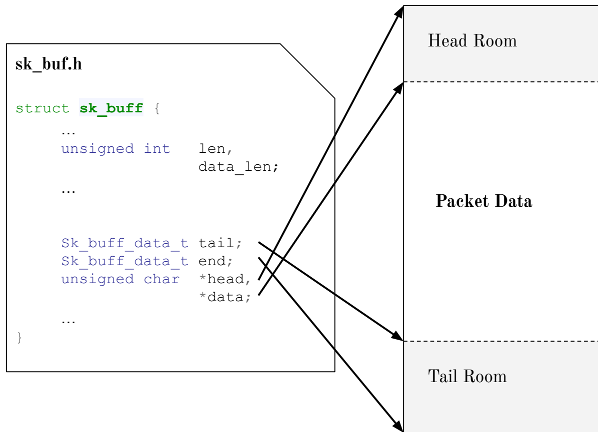

Figure 3: Layout of sk_buf and its data

true bugs from false positives. In the end, we only fuzzed the data packet data.

4.2.2 Results

After more than 6500 cycles, AFL found 62 new edges and the input got copied to the correct address. There were no crashes or hangs found in key_extract. Possibilities are that even if we had corrupted logics in the out parameter (sw_flow_key), bugs would not occur until later in the code. Choosing a larger scope, starting further up in the stacktrace and running larger chunks of code, a few functions up the stacktrace, will be tested in the future.

4.3 Rediscovering Syzbot Bugs

The team behind syzkaller [17] offer syzbot, an open dashboard listing hundreds of bugs found with syzkaller. As part of this evaluation, we chose a reproducible syzkaller bug in:

```
int ip_do_fragment(struct net *net, struct sock *sk, struct sk_buff *skb,
        int (*output)(struct net *, struct sock *, struct sk_buff *))
```

The function appeared perfect, since it triggers an actual BUG() deeper in the function and is calls using sk_buf, a the same internal socket data structure we already fuzzed for Open vSwitch. After setting up the correct kernel function, the breakpoint at the function entry and executing the reproduction script, *Unicorefuzz* uncovers the bug through fuzzing. While successful this time, we don't deem the results valuable. In this scenario, the achievement of syzkaller was getting the kernel to a state where this function would crash, something *Unicorefuzz* does not address, since state is being reset after each function call.

Listing 2: A "ported" kernel module

```
#include <stdlib.h>
#include <stdio.h>

static size_t
write_callback(void* file, char* buf,
               size_t len, void* offset) {
  if (buf[0] == 'A') {
    int a = 2;
    a -= 2;
    if (5/a > 0) {
      printf("nop\n");
    }
  }
}

void
main() {
  char input[1024];
  fgets(input, sizeof(input), stdin);
  write_callback(0, input, 0, 0);
}
```

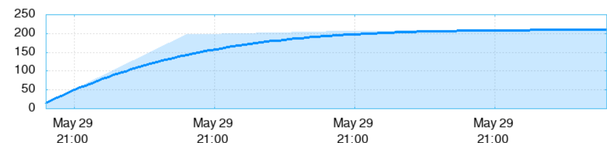

Figure 4: Around 200 Execs/Sec when emulating a larger kernel function on Intel Core i5-5257U (2.7 GHz)

4.4 Speed

To evaluate the performance impact of *Unicorefuzz*, we compared it's performance against AFL's QEMU mode. Since QEMU mode can only fuzz user space binaries, the tested function from the kernel module in section 4.1 was ported into a user space binary. Because of the low complexity of the function with minimal dependencies, porting was trivial. The entire source code is depicted in listing 2. The tests were run on a ThinkPad T520 (Intel Core i7-2670QM @2.20GHz), the results are shown in figure 4.4. The Unicorn based fuzzing setup yields 47% the amount of execs/per second, when compared with the user space binary in QEMU mode, which — in turn — has performance overhead over a natively compiled AFL, see figure 4.

Framework	Execs/Sec
Unicorefuzz	459
AFL QEMU Mode	939
AFL	4860

Figure 5: Speed comparison of AFL, AFL in QEMU mode and *Unicorefuzz*

5 Discussion

In the following, we will discuss design decisions as well as practical roadblocks we encountered while building *Unicorefuzz*.

5.1 Seeds

Collecting valid input for a program poses a simple and effective method for initial input generation. The valid use of a tool, or function call in the case of *Unicorefuzz*, contains domain specific knowledge about the tool in question. Larger test cases take longer to execute since more data has to be processed. Many similar seeds covering the same code flow lead to runs that take additional time without ever triggering new and interesting corner cases. Empty input or single bytes hardly ever trigger code paths in fuzz targets. Instead, some sort of initial input or a way to generate it is usually applied when fuzzing a target. For *Unicorefuzz* users can collect data of the parameters when breakpoints are triggered. However, mindlessly following this methodology might not be successful, since, depending on the parameter, internal data structures might have changed between calls. On top seeds must not destroy current data structures and parameters.

5.2 Sanitization

Muench et al. classify the results of memory corruptions found through fuzzing in different categories, of which *observable crashes* and *hangs* are easy to track. *Late crashes* and *malfunctioning* may be hard to spot while the class of *no effect* is impossible to track down with no added observation channels like address sanitization. Sanitization methods check if certain assumptions hold true at any given time and report fails or even shut down the target in such a case. Many vulnerabilities in binary application written in C/C++ are memory corruption bugs, which are triggered, for example, by an out of bounds read or an out of bounds write. However, not every input that triggers a memory corruption bug translates to necessarily crashing input immediately. The memory corruption itself and the subsequent use of the corrupted memory, which triggers the program crash, may often be far apart. Thus, even when a crashing input is given, it may be hard to actually detect the underlying memory bug. We evaluated *Unicorefuzz* against Kernels built with *Kernel Address Sanitizer (KASAN)*. `Address Sanitizers` use compile time instrumentation in order to detect, for instance, out-of-bounds memory access and use-after-free bugs [35]. KASAN makes use of a shadow memory to assess whether a particular byte of the kernel memory is safe to access. Address sanitization introduces relatively low performance costs due to the way it maps memory to the shadowed memory [35]. Since the Linux kernel on x64 makes use of obscure functionality, `KASAN` broke Unicorn in the beginning and thus our test cases had to be patched. Thanks the fixes, however, we are able to make full use of KASAN, should a kernel and module be built with it.

5.3 Replayability

Even though valid input leads to a valid crash, depending on the preconditions, the input may not immediately be replayable. In the case of Unicorefuzz, however, all crashes are 100% replayable. Unicorefuzz, as opposed to syzkaller, is stateless. Stateless fuzzing calls the target over and over again, from the same test state. In this case, it is the complex state, but the same properly initialized state for every execution. The input state is persisted to disk, all input is persisted through the usual AFL folder structure. Since Unicorn does not have external input sources in this scenario, replaying one input will always yield the same crash.

5.4 Constraining Inputs

Fuzzing a function that is located inside of a module and expects the input to be according to a certain structure, can be challenging. An example of such a function could be a parser that expects an sk_buf, as discussed in 4.2.1. Since this function is embedded inside the kernel, there are other layers above checking the input before the function is called. If the function is then triggered directly by the fuzzer, an unconstrained input can lead to crashes that can never occur as the state is already constrained inside the calling functions. Thus, false positives are likely. When constraining the input more than necessary, the fuzzer misses bugs. It's a non-trivial problem that still requires manual labor. However, definitions which fields of common structures should and should not be open to the fuzzer may be shared.

5.5 The Curious Case of GS_BASE

The segment registers are not used in their original — x86 — context in x86_64 bytecode, but the GS register instead has a GS_BASE register that is used to access CPU-specific memory. The same is true for FS_BASE Among others, internal bookkeeping, stack canaries and the kernel address sanitizer (KAZAN) use this register, so support for it has to be ensured. The FS_BASE and GS_BASE are mapped to model-specific registers (C000_0100h and C000_0101h) [1]. Model-specific registers (MSRs) were originally experimental CPU registers that were not guaranteed to be built into successor models, however GS and FS base are commonly used. Still, Unicorn Engine, at the time of writing, does not offer an API to interact with these registers directly. Instead, for *Unicorefuzz* we have to map some unmapped space and write the base registers using shellcode for the wrmsr instruction.

Features	Trinity	TriforceAFL	In-Kernel AFL	syzkaller	kAFL	Userland Port	Unicorefuzz
Fuzz Anywhere	-	-	-	-	-	+	+
Peripherals	++	+	-	+	+	-	-
Binary-Only	-	+	-	-	+	-	+
Multi Arch	+	+	?	+	-	+	+
Speed	++	-	-	+	+	++	-
Instrumentation	-	AFL-QEMU	KCOV	KCOV	Intel PT	Any	AFL-Unicorn

Figure 6: Kernel fuzzing methods comparison

5.6 Initial Emulated Operation

AFL-Unicorn starts the forkserver after executing the first instruction, then loads the input [41]. If the first instruction interacts with a part of the input, the results of that instruction are not based on the appropriate modified input from AFL, but on the original input from the target. Since the segment register workaround described in 5.5 has to execute shellcode and therefore, first instructions, it mitigates this problem on X86_64 by coincidence.

5.7 Scheduling

AFL-Unicorn emulates only one processor core, so, while emulating the scheduler on a single core works, fuzzing functions that need an active secondary core to complete will not work. Furthermore, if a function needs access to a lock-protected resource whose lock is busy at the moment the target's breakpoint fires, it cannot return since no other core will ever release the lock.

5.8 Race Conditions

A single processor emulator setup with a tight harness will not find race conditions because there is no other load on the system. There are no other CPUs which could introduce races for locks or concurrent data structures and when fuzzing an individual, synchronous kernel function, no user space programs will interfere. While this greatly reduces the odds of discovery of race condition-related bugs, it makes crashes perfectly reproducible.

5.9 Comparison Against Other Fuzzers

Evaluating fuzzer performance is hard (see section 6.1). However, we can compare certain features. As outlined in figure 6, the non-emulated fuzzers don't incur the emulators' performance impact, but lack the ability to target binaries where the source code is not available, with the exemption of kAFL which only works on recent Intel processors. Despite the usefulness of the execution speed being boosted by proper instrumentation, it has to be acknowledged that the raw throughput of non-emulating fuzzers may lead to faster results, if their user space agents can quickly trigger the desired target function. If the target is not easily triggered by a user space agent, the targeted approach of Unicorefuzz is superior, as one halted virtual machine is enough to conduct an arbitrary number of possibly distributed fuzz tests.

6 Future Work

While we presented novel research along the field of fuzzing during the course of this paper, there is plenty of room for improvements.

6.1 Unified Evaluation Criteria

For years, researchers have been searching for a way to properly evaluate fuzzer performance against each other, with limited success. In 2009, Clarke already stated that "it is impossible to accurately measure the effectiveness of fuzzing, since the only practical metric, code coverage, only measures one 'dimension' of fuzzing: the amount of (reachable) code executed; it does not measure the range of input values fed to the target at each code region. We have also seen that target monitoring is often less than ideal" [8].

Dolan-Gavitt et al. propose LAVA-M, a test set for binary analysis that can be used for performance evaluations [14], however artificial benchmarks may lead to overfitting. Mu et al. show that reproducibility of real-world security vulnerabilities, however, is complex, nevertheless their efforts could be used to partly evaluate os level fuzzers like syzkaller [29]. Notably, Trent Bunson postulates how to spot good Fuzzing [5]. Klees et al. perform a scientific evaluation of different fuzzers with a larger number of runs [25]. We demand a similar test suite for kernel bugs.

6.2 Embedded Fuzzing

Since every embedded processor has some debugging port which can export memory pages and register values, adapting the setup to embedded operating systems fuzzing is viable. A single target is enough to answer dynamic memory mapping requests, so distributing the fuzzing to multiple local or remote machines is also straightforward.

6.3 Static Rewriting

Instead of adding instrumentation at compile time, or dynamically at runtime, it can also be added using static analysis. The obvious benefit is an increased throughput as opposed to emulated instrumentation. Projects that modify (userland-)binaries to add them, report code coverage and fuzz them exist [32].

6.4 Emulation Performance Increases

Fuzzing speed depends on various factors. Recent advancements focus on speed of path finding, but also on sheer execution speed of the instrumentation through lightweight hardware features [34, 44]. Current research has shown that AFL's QEMU mode's performance can be improved by re-enabling QEMU's block chaining, which merges code blocks if one ends with a direct jump. It is disabled because it interferes with AFL's instrumentation: merged blocks don't jump back into the emulator after every single contained block, so it effectively disables tracing direct jumps. The author injects the instrumentation code into the translated code and thus can safely enable block chaining. Combined with proper caching, this yields a speedup of 3-4 times the mainline QEMU mode [4]. This patch could be ported to AFL-Unicorn and could significantly reduce the performance gap to compiler-assisted instrumentation. For this to work, however, further patches to Unicorn itself are needed: block chaining was removed from Unicorn, presumably to decrease code complexity and simplify hooking. Given enough time and energy, it might even be possible to port vectorized emulation, as proposed by Falk, to AFL Unicorn [19].

6.5 Triaging

Fuzzing outputs a large amount of potential bugs. Depending on the setup, they may be false-positives (especially when underconstraining parameters), or all represent the same bug. Fuzz testing yields numerous crashes, even for a single application [28]. The amount of crashes may be too high to manually audit [7]. To keep pressure from the manual analyst, a fuzz setup usually triages the target in some way. The amounts of data are, therefore, unstructured:

1. Fuzzing a function may lead to multiple crashing inputs, which can be attributed to the same bug.

2. Not every crash points to a bug that poses a security threat. But often both security researchers as well as developers want to focus on bugs that expose security vulnerabilities. The fuzzing pipeline needs to assess the severeness of a crash [28].

3. We also want to find the underlying bug that caused a crash. This step has been done in a manual fashion so far, although there are attempts to automate it: DeMott et al. identify which part of the source code is responsible for the bug. Using the assumption that lines of code responsible for a bug are mainly exercised by crashing inputs, they calculate a suspiciousness score for each line of code [13].

Deduplication and analysis of crashes can be done by examining the similarity of the program's stack trace at the time of crashing. Methods that do so have already been employed successfully [10]. Similarly, heuristics exist to classify crashes according to the exploitability of the bug that caused the crash [20]. The heuristics work by executing the program with the crashing input and then inspecting the crashing state. For example, a NULL-pointer dereference bug can cause a crash, but is hard to exploit or not at all exploitable on modern operating systems. However, if the EIP register can be controlled with user defined input, the bug is extremely critical [37]. Huang et al. and Thanassis et al. both propose to automatically generated exploits from crashes. If a crash can be marked exploitable fully automated, the bug is critical [2, 22]. For other bugs, assessing a crash properly is not trivial.

7 Conclusion

After evaluating the current state of the art, we provide an implementation of a novel fuzzer, emulating kernel code. We show that fuzzing kernel functions with an emulator is possible, viable and relatively easy to set up, even if the target function is not exposed to the user space. With this method, any parsing functions can be fuzzed with coverage-guided feedback, as long as they do not interact with hardware. Though the use of an emulator with support for memory access hooks has an obvious impact on execution speed. Still, the throughput is at roughly 46% of AFL's QEMU usermode — and thus acceptable.

The innate advantage of being able to start a fuzz-test at any point in the kernel code, even on binary blobs and across architectures is, to the best of our knowledge, unmatched by other approaches to kernel fuzzing. We do see drawbacks in the manual overhead needed to chose valid regions for parameter structs or arrays, however we hope to improve upon this point. Kernel fuzzing has already found a significant amount of bugs when only looking from afar — the syscall API — but tests could never fuzz internal functions directly. Unicorn is able to emulate a number of processor architectures, whose kernels can be easily fuzzed with the described technique, if it is possible to write a probe wrapper for that platform. *Unicorefuzz* is able to execute everything where no multiprocessor coordination is required. We will open source the implementation of *Unicorefuzz* and hope to improve on the concept further.

Acknowledgments

The authors wish to thank Vincent Ulitzsch, Fabian Freyer and Marius Muench for valuable input.

References

[1] AMD: AMD64 Architecture Programmer's Manual Volume 2: System Programming. Advanced Micro Devices

[2] Avgerinos, T., Cha, S.K., Rebert, A., Schwartz, E.J., Woo, M., Brumley, D.: Automatic exploit generation. Commun. ACM **57**(2), 74–84 (Feb 2014). https://doi.org/10.1145/2560217.2560219

[3] Bellard, F.: Qemu, a fast and portable dynamic translator. In: USENIX Annual Technical Conference, FREENIX Track. vol. 41, p. 46 (2005)

[4] Biondo, A.: Improving afl's qemu mode performance. 0x41414141 in ?? () (Sep 2018), https://abiondo.me/2018/09/21/improving-afl-qemu-mode

[5] Brunson, T.: How to Spot Good Fuzzing Research (Oct 2018), https://blog.trailofbits.com/2018/10/05/how-to-spot-good-fuzzing-research, [Online; accessed 11. Nov. 2018]

[6] Butti, L., Tinnes, J.: Discovering and exploiting 802.11 wireless driver vulnerabilities. Journal in Computer Virology **4**(1), 25–37 (2008)

[7] Chen, Y., Groce, A., Zhang, C., Wong, W.K., Fern, X., Eide, E., Regehr, J.: Taming compiler fuzzers. In: ACM SIGPLAN Notices. vol. 48, pp. 197–208. ACM (2013)

[8] Clarke, T.: Fuzzing for software vulnerability discovery. Department of Mathematic, Royal Holloway, University of London, Tech. Rep. RHUL-MA-2009-4 (2009)

[9] Corina, J., Machiry, A., Salls, C., Shoshitaishvili, Y., Hao, S., Kruegel, C., Vigna, G.: DIFUZE: interface aware fuzzing for kernel drivers. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 2123–2138. ACM (2017). https://doi.org/10.1145/3133956.3134069

[10] Dang, Y., Wu, R., Zhang, H., Zhang, D., Nobel, P.: Rebucket: A method for clustering duplicate crash reports based on call stack similarity. In: 2012 34th International Conference on Software Engineering (ICSE). pp. 1084–1093 (Jun 2012). https://doi.org/10.1109/ICSE.2012.6227111

[11] DeMott, J.: The evolving art of fuzzing. DEF CON **14** (2006)

[12] DeMott, J., Enbody, R., Punch, W.F.: Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing. BlackHat and Defcon (2007)

[13] DeMott, J.D., Enbody, R.J., Punch, W.F.: Towards an automatic exploit pipeline. In: Internet Technology and Secured Transactions (ICITST), 2011 International Conference for. pp. 323–329. IEEE (2011)

[14] Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., Ulrich, F., Whelan, R.: LAVA: Large-scale automated vulnerability addition. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 110–121 (May 2016). https://doi.org/10.1109/SP.2016.15

[15] Drewry, W., Ormandy, T.: Flayer: Exposing application internals. WOOT **7**, 1–9 (2007)

[16] Drewry, W.A., Ormandy, T.: Software testing using taint analysis and execution path alteration (Feb 2013), uS Patent 8,381,192

[17] Drysdale, D.: Coverage-guided kernel fuzzing with syzkaller (2016), https://lwn.net/Articles/677764/

[18] Duran, J.W., Ntafos, S.: A report on random testing. In: Proceedings of the 5th international conference on Software engineering. pp. 179–183. IEEE Press (1981)

[19] Falk, B.: Vectorized Emulation: Hardware accelerated taint tracking at 2 trillion instructions per second (Oct 2018), https://gamozolabs.github.io/fuzzing/2018/10/14/vectorized_emulation.html, [Online; accessed 11. Nov. 2018]

[20] Foote, J.: Exploitable (2018), https://github.com/jfoote/exploitable, [Online; accessed 2018-09-09]

[21] Hertz, J., Newsham, T.: Project triforce, https://raw.githubusercontent.com/nccgroup/TriforceAFL/master/slides/ToorCon16_TriforceAFL.pdf

[22] Huang, S.K., Huang, M.H., Huang, P.Y., Lai, C.W., Lu, H.L., Leong, W.M.: Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. In: 2012 IEEE Sixth International Conference on Software Security and Reliability. pp. 78–87 (Jun 2012). https://doi.org/10.1109/SERE.2012.20

[23] Kernel.org: kcov: code coverage for fuzzing, https://www.kernel.org/doc/html/v4.17/dev-tools/kcov.html

[24] Kerrisk, M.: Lca: The trinity fuzz tester (2013), https://lwn.net/Articles/536173/

[25] Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M.: Evaluating fuzz testing. CoRR **abs/1808.09700** (2018), http://arxiv.org/abs/1808.09700

[26] Lemieux, C., Sen, K.: Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage. CoRR **abs/1709.07101** (2017), http://arxiv.org/abs/1709.07101

[27] LLVM Project: libFuzzer – a library for coverage-guided fuzz testing. (Sep 2018), https://llvm.org/docs/LibFuzzer.html, [Online; accessed 15. Sep. 2018]

[28] Miller, C.: Crash analysis with bitblaze. Blackhat (2010)

[29] Mu, D., Cuevas, A., Yang, L., Hu, H., Xing, X., Mao, B., Wang, G.: Understanding the reproducibility of crowd-reported security vulnerabilities. In: Enck, W., Felt, A.P. (eds.) 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018. pp. 919–936. USENIX Association (2018), https://www.usenix.org/conference/usenixsecurity18/presentation/mu

[30] Muench, M., Francillon, A., Balzarotti, D.: Avatar²: A multi-target orchestration platform. In: BAR 2018, Workshop on Binary Analysis Research, colocated with NDSS Symposium, 18 February 2018, San Diego, USA. San Diego, UNITED STATES (Feb 2018), http://www.eurecom.fr/publication/5437

[31] Ngyuen, A.Q., Dang, H.V.: Unicorn: Next generation cpu emulator framework, http://www.unicorn-engine.org/BHUSA2015-unicorn.pdf

[32] Nikolich, A.: Afl fuzzing blackbox binaries (2015), https://groups.google.com/forum/#!topic/afl-users/HlSQdbOTlpg

[33] Nossum, V., Casasnovas, Q.: Filesystem fuzzing with american fuzzy lop (2016), https://events.static.linuxfound.org/sites/events/files/slides/AFL%20filesystem%20fuzzing,%20Vault%202016_0.pdf

[34] Schumilo, S., Aschermann, C., Gawlik, R., Schinzel, S., Holz, T.: kafl: Hardware-assisted feedback fuzzing for OS kernels. In: 26th USENIX Security Symposium (USENIX Security 17). pp. 167–182. USENIX Association, Vancouver, BC (2017)

[35] Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: Addresssanitizer: A fast address sanity checker. In: USENIX Annual Technical Conference. pp. 309–318 (2012)

[36] Serebryany, K.: Oss-fuzz-google's continuous fuzzing service for open source software. In: USENIX Security Symposium (2017)

[37] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In: IEEE Symposium on Security and Privacy (2016)

[38] Song, D., Hetzelt, F., Das, D., Spensky, C., Na, Y., Volckaert, S., Vigna, G., Kruegel, C., Seifert, J., Franz, M.: Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In: 26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019. The Internet Society (2019), https://www.ndss-symposium.org/ndss-paper/periscope-an-effective-probing-and-fuzzing-framework-for-the-hardware-os-boundary/

[39] Sthamer, H.H.: The automatic generation of software test data using genetic algorithms. Ph.D. thesis, University of Glamorgan (1995)

[40] Thimmaraju, K.: Ve-2018-1000155: Denial of service, improper authentication and authorization, and covert channel in the openflow 1.0+ handshake (2018), https://www.openwall.com/lists/oss-security/2018/05/09/4

[41] Voss, N.: afl-unicorn: Fuzzing arbitrary binary code (October 2017), https://hackernoon.com/afl-unicorn-fuzzing-arbitrary-binary-code-563ca28936bf

[42] Vyukov, D.: Add fuzzing coverage support (2015), https://gcc.gnu.org/viewcvs/gcc?view=revision&revision=231296

[43] Zalewski, M.: Technical "whitepaper" for AFL-fuzz (2016)

[44] Zhang, G., Zhou, X., Luo, Y., Wu, X., Min, E.: Ptfuzz: Guided fuzzing with processor trace feedback. IEEE Access **6**, 37302–37313 (2018). https://doi.org/10.1109/ACCESS.2018.2851237