Drifuzz: Harvesting Bugs in Device Drivers from Golden Seeds

Zekun Shen, Ritik Roongta, and Brendan Dolan-Gavitt, *NYU*

https://www.usenix.org/conference/usenixsecurity22/presentation/shen-zekun

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open access to the Proceedings of the 31st USENIX Security Symposium is sponsored by USENIX.

Drifuzz: Harvesting Bugs in Device Drivers from Golden Seeds

Zekun Shen
New York University
zekun.shen@nyu.edu

Ritik Roongta
New York University
rr3953@nyu.edu

Brendan Dolan-Gavitt
New York University
brendandg@nyu.edu

Abstract

Peripheral hardware in modern computers is typically assumed to be secure and not malicious, and device drivers are implemented in a way that trusts inputs from hardware. However, recent vulnerabilities such as Broadpwn have demonstrated that attackers can exploit hosts through vulnerable peripherals, highlighting the importance of securing the OS-peripheral boundary. In this paper, we propose a hardware-free concolic-augmented fuzzer targeting WiFi and Ethernet drivers, and a technique for generating high-quality initial seeds, which we call *golden seeds*, that allow fuzzing to bypass difficult code constructs during driver initialization. Compared to prior work using symbolic execution or greybox fuzzing, Drifuzz is more successful at automatically finding inputs that allow network interfaces to be fully initialized, and improves fuzzing coverage by 214% (3.1×) in WiFi drivers and 60% (1.6×) for Ethernet drivers. During our experiments with fourteen PCI and USB network drivers, we find twelve previously unknown bugs, two of which were assigned CVEs.

1 Introduction

Built-in peripherals such as PCI devices have traditionally been considered trusted. As a result, device drivers are often written with the assumption that these peripherals' control interfaces are reliable (although they may ultimately deliver untrusted data, such as network packets, to higher layers). Modern peripherals, however, are complex devices with their own firmware and vulnerabilities. If the firmware of a peripheral is compromised, the device can be used to launch attacks on the main OS and gain control over the whole system.

Recent vulnerabilities in WiFi [2,42] and Bluetooth [33,44] adapters illustrate the possibility of attacking the host remotely through flaws in device drivers. Although this attack vector is complex and requires chaining multiple vulnerabilities from device drivers and device firmware, it demonstrates the possibility of gaining kernel privilege discreetly. In other cases, corrupted or buggy peripherals can affect the host kernel via memory errors in device drivers such as out-of-bound writes. Moreover, even without finding vulnerabilities in the peripheral firmware, attackers with physical access can perform "evil maid" attacks by connecting a malicious device that identifies itself as a supported peripheral and then exploits vulnerabilities in that device.[1]

There are two major security boundaries for a device driver: the userspace-OS boundary and the OS-peripheral boundary. Many works [10,12,21,27,37,41,52] in device driver security focus on the former boundary by testing `ioctl` system calls. Although some prior work [3, 47, 48, 50] has investigated the low-level OS-peripheral boundary, it is overall less well-explored. The latter boundary consists of hardware-related inputs including port I/O, Memory Mapped I/O (MMIO), Direct Memory Access (DMA), and interrupts. MMIO is used to access individual hardware registers while DMA is used for bulk transfers. Finally, hardware interrupts cause the CPU to halt and transfer control to an interrupt handler within the device driver.

A major challenge in testing device drivers is the diversity of hardware peripherals. Some prior work [3, 47, 50] has assumed the presence of actual hardware peripheral for each driver under test, but such approaches are difficult to scale (since multiple physical devices are needed in order fuzz in parallel) and can be expensive (since one must obtain the target hardware; for older drivers obtaining real hardware may not even be feasible).

An attractive alternative to using real hardware is to test the driver in an emulated environment such as QEMU. Emulators do not need any specialized hardware, and testing can be scaled up by simply launching more copies of the emulator. This approach raises new challenges, however. Outside of the relatively small set of peripherals supported by QEMU, testing a device driver requires some understanding of the expected behavior of the corresponding peripheral. For example, many device drivers probe the values of identification

[1] Although historically such attacks (at least over PCI) were trivial due to unrestricted direct memory access (DMA) to RAM, in modern systems the increasingly prevalent use of IOMMUs means that malicious peripherals typically only have access to a small portion of RAM.

and status registers during initialization, and will regard the device as broken or unsupported if the expected values are not returned. A failed check on such an important path results in immediate termination, leaving further driver code untested and any deeper bugs undiscovered.

This problem is exacerbated by code patterns such as magic value checks and polling loops in driver code, which present roadblocks to traditional fuzzing techniques that generate random values. We regard these roadblocks as **blocking branches** and find them pervasive in device drivers. Our key insight in this work is that while there are many ways for driver initialization to fail, there is typically a "golden path" that passes these checks and fully initializes the device driver. Without knowledge of this golden path, fuzzing can only find shallow paths where the driver terminates early. We introduce a technique that combines concolic testing and forced execution to generate "golden seeds": fuzzer inputs that can be used to guide driver initialization. Concolic execution allows Drifuzz to quickly solve hard-to-fuzz constraints such as magic value checks, while forced execution lets us bypass repetitive checks (e.g., polling loops) that would otherwise hinder both concolic and symbolic approaches.

We leverage this idea to build Drifuzz, a concolic-assisted fuzzer designed to uncover device driver vulnerabilities at the OS-peripheral boundary. We implement our system as a hybrid fuzzer: after generating a golden seed we alternate between coverage-guided random fuzzing and concolic execution. Although hybrid fuzzing is a common technique in other domains [21, 35, 36, 49, 54], it has not previously been used at the OS-peripheral boundary. Hybrid fuzzing is particularly effective for testing device drivers, which have relatively low execution throughput and code patterns that random fuzzing struggles with.

In our experiments we find that starting hybrid fuzzing with a golden seed improves coverage by 214% (3.1×) for complex WiFi drivers. On simpler drivers such as Ethernet NICs, we find that Drifuzz outperforms the baseline fuzzer and achieves 60% (1.6×) more coverage. In addition, we find six new bugs across the ten tested PCI drivers and six bugs in four tested USB WiFi drivers.

We make the following contributions:

- We introduce a technique for generating "golden seeds" that allow device driver code to execute successfully *without* access to real hardware, allowing fuzzing to reach deeper into driver code.
- We implement these ideas by adding support for concolic execution and forced execution to the PANDA dynamic analysis platform; to the best of our knowledge, Drifuzz is the first whole-system hybrid fuzzer that tests the OS-peripheral boundary.
- We use Drifuzz to find and fix 12 Linux device driver bugs, two of which were assigned CVEs.

2 Background

2.1 Device Driver Security

Operating system kernels are implemented as the abstraction between user-level software and hardware. Between the hardware and kernel subsystem abstraction lie the device drivers. Because there is a wide variety of hardware to support, device drivers are many and varied. This makes it difficult for maintainers to audit the security of device drivers; Renzelmann et al. [38] point out that device drivers also generally have low test priority. At the same time, device drivers run in kernel mode and are highly privileged, making them compelling targets.

Device drivers are exposed to attacker-controlled input from two sources: userspace and hardware. A Linux userspace program can use the `ioctl` system call to invoke and exploit a driver to perform a local privilege escalation to kernel-mode. The other kind of attack comes from the hardware side; here the peripheral is assumed to be either malicious or compromised. In this paper, we focus on the hardware attack vector in PCI and USB devices such as WiFi and Ethernet peripherals. Because they receive external inputs via Ethernet or wireless signals, vulnerabilities in these drivers may be exploitable remotely and discreetly.

2.2 Whole System Emulation and Analysis

Full-system emulation using QEMU [1, 13] is useful for testing kernel subsystems and drivers. Using a full-system emulator allows us to attach an emulated hardware peripheral and respond to interactions from the OS with fuzzed input. Prior work, such as USBFuzz [34], Syzkaller's USB Fuzzing [12] and Agamotto [48], also use whole-system emulation for this purpose.

KVM and TCG Mode QEMU provides two modes for full-system emulation, Kernel-based Virtual Machine (KVM) and Tiny Code Generator (TCG) modes. QEMU-KVM uses CPU support for hardware virtualization via Linux's KVM virtualization subsystem [23], which allows it to run the compiled OS binary with near bare-metal speed. QEMU-TCG, in comparison, translates the binary to an intermediate representation (IR), and then translates the IR to machine code for the host architecture for execution. Although TCG mode is slower than hardware, TCG mode makes taint analysis [15] and symbolic execution [36, 54] possible.

LLVM Mode Chipounov et al. [7] introduced a technique for dynamically translating the TCG IR to LLVM IR so that existing LLVM analysis techniques can be used with QEMU. This work reduces the development overhead for creating whole-system dynamic binary analyses, but it introduces a large performance penalty that we discuss shortly.

PANDA PANDA (Platform for Architecture-Neutral Dynamic Analysis) [15] is a record-and-replay tool built upon QEMU's TCG and LLVM modes. PANDA inserts hooks in full-system emulation to record essential inputs for a later deterministic replay. The record-and-replay feature allows users to record an execution in reasonably-fast TCG mode and perform heavier analyses at replay time. This is important when analyzing driver code, as many drivers have timeouts and watchdog threads that may be triggered if execution speed is too slow.

PANDA also supports taint analysis in a plugin system. A user can mark input bytes as tainted and observe taint flow and the branch instructions that depend on tainted data. It first translates TCG-IR to LLVM-IR and instruments the LLVM to add taint tracking, and then compiles and runs the LLVM code. The process introduces about 20x overhead compared to TCG mode: a ten-second recording can take about one minute in TCG mode replay and twenty minutes in LLVM-mode replay. When used in a fuzzer, this overhead can impede progress. In section 4.3, we discuss our optimizations to run LLVM mode selectively to improve speed.

Concolic Execution The random mutation strategy in fuzzing may have trouble with hard-to-pass checks, such as magic values or checksum validation, in real-world programs. Concolic execution is a method that uses symbolic execution on a concrete input to generate new inputs that diverge from the concrete path. These neighboring paths can potentially lead the fuzzer to an unexplored section of code and help the fuzzer overcome these blocking branches [28].

2.3 Challenges

2.3.1 Hardware Diversity and Availability

Hardware diversity is the major drawback of previous work that has hardware in the loop [47, 50]. Charm [50] requires porting new device drivers to the host virtual machine and components to the Android system, a task which the authors estimate takes two to five person-days. Although Periscope [47] needs less human labor, porting its components to the tested Android device is still necessary.

More importantly, the hardware-in-the-loop approach is not scalable. Even if the driver source code is available, researchers need to purchase the hardware that corresponds to each driver, which severely limits the ability of researchers to conduct broad testing of many drivers. The reliance on physical hardware also bounds the speed at which testing can be performed, since each device can only run one test case at a time. And powerful server hardware with many cores cannot be effectively used for such testing, since the device under test serves as a bottleneck. For these reasons, we believe fuzzing with emulation is a better solution for securing the OS-peripheral boundary.

```
1 #define VNIC_RES_MAGIC 0x766E6963L
2 #define VNIC_RES_VERSION 0L
3 if (ioread32(&rh->magic) != VNIC_RES_MAGIC ||
4     ioread32(&rh->version) != VNIC_RES_VERSION) {
5     return -EINVAL;
6 }
7 return 0;
```

Listing 1: Magic value check in `snic`.

2.3.2 Hard-to-Fuzz Code Patterns in Drivers

The benefit of fuzzing without real devices is that we can easily scale up to running on multiple virtual machines. At the same time, the lack of real hardware means that we have little idea what the input to the driver *should* look like. When fuzzing user-space programs, this problem can be alleviated by building a corpus of valid inputs to use as initial seeds; however, the inputs to device drivers do not conform to any particular file format, so seed inputs are not available. As You et. al. [53] note, fuzzing performance degrades when valid inputs are not present.

Moreover, device drivers contain many constructs such as magic values and polling loops that are challenging for traditional fuzzers and symbolic execution engines, and drivers will typically abort as soon as they detect unusual responses from the device. An example of a magic value check in the Cisco snic driver is shown in Listing 1. To pass the check, the fuzzer needs to provide a 4-byte value equal to the magic value and another 4-byte value equal to the version id. In our experiments, we found that a state of the art greybox fuzzer, Agamotto, was unable to pass this condition and failed to initialize the device driver. Magic values are often masked or shifted before comparison, so even dictionary-based fuzzing may not help.

Polling loops are also common in driver code: the driver may repeatedly query some device registers in a loop and abort if an unexpected value is encountered. An example of such a loop can be seen in Listing 3 later in the paper, where the ath9k driver tests an MMIO register in a loop 256 times. Existing techniques may struggle with such constructs: fuzzing is unlikely to generate the correct value on each iteration, symbolic execution may encounter path explosion (as a new state is generated on each iteration of the loop), and concolic execution would need to generate 256 inputs one at a time in order to get past the loop.

3 Drifuzz Design

As shown in Figure 1, we design Drifuzz with three major parts: seed generation, fuzzing, and concolic execution. We first generate the golden seed (Section 3.2) that helps us reach deep driver code with the help of concolic execution and forced execution (Section 3.3). We then pass the golden

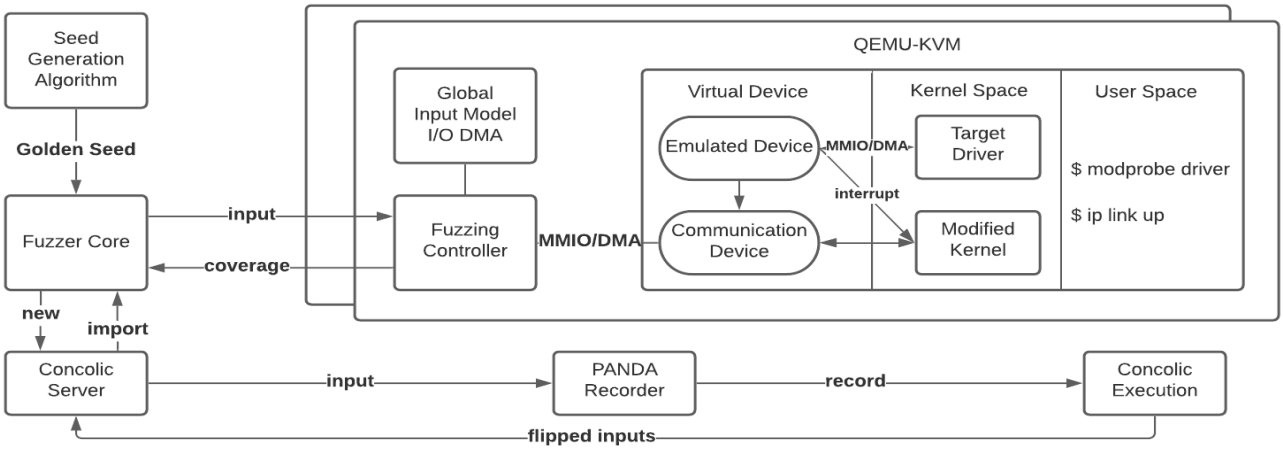

Figure 1: OS-peripheral boundary fuzzing with Drifuzz: our golden seed generation algorithm creates good initial inputs to overcome difficult checks during driver initialization. These seeds are then used as a starting point for a hypervisor-based concolic fuzzer that tests deeper code in the driver by repeatedly restoring a VM snapshot, initializing the driver, and feeding fuzzer-generated inputs via a virtual peripheral.

seed to our fuzzer, which sends mutated inputs to multiple KVM virtual machines. Each virtual machine (VM) handles MMIO/DMA reads with the content of the input seeds (Section 3.1); seeds are split up and reserved by I/O address to improve stability (Section 4.1). The virtual machine guest initializes the device driver and brings up the target network interface. After execution, the virtual machine reports the branch coverage bitmap to the fuzzer core. The fuzzer supports hybrid fuzzing using a concolic execution thread (Section 3.4).

3.1 Device Driver Inputs

MMIO and DMA are the two major sources of input for a device driver. Working on low level, PCI drivers usually use port I/O, MMIO and DMA directly. On the other hand, USB drivers work on higher layer with a universal protocol where data are transferred as packets.

MMIO is handled by QEMU's PCI device emulation interface. When we create an emulated device, we add memory-mapped regions that reflect the PCI memory layout on real hardware. Whenever the guest OS accesses these MMIO regions, QEMU queries our emulated device, which forwards MMIO reads to the fuzzer core. We ignore any writes to the emulated MMIO. Because QEMU implements the port I/O space as another memory region in its own address space, we also handle port I/O in this way.

DMA accesses require special handling. The Linux documentation defines two types of DMA buffers: consistent DMA and streaming DMA. Consistent DMA works synchronously: the driver and the device can read and write to the allocated space at any time and the result is visible to the other end immediately. Streaming DMA uses asynchronous communication, where the driver allocates a buffer and lets the device asynchronously access the buffer. When the device finishes the work, it notifies the driver through MMIO or an interrupt. The driver then deallocates the buffer and reads the data if needed.

We modify the kernel code to intercept DMA allocation and feed fuzzed input via the DMA buffer. We handle consistent and streaming DMA in different ways. Because consistent DMA is similar to MMIO, we register the memory region as an MMIO region with QEMU when allocating consistent DMA, and remove the region when it is deallocated. On the other hand, streaming DMA is used for transferring a larger amount of data asynchronously. Reading input happens after deallocation. Therefore, we fill the buffer with fuzzing input whenever deallocation occurs.

USB devices communicate data mostly via bulk transfers. At the lowest level, these data packets are transferred by the DMA controller and work similarly to streaming DMA. However, we leverage the USB layer of QEMU and handle them at the packet level, similar to USBFuzz [34].

3.2 Golden Seed Search

In Section 2.3.2, we noted that the key problem of fuzzing drivers without the corresponding hardware is the lack of a good initial seed: without such input, fuzzing mostly gets stuck early in driver initialization and cannot explore more complex device driver code. In this subsection, we introduce a search algorithm based on concolic execution to find such a "golden seed" for our fuzzer.

Many symbolic branches in driver code have preferred

conditions. Examples include an I/O flag that indicates the device is still alive, a check if there is new input, or a version ID check. Such branches must always resolve the same way if driver initialization is to succeed. If they do not, the driver will typically abort and prevent further exploration of the driver code; we call these **blocking branches**. Hence, we can find such branches using concolic execution and fix them to the preferred condition to reach deeper code.

We define a *preference* as the mapping from branch instructions to their preferred conditions ($Preference : Branch \rightarrow Condition$). A *blocking branch* is often indicated by an increase in coverage when flipped and there may be multiple blocking branches in one execution. Our algorithm greedily attempts to maximize a score based on the number of unique symbolic branches. Our golden seed search algorithm iteratively identifies the blocking branches while executing the seeds and improves the seed using a constraint solver to unlock key branches.

Listing 2 shows our algorithm in detail. Starting from an empty **preferences** map (line 2), we initialize the set of symbolic branches to consider by executing a random input and noting any branches that depend on input from the device and storing it in **new_branches**. The while loop (lines 6–27) then iteratively grows the preference map by attempting to find the preferred condition for each new branch. It does so by using *forced execution* (described in the next section) to run the input with the branch set to either true or false (line 13), which records all symbolic branches and produces a new input that induces the path. If this exposes new branches, **preferred_results** is updated so the branches can be considered in the next iteration. Branch conditions that are already present in the most recent execution are skipped (lines 11–12), since they are already satisfied by the current trace.

Once all branches in the current iteration have been evaluated, we pick the highest-scoring branch condition, save it in the **preferences** (lines 23–24), and select corresponding input and the branches it exposed to work on in the next iteration (lines 26–27). The loop terminates when no new branches are uncovered while testing branch conditions (lines 18–20), and designates the current input as the *golden seed*.

Empirically, we have found that the best metric for evaluating inputs is the number of *unique* symbolic branches they expose. Compared to block or branch coverage, the number of unique symbolic branches emphasizes how our inputs affect the execution path. If there is a tie, we prefer the input that results in fewer *total* symbolic branches (as shorter traces are more efficient for fuzzing).

```
1  def greedy_search(input):
2      preferences = {} # pc: cond
3      result = forced_execute(input, preferences)
4      new_branches = result.concolic_branches()
5
6      while True:
7          preferred_results = {}
8          for br in new_branches:
9              # Test for the preference condition
10             for c in [True, False]:
11                 if satisfy(result, {br, c}):
12                     continue
13                 test_result = forced_execute(input,
                       merge(preferences, {br: c}))
14                 if has_new_branch(test_result):
15                     preferred_results[(br, c)] =
                           test_result
16
17         # No new branches found.
18         if len(preferred_results) == 0:
19             print("The end.")
20             break
21
22         # Prepare for next iteration
23         br, cond, result =
24             select_best_preference(
                   preferred_results)
25         preferences = merge(preferences, {br:cond})
26         new_branches = new_branches(result)
27         input = result.output
28     golden_seed = input
```

Listing 2: Golden seed search algorithm

3.3 Forced Execution

Forced execution optimizes flipping repetitive concolic branches in device drivers by simply forcing a branch to go in the desired direction rather than attempting to solve the constraints. Conventional concolic execution can only flip one branch per execution; if a check is repetitive and has a preferred condition, this will result in many wasted executions and calls to the constraint solver. The code in Listing 3 shows a real-world example in the Atheros ath9k driver. The optimal path traverses the loop 256 (0x100) times with the condition on line 6 returning false each time, requiring 256 concolic runs.

To overcome this limitation of concolic execution, we use forced execution [31] to get the desired path and generate the correct input that will traverse that path, avoiding unnecessary branch flips. For the example in Listing 3, forced execution would allow us to traverse all 0x100 iterations of the loop with a single execution by setting the branch on line 6 to false. Concolic execution can then be used on this path to find the inputs that satisfy the branch condition in a single step, rather than having to solve each instance of the branch one at a time.

One pitfall of forced execution is that we cannot guarantee that the executions we generate correspond to any input: we may traverse paths that have conflicting conditions, resulting in an *infeasible path*. During golden seed search, we always try forced execution first, and retry with an input provided by the solver if the first execution fails due to an infeasible

```
int test_io() {
    for (u32 i = 0; i < 0x100; i++) {
        iowrite(OFFSET, i);
        delay(10);
        reg = ioread(OFFSET);
        if (reg != i)
            return -EIO;
    }
    return 0;
}
```

Listing 3: Atheros ath9k driver initialization test code snippet

path. If that also fails, we exclude the tested branches from consideration in the golden seed search. In Section 4.5 we describe in detail how we implement forced execution by modifying the generated TCG IR on the fly.

3.4 Traditional and Hybrid Fuzzing

Our design inherits kAFL's [41] traditional fuzzing design. The core fuzzer mutates and records inputs based on the coverage feedback. With the ability to run concolic execution, we are able to provide hybrid fuzzing as well. After seed generation, we use traditional fuzzing most of the time and invoke concolic execution to get over hard-to-pass branch conditions. When encountering a new path, the fuzzer core forwards the input to the concolic executor to generate inputs for neighboring paths. The new inputs are sent to the fuzzer core to test whether they result in new coverage.

4 Implementation

In this section, we discuss several relevant implementation details of our implementation. Overall, our implementation consists of 8,754 lines of new or modified code in C and Python; the total lines of code for each component are listed in Table 1. All code is released as open source to help future research and replication of our results.

Component	Lines
Linux Comm Driver and DMA Tracking	470 + 0
PANDA Concolic Support	842 + 77
PANDA Customization	2421 + 146
Fuzzing Backend (adapted from kAFL)	872 + 331
Fuzzing Scripts	874 + 0
Concolic Scripts	2721 + 0

Table 1: Drifuzz components and lines of code, as counted by `cloc`. We describe changes by added line + modified line.

4.1 Multi-buffer Input Feeding

Prior OS-peripheral boundary fuzzers [48] represent the fuzzer input as a single file, returning data from this file sequentially as the driver attempts to read data from the device. This compact sequential representation allows mutation strategies such as bit-flips and interesting bytes to work well, but may cause the same input to exhibit different behavior with concurrent drivers, as the kernel scheduler could run threads in a different order. This in turn could make it more difficult to reproduce test cases produced by the fuzzer and makes coverage measurements less stable. Concurrency is common in device drivers, which may register some tasks to run in background threads while interrupt handling occurs in another thread.

Instead, we store the fuzzer input as a collection of sequences separated according to their I/O address or DMA buffer size. The intuition is that different I/O addresses usually have different purposes and different threads usually work on different tasks, so this separation is more likely to provide the same behavior and coverage even if threads run in a different order. Our evaluation of this technique did not uncover major differences in bitmap coverage for the drivers we tested. However, because it does not add additional overhead and could still have benefits for drivers we have not yet tested, we leave it enabled.

4.2 KASAN Optimization

A major factor in the effectiveness of a fuzzer is the speed at which it can test a single input. While evaluating prior work [48], we found that virtual machine execution speed is much slower than native speed for tasks such as running the Linux `modprobe` command, which loads and initializes a device driver. Although the command can finish in milliseconds natively, it takes more than one second for some drivers inside the virtual machine.

Analyzing the performance of the execution, we found that the overhead is caused by stack walking in Kernel Address Sanitizer (KASAN). By default, KASAN records each memory (de)allocation and its call stack; although this information is not used in *detecting* memory errors, it allows KASAN to produce stack traces for the (de)allocation site, which can be helpful for debugging or for deduplicating crashes. However, during fuzzing, this cost is borne on every process and thread, and kernel module loading in particular involves many allocations (e.g., for the module code and all of its data structures).

We find that disabling this stack walk in KASAN results in a 4× speedup on average. Although this produces slightly less informative bug reports, we can easily reconstruct the missing information and generate a complete report after the fact by replaying interesting inputs with unmodified KASAN. In Appendix A we provide further details on the performance impact and causes of stack unwinding overhead and evaluate

alternative solutions.

4.3 Selective Symbolic Execution

As discussed earlier, PANDA is built for offline analyses, so it trades off speed for precision. However, high throughput is critical for effective bug-finding in fuzzing. Inspired by previous works [8, 9], we apply *selective* symbolic execution and only run device driver code in LLVM mode. In the absence of KASLR (which we disable), loadable device drivers are located starting at address 0xffffffffa0000000 on a 64-bit x86 system. We modify PANDA to only use LLVM translation if the code address is above this address.

We also enable LLVM for three additional pieces of kernel code that are needed for analysis: **I/O** functions, **memcpy**, and **context switch** functions. **I/O** functions (such as `ioread32`) are essential for us to register symbolic or taint values. The **memcpy** function can be called from a kernel module but resides in the code of the main kernel. And **context switch** functions are important for the correctness of the symbolic or taint tracking. When driver code is interrupted, the kernel saves the current registers on the stack. When the driver code gets to run again, the scheduler first restores the registers. To properly preserve symbolic labels in registers, we force context switch functions to run in LLVM mode.

By only running relevant driver code in LLVM mode, we can speed up the analysis by up to 20×. With this optimization, the overhead of running symbolic execution is almost identical to the TCG-mode overhead. The reason is that there are typically many kernel threads running in the background, and so driver code is a relatively small fraction of the total execution.

4.4 Concolic Execution

Similar to Siewertsen [46] (though independently implemented), we build our concolic execution on PANDA's existing taint analysis plugin. The design gives us many advantages. The record-and-replay system can let us run the time-sensitive driver code in fairly fast TCG mode, and perform analysis in compute-heavy LLVM mode. We need full-system emulation to attach emulated devices, and PANDA can already handle such use case. Lastly, PANDA is built upon QEMU and retains QEMU-KVM support; although we can neither record nor replay under KVM mode, using PANDA directly reduces redundant code. For constraint solving we use the Microsoft's Z3 [11] as it is fast, widely used, and robust.

PANDA's taint system translates TCG IR to LLVM IR, instrumenting it with functions that provide fine-grained (byte-level) taint tracking. Depending on the LLVM IR instruction type, taint can be propagated or removed. For example, multiplication mixes the taint of the input registers, while a store instruction replaces taint on the destination with the taint labels from the source.

To implement concolic execution, we extend the PANDA taint system's shadow memory to hold pointers to Z3 expressions, and then implement symbolic rules for each LLVM IR operation. For each IR instruction type, we apply the equivalent symbolic expression and store the result in the shadow memory. For example, in LLVM IR `%result = %reg1 + %reg2`, if `%reg1` has a symbolic value x and `%reg2` has a concrete value 7, `%result` will hold the symbolic value $x+7$.

To mark the initial symbolic input, we wrote a plugin that intercepts MMIO and DMA reads and creates byte-level symbolic labels. We can then collect symbolic branch conditions during execution; the full path condition is simply the conjunction of these constraints. For traces generated during forced execution, we can use the path condition as-is to obtain an input from the solver that would produce the trace in the absence of forced execution (if one exists). During hybrid fuzzing, we can use the path condition to invert a particular branch by truncating the conjunction of constraints after the term corresponding to that branch, negating that term, and passing the resulting formula to the Z3 solver to obtain a new input that causes the branch to go the opposite direction.

To handle symbolic pointers, Drifuzz concretizes the symbolic address using the concrete value observed at runtime. This is a common design choice shared with other implementations of concolic execution [19, 43, 45], but can limit our ability to handle driver code that produces complex constraints involving symbolic pointers. This could be improved by implementing symbolic pointer reasoning as found in systems such as Mayhem [5].

4.5 Forced Execution via TCG Modification

To achieve branch modification while still capturing useful path constraints, we hook into QEMU's TCG generation module. During seed generation, we modify the target branch to always go to the desired condition (either always true or always false). At the same time, we track the original branch condition so that we can collect the corresponding path constraints.

When the concolic executor sees a modified branch, it adds a path constraint that corresponds to the desired condition. At the end of concolic execution, when we have collected all path constraints, we use Z3 to solve the constraints and find a real input for this path.

In the ath9k example (Listing 3), we can quickly test the always-true or always-false condition for the branch. If the condition is always true, the test I/O function quickly fails and returns an error code to the caller, which aborts initialization. On the other hand, 256 iterations of the loop can finish in a single execution when the branch is always false, allowing the core to continue execution and find unseen symbolic branches. The new branches give the always-false condition a favorable

```
1  int ath10k_wait_for_target_init() {
2      for (int i = 0; i < 0x100; i++) {
3          reg = ioread(); // concrete: 0
4          if (reg & FLAG_INIT) // forced true
5              break;
6          delay();
7      }
8      if (reg & FLAG_INIT) // not forced
9          return 0; // success;
10     return -ETIMEOUT; // error;
11 }
```

Listing 4: Infeasible path

```
1  #!/bin/bash
2  target=$1
3  # Intilize the driver
4  modprobe $target # Return if failed
5  # Wait for async tasks
6  sleep 1
7  # Bring up network interface
8  if ip link | grep "eth0"; then
9    ip link set dev eth0 up
10 elif ip link | grep "wlan0"; then
11   ip link set dev wlan0 up
12 fi
```

Listing 5: Scripts run by guest system

score and we can proceed to find and solve the remaining blocking branches.

Infeasible Paths Although forced execution works well in practice, it can sometimes lead to infeasible paths (i.e., there is no actual input that corresponds to the path). When there are two symbolic branches that depend on one another, fixing the outcome of one branch can result in an infeasible path. The underlying cause is that during forced execution variables involved in the modified branch condition retain their original concrete values. If a later branch depends on the same variable, the branch outcome will be consistent with the concrete value, producing a pair of contradictory constraints and an unsatisfiable path condition. For example, in Listing 4, forcing line 4 to true without changing the condition at line 8 might cause a timeout error to be returned at line 10.

Due to the design of our TCG modification, we cannot force the latter branch on the fly. TCG modification occurs during execution, and it must be consistent throughout the record-and-replay phase. Since we are unaware of symbolic values at record time, we cannot modify the path to enforce consistency with the earlier forced branch.

However, we can detect this condition: in an infeasible path, the path constraints from the two conditions are contradictory and Z3 will return `unsat`. When such an inconsistency is detected, we remove the later path constraint and generate a new input that satisfies the first path. We then repeat concolic execution with newly generated input. Because we actually change the concrete value, this allows the second branch to be satisfied as well.

Although this works for most cases of infeasible paths, there are more complicated cases (e.g., with more than two dependent conditions) where this technique fails. In this case we simply ignore the preference for that particular branch when generating the golden seed.

4.6 Fuzzer Implementation

We build our fuzzer on top of kAFL [41] as it is a mature fuzzer that works with QEMU-KVM instances. This allows us to take advantage of fast hardware virtualization when testing new inputs and collecting coverage.

The fuzzer starts up multiple QEMU-KVM virtual machine instances in parallel. For PCI targets, each QEMU instance has an emulated peripheral configured with the PCI device IDs[2] of the target device to send fuzzing input. For USB targets, we attach an emulated USB peripheral with a USB descriptor containing the correct device identifier for the target driver; this identifier can be obtained from the USB_DEVICE declaration in the driver's code. We attach another virtual device to the VM for communication with the fuzzing backend. The VM also runs a modified guest Linux kernel that exposes DMA mapping.

Each instance starts with a fresh environment by loading a VM snapshot. It then runs a script inside the guest VM (shown in Listing 5) that uses `modprobe` to load the driver, waits for one second to let any background setup finish, and then attempts to bring up the network interface so it is able to receive or transmit packets. Our customized kernel detects failures during initialization that prevent the module from loading and will abort the run early to avoid wasting time on unsuccessful runs.

When the script finishes, it reports coverage and restores the VM to the saved snapshot. If the instance times out or has a memory error, we record the input and restart the virtual machine. For fuzzing PCI drivers, we also trigger an interrupt for the device every 75 MMIO reads/writes. This allows the interrupt handling code to be explored while preserving determinism and reproducibility.

Our fuzzer uses the golden seed generated according to the algorithm in Section 3.2 for its initial input and creates test cases for each execution that reaches new coverage. Both the initial seed and the test cases are stored in a format that contains the binary file with actual values to return as well as the mapping of MMIO addresses and DMA read sizes to the values (as described in Section 4.1) for PCI devices. Because input to USB devices is handled at the higher-level packet interface, only the binary file is required.

[2]These IDs are readily available for each driver in the kernel's `modules.alias` file.

5 Evaluation

In this section, we evaluate the efficiency and effectiveness of Drifuzz on a total of ten real-world Ethernet and WiFi drivers, including both PCI and USB targets. We are interested in the following questions:

RQ1. **Evaluating Drifuzz**: How efficient and effective is Drifuzz's golden seed search? What are the individual contributions of the golden seeds and hybrid fuzzing components at achieving code coverage in drivers, compared to a baseline coverage-guided fuzzer?

RQ2. **Comparison with prior work**: How does Drifuzz compare to prior work on device driver testing that used symbolic execution (SymDrive [38]) or coverage-guided greybox fuzzing (Agamotto [48])?

RQ3. **Bug-finding**: Can Drifuzz find previously unknown bugs in Linux network drivers?

5.1 Experimental Setup

Our evaluation is performed on two separate systems: a server with 2x AMD EPYC 7542 32-core CPUs and 512 GiB of RAM (referred to as just the "server machine" throughout this section), and a desktop with a 16-core AMD Ryzen 5950x CPU and 32 GiB of RAM (the "desktop machine"). We use the former for our larger fuzzing experiments, and the latter for smaller experiments and those that require a specialized kernel (e.g., for comparisons with Agamotto) or other extensive local modifications (such as the build and runtime environment for SymDrive).

5.2 Evaluating Drifuzz

5.2.1 Golden Seed Generation

In this experiment, we run our golden seed generation algorithm on the desktop machine with ten target drivers: three PCI WiFi, four PCI Ethernet and three USB WiFi. In Table 2, we list the mean time needed to generate the seed across three trials, the total number of symbolic branches discovered, and the number of blocking branches that are discovered and solved. Ten of our twelve previously unknown bugs were discovered during seed generation.

The time required for seed generation varies depending on the complexity of the driver, from a few minutes for ris_usb and mwifiex_usb up to 138 minutes for the ath9k driver. Seeds are intended to be generated once and then used for longer fuzzing campaigns, so we feel that these times are reasonable. The number of symbolic branches found indicate that it is capable of reaching deeper paths in driver code, particularly for PCI drivers. Our results also show that blocking branches are relatively common and present in all tested drivers.

Driver	Time	Sym Branch	Block	Bugs
ath9k	138m	118	7	0
ath10k_pci	25m	20	4	1
rtwpci	76m	22	6	0
8139cp	40m	19	2	0
atlantic	16m	20	1	2
snic	14m	7	4	0
stmmac_pci	75m	54	3	1
ar5523	57m	7	1	1
mwifiex_usb	2m	15	2	1
rsi_usb	3m	8	2	2

Table 2: Golden seed search statistics: runtime, number of unique symbolic branches discovered, number of fixed blocking branches and number of bugs we found and fixed.

5.2.2 Ablation Study

To better understand the contribution of the seed generation and concolic fuzzing components to Drifuzz's ability to cover code in complex drivers, we conduct an *ablation* study in which we compare the coverage achieved in seven PCI network drivers (three WiFi and four Ethernet) under different configurations: greybox fuzzing with a random seed (RandomSeed, our baseline), concolic fuzzing with random seed (RS+C), greybox fuzzing with the generated golden seed (GoldenSeed) and concolic fuzzing with golden seed (GS+C, our full system).

The WiFi drivers tested are rtwpci, ath9k, and ath10k, and the Ethernet drivers are stmmac_pci, 8139cp, atlantic, and snic. The WiFi drivers are chosen because they are complex SoftMAC drivers which implement most of the 802.11 functionality in software rather than hardware, while the Ethernet drivers were picked to allow comparison with prior work [48].

Our experiments run in parallel with 64 QEMU instances, utilizing all physical cores. For each configuration, we run a fuzzing experiment for 1 hour, for a total of 64 core-hours. When running concolic execution, we pause a QEMU instance to ensure all settings use the same amount of CPU time. Following the fuzzing recommendations of Klees et al. [24], we repeat each experiment ten times; we also provide the coverage increase of the full system (GS+C) compared to the baseline and assess statistical significance using the two-sided Mann-Whitney U test. To measure coverage we count the number of non-zero bytes in the coverage map (this is equivalent to the number of unique edges discovered, modulo hash collisions). The results are summarized in Table 3.

We find that compared to the baseline, the full Drifuzz configuration can boost coverage by 214% in WiFi PCI drivers and 60% in Ethernet PCI drivers, on average. The results are statistically significant for all drivers except stmmac_pci (although the gains for the 8139cp driver are quite small). Through manual inspection, we find that stmmac_pci and 8139cp have relatively simple initialization functions, so stan-

Driver	RandomSeed	RS+C	GoldenSeed	GS+C	Increase	Signif
ath9k	310.9	522.9	2070.9	2793.7	798.6%	***
ath10k_pci	462.8	657.2	785.6	793.4	71.4%	***
rtwpci	183.1	163.6	384.1	386	110.8%	***
8139cp	173.1	172.4	173.3	173.7	0.3%	*
atlantic	372.1	1441.9	1033.7	1532.5	311.9%	***
stmmac_pci	798.9	749.5	818.5	812.9	1.8%	n.s.
snic	54	81.7	83	83.7	55.0%	****

Table 3: Mean bitmap byte coverage when fuzzing PCI network drivers across 10 trials with coverage increase between the baseline (RandomSeed) and our full system (GS+C). RS: random seed; GS: golden seed; +C: concolic-assisted. Asterisks indicate the significance level as measured by the Mann-Whitney U test: *: p<0.05, **: p<0.01, ***: p<0.001, and ****: p<0.0001.

dard greybox fuzzing suffices on these drivers.

The configurations that use golden seeds tend to outperform those that start with random seeds. This is true even when we do not employ concolic execution during fuzzing. We believe that this may indicate that many of the more difficult conditions in driver code are found during the initialization phase, so configurations using golden seeds capture most of the benefits of concolic execution during seed generation. We do not find statistically significant differences between the two golden seed configurations, providing further evidence that concolic fuzzing does not add much additional benefit over golden seeds alone.

5.3 Comparison with Prior Work

5.3.1 SymDrive

In Section 2.3.2, we noted that purely symbolic approaches may suffer from path explosion when faced with code constructs found in complex device drivers, leading us to employ a combination of concolic and forced execution in the design of our golden seed generation algorithm. To test this, we compare Drifuzz's golden seed generation to a previous symbolic-execution based system for testing device drivers, SymDrive [38].

Because SymDrive is relatively old (it was published in 2012) and is implemented for Linux 3.1.1, we backport Drifuzz to this kernel version and then use both systems to test four PCI WiFi drivers present in this version of the kernel: ath5k, ath9k, atmel_pci, and orinoco_pci. Because Linux 3.1.1 does not support in-kernel coverage measurement via `kcov`, we instead compare the time spent by each system to complete its exploration, whether the network interface (eth0 or wlan0) is successfully initialized once each system's search terminates, and the number of bugs found. As the SymDrive paper, we perform three trials and report the mean time. The result is shown in Table 4: interface discovery and bugs found results are consistent in all three trials.

SymDrive finishes faster than Drifuzz for three out of the four drivers, but encounters path explosion in the orinoco driver and takes 420 minutes (7 hours) to complete. However, SymDrive only successfully initializes the network interface in one of the four drivers, ath9k, while Drifuzz is able to initialize the interface in all four. SymDrive's symbolic exploration is based on a prioritization strategy that favors successful return values (zero, in most parts of the Linux kernel). It is possible that implementing a different search strategy could allow SymDrive to succeed on the other three drivers, but this is outside the scope of our work.

Driver	SymDrive	Intf	Drifuzz	Intf	Bugs
ath5k	13s	×	65m	✓	1
ath9k	193s	✓	138m	✓	×
atmel_pci	2s	×	29m	✓	×
orinoco_pci	~420m	×	64m	✓	1

Table 4: Comparison between SymDrive and Drifuzz's concolic search; we perform three trials and report the mean. Intf indicates whether the network interface is found. The Bugs column shows bugs Drifuzz discovered in Linux v3.1.1 that SymDrive does not find.

Finally, whereas SymDrive's exploration does not uncover any bugs in the tested drivers, Drifuzz found memory safety issues in the ath5k and orinoco_pci drivers. We find that the orinico bug has been independently discovered and patched in later kernel versions; however, the ath5k bug was still present in modern kernels. We reported the flaw and provided a fix to the kernel maintainers.

5.3.2 Agamotto

Agamotto is the most closely related work to our own, as it applies greybox fuzzing at the OS-peripheral boundary. Due to hardware compatibility issues, we perform our comparison on the desktop rather than the server machine. As before, we run the experiment for one hour, but due to the lower number of CPU cores on the desktop machine this gives a 16 core-hour experiment; we repeat the experiment ten times and report significance. We add support for the three PCI WiFi drivers tested in Section 5.2.2 to Agamotto to make the

Driver	Agamotto	Drifuzz	Increase	Signif
ath9k	503.4	2782.5	452.7%	***
ath10k_pci	412.9	889.9	115.5%	***
rtwpci	163	394.2	141.8%	***
8139cp	105.7	171.8	62.5%	****
atlantic	265.8	841	216.4%	***
stmmac_pci	742.9	914.8	23.1%	***
snic	51	86.1	68.7%	****

Table 5: Mean bitmap byte coverage from 10 trials for Agamotto and Drifuzz with coverage increase and statistical significance: *: p<0.05, **: p<0.01,***: p<0.001 and ****: p<0.0001).

Driver	Agamotto	Drifuzz	Bug	Signif
ar5523	47	60.7	1	****
mwifiex	66	126.7	1	****
rsi	76	217.3	2	****

Table 6: Mean block coverage for USB targets from 10 trials, Agamotto vs Drifuzz, the number of newly discovered bugs by Drifuzz, and statistical significance: *: p<0.05, **: p<0.01, ***: p<0.001 and ****: p<0.0001). GS: golden seed byte coverage.

evaluation more consistent with our ablation study, and test four PCI Ethernet and three USB WiFi drivers (included in Agamotto's evaluation).

We also made minor modifications to Agamotto to make the configuration more consistent and comparable with Drifuzz. Agamotto's coverage measurement uses a shared global variable for the previous program counter, which leads to spurious edges added to the coverage map in the presence of multiple threads; we convert it to a thread-local variable. Agamotto also measures coverage in the entire network subsystem; because Drifuzz is optimized for driver fuzzing (in particular, its concolic analysis is disabled outside of driver code); we limit Agamotto's coverage tracking to driver code as well.

PCI Drivers We compare Drifuzz with Agamotto on PCI-based drivers in Table 5. We find that Drifuzz achieves, on average, 154% (2.5×) higher coverage across PCI drivers. These results are significant at the p<0.001 level or better for all tested drivers. Despite a shorter experiment, the coverage is generally on par with the results of our ablation study; we believe this is due to the significantly faster single-core performance on the desktop system.

USB Drivers Here, we evaluate Agamotto and Drifuzz on the three USB WiFi drivers that were included in Agamotto's evaluation. For consistency with Drifuzz, we omit the USB disconnect option when testing Agamotto and compare coverage achieved within the driver. Note that unlike the other experiments in our evaluation, we measure *block* coverage rather than *edge* coverage because Agamotto (and Syzkaller, on which Agamotto's USB support is based) do not report branch coverage when fuzzing USB devices. The results are shown in Table 6.

Drifuzz outperforms Agamotto on every driver, finding four previously unknown bugs that Agamotto fails to detect. We use the interactive coverage UI (inherited by Syzkaller) and confirm that the buggy code is not covered by Agamotto in any of the three drivers.

5.4 Bug Finding

In addition to the bugs discovered in the course of our main evaluation, we also conducted an *ad hoc* test of the ath9k USB driver with the golden seed generation algorithm. We were able to find two bugs in this driver using Drifuzz: one bug which was already reported by Syzkaller, and a previously unknown bug that was uncovered by Drifuzz after applying a patch to fix the Syzkaller-reported flaw.

Overall, we have used Drifuzz to find twelve previously unknown bugs in four PCI drivers and four USB drivers. We discovered two other PCI driver bugs manually during the development of our fuzzer. We have submitted patches to the Linux open-source community for fourteen discovered bugs; thirteen of these patches have been accepted into the kernel and one is still under review.

Table 7 lists the twelve memory bugs we discovered with Drifuzz. Ten were found during the concolic exploration phase of golden seed generation and two were found during hybrid fuzzing.

5.4.1 Vulnerability Disclosure

We reported all bugs found to the Linux kernel developers, and provided patches to fix the issues. We additionally evaluated the severity of the issues, applied for CVE identifiers for the two we felt were likely to be exploitable, and were assigned CVE-2021-XXXXX5 and CVE-2021-XXXXX6. CVE-2021-XXXXX5 is an out-of-bounds read followed by an out-of-bound write with attacker-controlled length in the atlantic PCI Ethernet driver, and may be exploitable locally by an attacker with the ability to attach a malicious PCI device, or by first exploiting a flaw in the atlantic firmware and then using this vantage point to attack the host. CVE-2021-XXXXX6 is a kernel panic (denial of service) in the Marvell mwifiex USB driver and can be triggered by an attacker who can insert a malicious USB device or compromise the mwifiex firmware.

Aside from disclosing and helping to fix the vulnerabilities we discovered, we also worked with a downstream vendor to help them understand the potential impact on their distribution.

Summary	Driver	Type	Fixed	Stage
KASAN: slab-out-of-bounds in ath10k_pci_hif_exchange_bmi_msg	ath10k	PCI	✓	seed-gen
KASAN: slab-out-of-bounds in hw_atl_utils_fw_upload_dwords	atlantic	PCI	✓	fuzzing
KASAN: double-free or invalid-free in consume_skb	atlantic	PCI	✓	seed-gen
KASAN: use-after-free in stmmac_napi_poll_rx	stmmac	PCI	✓	seed-gen
KASAN: use-after-free in aq_ring_rx_clean	atlantic	PCI	✓	seed-gen
KASAN: slab-out-of-bounds in ath5k_eeprom_read_pcal_info_5111	ath5k	PCI	✓	seed-gen
KASAN: null-ptr-deref	ar5523	USB	✓	seed-gen
skbuff: skb_over_panic	mwifiex	USB	✓	seed-gen
KASAN: slab-out-of-bounds in ath9k_hif_usb_rx_cb	ath9k_htc	USB	✓	seed-gen
KASAN: slab-out-of-bounds in rsi_read_pkt	rsi	USB	✓	seed-gen
KASAN: use-after-free in rsi_rx_done_handler	rsi	USB	✓	seed-gen
KASAN: use-after-free in rsi_read_pkt	rsi	USB		fuzzing

Table 7: Summary of new memory/panic bugs we found, the name of the buggy device driver, bus type, whether fixed upstream and the stage we found the bug

6 Limitations and Future Work

Similar to many other systems that use symbolic execution [19, 43, 45], we have a limitation in our handling of pointers with symbolic addresses. Ideally, a read or write at a symbolic address should consider that all possible addresses that satisfy the constraints could be modified; however, this can be very expensive in practice. We instead adopt a common practical workaround and simply concretize the pointer. As a consequence, however, we cannot solve complex constructs involving symbolic addresses, such as parsing the contents of non-volatile read-only memory (NVRAM) in the rtwpci driver. We hope to remove this limitation in future work by adopting some form of symbolic array modeling, similar to that found in Mayhem [5].

As we discuss in our related work, the nascent field of *embedded device rehosting* [16, 51], which attempts to automatically model device peripherals so that the embedded device firmware can be emulated entirely in software, shares many key goals with our work. Both rehosting and hardware-free driver fuzzing need to generate satisfactory responses to driver queries, lest the driver conclude that the hardware is malfunctioning and abort. Although the level of driver functionality needed to *fuzz* a driver is significantly less than that required to emulate a whole embedded system, we believe that some of our techniques for finding good values for peripheral responses while avoiding path explosion may be applicable to the rehosting problem as well, and we aim to explore this connection further in future research.

We hope to extend our work to closed-source Android device drivers. Because these devices are mobile, they are more exposed to remote attacks via WiFi and Bluetooth; at the same time, many of the drivers are closed source and proprietary, which makes large-scale testing difficult. Our current workflow requires the driver source only for coverage collection; however, recent advances in both hardware-assisted coverage collection [40, 41] and static binary rewriting [14, 18, 55] may make coverage collection on closed-source drivers easier. Finally, we also hope to extend our fuzzer to explore different interrupt schedules in order to expose harder-to-find driver bugs such as race conditions.

7 Related Work

Fuzzing has recently received a great deal of attention from academic researchers. Here we focus on the work most closely related to Drifuzz; for a more complete overview of recent fuzzing research we direct the reader to the survey of Manès et al. [29]. Drifuzz extends traditional fuzzing by using forced execution and concolic testing to generate good initial seeds for complex driver code, allowing the OS-peripheral boundary to be efficiently tested without requiring actual hardware.

Hybrid Fuzzing Hybrid fuzzing is a popular approach to overcome some of the limitations of random fuzzing; however, the majority of this work focuses on userspace programs [6, 19, 21, 35, 49, 54]. HFL [21] works with the Linux kernel but mainly examines the system call interface and does not handle MMIO, DMA, or interrupts.

USB Fuzzing Although PCI device drivers have so far received relatively little attention, some prior work has examined the security of USB drivers [25, 34]. Here, the motivating threat model is an attacker who has physical access to a machine and can insert a malicious USB peripheral.

Program Transformation Prior work uses program transformation in fuzzing [35] and malware detection [22]. T-Fuzz [35] applies forced execution to disable complex condition checks and later recovers the path through symbolic execution. Although the basic idea is similar, Drifuzz works with kernel code and modifies the driver code during concolic execution to quickly identify preferred branch conditions.

Checkpoint-based Fuzzing Agamotto [48] uses snapshots to accelerate device driver fuzz testing. Finding new paths is a rare occasion in fuzzing; most of the fuzzer's time is spent in already-found paths. Agamotto actively creates snapshots during execution. When it executes a new input, it finds the closest snapshot by longest common prefix, and starts execution from the snapshot. The time difference between real execution and restoration is saved and thus speeds up fuzzing. Although our evaluation finds that Drifuzz's seed generation is better able to handle complicated drivers and reach deeper paths, the snapshot mechanism could also potentially benefit Drifuzz if the necessary symbolic state were saved in the snapshot.

Symbolic Execution SymDrive [38] and DDT [26] are earlier works that use symbolic execution to test device drivers. However, as we see in our comparison with SymDrive (Section 5.3.1), these symbolic approaches may struggle with complex code such as WiFi drivers due to path explosion.

Hardware-based Device Driver Testing Hardware-in-the-loop testing can be an effective bug-finding strategy [32, 39, 47, 50]. Unfortunately, this technique requires significant human effort and resources to test a new device. For example, Periscope [47] needs to flash a custom Android kernel to the device under test and Charm [50] requires porting the device driver to a modified kernel. One recent work, BOSD [27], uses record and replay to scale fuzzing of GPU drivers by replaying recorded responses from the real hardware on multiple cores, but at least one real device is still needed, and it focuses only on the system call boundary.

Firmware Rehosting On the embedded side, there are firmware testing tools that apply fuzzing [17, 30] and symbolic execution [4, 20, 56]. These tools mainly work with IoT devices that run custom firmware on the ARM architecture. P2IM [17] and DICE [30] use heuristics tuned for microcontrollers to categorize the type of input registers, such as control registers, status registers and data registers.

In concurrent work, μemu [56] and Jetset [20] use symbolic execution in an attempt to create higher-fidelity models of embedded peripherals; while μemu targets relatively simple microcontrollers, Jetset can partially emulate more complex embedded systems such as a Raspberry Pi. However, Jetset currently only models relatively simple devices such as GPIO and UART interfaces. Based on our evaluation of SymDrive, we expect that both systems would struggle with emulating more complex peripherals such as the SoftMAC WiFi devices whose drivers we test; Drifuzz's use of forced execution allows it to more efficiently generate seeds for these drivers that satisfy the drivers' initialization checks and unlock deeper code paths. As discussed in Section 6, however, we believe our techniques may also help enhance rehosting efforts.

8 Conclusion

In this paper we presented a technique for efficiently generating "golden seeds" that allow the OS-peripheral boundary to be tested efficiently without access to real hardware peripherals. Our implementation augments PANDA's existing taint analysis to perform concolic execution, and leverages TCG modification to optimize concolic golden seed generation. Our evaluation of fourteen WiFi and Ethernet drivers shows that golden seeds and hybrid fuzzing allow Drifuzz to achieve higher coverage than the previous state of the art and uncover real vulnerabilities in device drivers. Two bugs with severe impact were assigned CVEs.

Availability

To facilitate future research, we will open-source Drifuzz. An anonymized version is available at:
https://github.com/messlabnyu/DrifuzzProject/

Acknowledgements

We would like to thank our reviewers for their feedback, and in particular our shepherd, Le Guan. The work in this paper is supported by funding from the National Science Foundation under Grant No. CNS-2001161 and CNS-2145482. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46. Califor-nia, USA, 2005.

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

[3] Laurent Butti. Wi-Fi advanced fuzzing. BlackHat EU 2007, 2007.

[4] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation. In *Annual Computer Security Applications Conference*, pages 746–759, 2020.

[5] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394, 2012.

[6] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1580–1596. IEEE, 2020.

[7] Vitaly Chipounov and George Candea. Dynamically translating x86 to LLVM using QEMU. Technical report, École Polytechnique Fédérale de Lausanne (EPFL), 2010.

[8] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective symbolic execution. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*, 2009.

[9] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. *Acm Sigplan Notices*, 46(3):265–278, 2011.

[10] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. DIFUZE: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2123–2138, New York, NY, USA, 2017. Association for Computing Machinery.

[11] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[12] Google Developers. Syzkaller. https://github.com/google/syzkaller.

[13] QEMU Developers. Official QEMU mirror. https://github.com/qemu/qemu.

[14] S. Dinesh, N. Burow, D. Xu, and M. Payer. RetroWrite: Statically instrumenting COTS binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1497–1511, Los Alamitos, CA, USA, may 2020. IEEE Computer Society.

[15] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, pages 1–11, 2015.

[16] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Oleinik, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, Davide Balzarotti, and William Robertson. SoK: Enabling security analyses of embedded systems via rehosting. In *16th ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, June 2021.

[17] Bo Feng, Alejandro Mera, and Long Lu. P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1237–1254, 2020.

[18] Antonio Flores-Montoya and Eric Schulte. Datalog disassembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1075–1092. USENIX Association, August 2020.

[19] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[20] Evan Johnson, Maxwell Bland, YiFei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. Jetset: Targeted firmware rehosting for embedded systems. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 321–338. USENIX Association, August 2021.

[21] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. HFL: Hybrid fuzzing on the Linux kernel. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2020.

[22] Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. J-Force: Forced execution on Javascript. In *Proceedings of the 26th international conference on World Wide Web*, pages 897–906, 2017.

[23] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the Linux virtual machine monitor. In *Proceedings of the Linux symposium*. Dttawa, Dntorio, Canada, 2007.

[24] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing, 2018.

[25] Andrey Konovalov. External USB fuzzing for Linux kernel. https://github.com/google/syzkaller/blob/master/docs/linux/external_fuzzing_usb.md.

[26] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. Testing closed-source binary device drivers with DDT. In *USENIX Annual Technical Conference*, 2010.

[27] Dominik Maier and Fabian Toepfer. *BSOD: Binary-Only Scalable Fuzzing Of Device Drivers*, page 48–61. Association for Computing Machinery, New York, NY, USA, 2021.

[28] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE'07)*, pages 416–426. IEEE, 2007.

[29] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11), Nov 2021.

[30] Alejandro Mera, Bo Feng, Long Lu, Engin Kirda, and William Robertson. DICE: Automatic emulation of dma input channels for dynamic firmware analysis. *arXiv preprint arXiv:2007.01502*, 2020.

[31] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy*, 2007.

[32] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar2: A multi-target orchestration platform. In *NDSS Workshop on Binary Analysis Research*, volume 18, pages 1–11, 2018.

[33] Andy Nguyen. BleedingTooth: Linux Bluetooth zero-click remote code execution. https://google.github.io/security-research/pocs/linux/bleedingtooth/writeup.html.

[34] Hui Peng and Mathias Payer. USBFuzz: A framework for fuzzing USB drivers by device emulation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2559–2575, 2020.

[35] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[36] Sebastian Poeplau and Aurélien Francillon. SymQEMU: Compilation-based symbolic execution for binaries. In *NDSS*, 2021.

[37] Ivan Pustogarov, Qian Wu, and David Lie. Ex-vivo dynamic analysis framework for Android device drivers. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1088–1105. IEEE, 2020.

[38] Matthew J Renzelmann, Asim Kadav, and Michael M Swift. Symdrive: Testing drivers without devices. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 279–292, 2012.

[39] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced wireless fuzzing to exploit new Bluetooth escalation targets. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 19–36, 2020.

[40] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wör-ner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021.

[41] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-assisted feedback fuzzing for os kernels. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 167–182, USA, 2017. USENIX Association.

[42] Denis Selyanin. Researching Marvell Avastar Wi-Fi: from zero knowledge to over-the-air zero-touch RCE. ZeroNights, https://2018.zeronights.ru/wp-content/uploads/materials/19-Researching-Marvell-Avastar-Wi-Fi.pdf, 2018.

[43] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, sep 2005.

[44] Ben Seri, Gregory Vishnepolsky, and Dor Zusman. BLEEDINGBIT: The hidden attack surface within BLE chips. Technical report, ARMIS Inc., 2019.

[45] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[46] Eike Siewertsen. Multi-platform binary program testing using concolic execution. Master's thesis, Chalmers University of Technology, 2015.

[47] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. PeriScope: An effective probing and fuzzing framework for the hardware-os boundary. In *NDSS*, 2019.

[48] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent ByungHoon Kang, Jean-Pierre Seifert, and Michael Franz. Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2541–2557. USENIX Association, August 2020.

[49] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, 2016.

[50] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, page 291–307, USA, 2018. USENIX Association.

[51] Christopher Wright, William A. Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A. Clements. Challenges in firmware re-hosting, emulation, and analysis. *ACM Computing Surveys*, 54(1), January 2021.

[52] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 818–834. IEEE, 2019.

[53] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. SLF: fuzzing without valid seed inputs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 712–723. IEEE, 2019.

[54] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, 2018.

[55] Z. Zhang, W. You, G. Tao, Y. Aafer, X. Liu, and X. Zhang. STOCHFUZZ: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*, pages 1884–1901, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[56] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. Automatic firmware emulation through invalidity-guided knowledge inference. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2007–2024. USENIX Association, August 2021.

A KASAN Optimization

Here we provide more detailed performance results for frame unwinding. We use the test cases generated by our fuzzer to evaluate the performance of Drifuzz on the 7 PCI network drivers. The result, shown in Table 8, shows an average speedup of 4.72× after removing stack unwinding for memory allocation and deallocation, with speedups of nearly 16× in allocation-intensive drivers.

As an alternative to disabling KASAN's stack unwinding, we could also attempt to use a faster unwinder; we therefore evaluated the two unwinders supported by Linux kernel. The Frame Pointer unwinder (FP) stores a pointer to the previous frame in a register so it can more quickly unwind calling stacks. The ORC unwinder (enabled by default) uses a binary format to store information about stack frame sizes, allowing unwinding to occur without the use of a dedicated frame pointer register. Although this makes unwinding much slower, it improves performance during normal execution by freeing up an extra register and reducing code size by ≈3.2%. However, with KASAN enabled, stack traces are collected frequently during **kmalloc** and **kfree**, causing a slowdown of up to 20% in our tests.

To more precisely measure the overhead of allocation-related unwinding we created a micro-benchmark module that performs `kmalloc`/`kfree` 500,000 times (comparable to the number of allocations made during Ethernet driver initialization). We test four configurations: KASAN+FP, KASAN+ORC, KASAN with our patch, and, for comparison, ORC with KASAN disabled. Across ten trials, the benchmark takes 286.7ms, 351.2ms, 57.8ms and 8ms. Our patched version is considered the baseline, where the same code is run but no stack unwinding happens. The data show that FP unwinder is 1.2× faster than ORC unwinder, but our patch still provides a speedup of 4.9× over the FP unwinder.

Driver	Optimized	Unmodified	Speedup
ath10k_pci	0.32s	2.23s	6.96×
ath9k	1.45s	1.96s	1.35×
rtwpci	0.28s	2.12s	7.65×
atlantic	0.16s	2.07s	12.93×
8139cp	1.14s	3.14s	2.74×
stmmac_pci	1.51s	1.93s	1.28×
snic	0.13s	2.04s	15.95×
geomean			4.72×

Table 8: Average execution time in seconds per input with optimized and unmodified KASAN.