

USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation

Hui Peng, *Purdue University;* Mathias Payer, *EPFL*

https://www.usenix.org/conference/usenixsecurity20/presentation/peng

This paper is included in the Proceedings of the 29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Open access to the Proceedings of the 29th USENIX Security Symposium is sponsored by USENIX.

USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation

Hui Peng
Purdue University

Mathias Payer
EPFL

Abstract

The Universal Serial Bus (USB) connects external devices to a host. This interface exposes the OS kernels and device drivers to attacks by malicious devices. Unfortunately, kernels and drivers were developed under a security model that implicitly trusts connected devices. Drivers expect faulty hardware but not malicious attacks. Similarly, security testing drivers is challenging as input must cross the hardware/software barrier. Fuzzing, the most widely used bug finding technique, relies on providing random data to programs. However, fuzzing device drivers is challenging due to the difficulty in crossing the hardware/software barrier and providing random device data to the driver under test.

We present USBFuzz, a portable, flexible, and modular framework for fuzz testing USB drivers. At its core, USBFuzz uses a software-emulated USB device to provide random device data to drivers (when they perform IO operations). As the emulated USB device works at the device level, porting it to other platforms is straight-forward. Using the USBFuzz framework, we apply (i) coverage-guided fuzzing to a broad range of USB drivers in the Linux kernel; (ii) dumb fuzzing in FreeBSD, MacOS, and Windows through cross-pollination seeded by the Linux inputs; and (iii) focused fuzzing of a USB webcam driver. USBFuzz discovered a total of 26 new bugs, including 16 memory bugs of high security impact in various Linux subsystems (USB core, USB sound, and network), one bug in FreeBSD, three in MacOS (two resulting in an unplanned reboot and one freezing the system), and four in Windows 8 and Windows 10 (resulting in Blue Screens of Death), and one bug in the Linux USB host controller driver and another one in a USB camera driver. From the Linux bugs, we have fixed and upstreamed 11 bugs and received 10 CVEs.

1 Introduction

The Universal Serial Bus (USB) provides an easy-to-use interface to attach external devices to host computers. A broad set of features such as wide range of bandwidth support, Plug and Play, or power delivery has contributed to its widespread adoption. USB is ubiquitous; it is supported on commodity PCs, smart TVs, and mobile phones. Further, software technologies like USBIP [46] and usbredir [43] allow a USB device on one machine to be remotely connected to another.

The ubiquity and external accessibility result in a large attack surface that can be explored along different categories: (i) exhaustive privileges for USB devices [27, 41] (e.g., the famous "autorun" attack that allows USB storage devices to start programs as they are plugged in), (ii) electrical attacks leveraging physical design flaws [65], and (iii) exploiting software vulnerabilities in the host OS [29]. Attacks against exhaustive privileges can be solved by reconfiguring the operating system through customized defenses (e.g., disabling "autorun", GoodUSB [58], USBFilter [60], or USBGuard [45]) and hardware attacks can be protected through improved interface design. We focus exclusively on software vulnerabilities in the host OS as these issues are hard to find and have high security impact.

Analogous to userspace programs that read inputs from files, device drivers consume inputs from connected devices. Failure to handle unexpected input results in memory bugs like buffer-overflows, use-after-free, or double free errors—with disastrous consequences. As device drivers run directly in the kernel or privileged processes, driver bugs are security critical. Historically, because the hardware was trusted and considered hard to modify, little attention was paid to this attack surface. Unaware of the potential attacks, host side software was implemented with implicit trust in the device. Due to the difficulty in providing unexpected inputs from the device side, drivers are also not exhaustively tested. Nowadays, using programmable USB devices like FaceDancer [13], it is trivial to launch an attack exploiting a vulnerability in a USB device driver.

Unfortunately, existing defense mechanisms to protect vulnerable drivers from malicious USB devices are limited. Packet filtering-based mechanisms (e.g., LBM [59]) can protect the host system from known attacks, potentially miss-

ing unknown ones. Other mitigations such as Cinch [1] are proposed to protect the host OS from exploits by running vulnerable device drivers in an isolated environment. These mitigations are not deployed due to their inherent complexities and hardware dependencies.

The best alternative to defense mechanisms is to find and fix the bugs. Fuzzing is an automated software testing technique that is widely used to find bugs by feeding randomly-generated inputs to software. Coverage-guided fuzzing, the state-of-art fuzzing technique, is effective in finding bugs in userspace programs [33, 73]. In recent years, several kernel fuzzers (e.g., syzkaller [16], TriforceAFL [19], trinity [22], DIFUZE [10], kAFL [48], or RAZZER [21]) have been developed to fuzz system call arguments, and have discovered many bugs in popular OS kernels [35, 39, 40, 52, 53, 74].

Fuzzing device drivers is challenging due to the difficulty in providing random input from a device. Dedicated programmable hardware devices (e.g., FaceDancer [13]) are expensive and do not scale as one device can only be used to fuzz one target. More importantly, it is challenging to automate fuzzing on real hardware due to the required physical actions (attaching and detaching the device) for each test. Some solutions adapt the kernel. For example, the kernel fuzzer syzkaller [16] contains a usb-fuzzer [14] extension which injects random data to the USB stack via extended syscalls. PeriScope [50] injects random data at the DMA and MMIO interfaces. These approaches are not portable, tightly coupled to a particular OS and kernel version, and require deep understanding of the hardware specification and its implementation in the kernel. In addition, as they inject random data at a certain layer of the IO stack, some code paths cannot be tested, missing bugs in untested code (shown in § 6.2). vUSBf [49] mitigates the requirement to understand the hardware specification by repurposing a networked USB interface [43] to inject random data to drivers. However, vUSBf is too detached from the kernel and only supports dumb fuzzing without collecting coverage feedback.

We introduce USBFuzz, a cheap, portable, flexible, and modular USB fuzzing framework. At its core, USBFuzz uses an emulated USB device to provide fuzz input to a virtualized kernel. In each iteration, a fuzzer executes a test using the emulated USB device virtually attached to the target system, which forwards the fuzzer generated inputs to the drivers under test when they perform IO operations. An optional helper device in the virtualized kernel allows the outside fuzzer to efficiently synchronize coverage maps with the fuzz target.

Due to its modular design and portable device-emulation, USBFuzz is customizable to fuzz USB drivers in *different environments*. We support coverage-guided fuzzing in the Linux kernel or dumb fuzzing in kernels where coverage collection is not yet supported. Similarly, we can either fuzz *broadly* or *focus on a specific driver*. *Broad* fuzzing covers the full USB subsystem and a wide range of drivers, focusing on breadth instead of depth. *Focused* fuzzing targets the specific functionality of a single specific driver (e.g., a webcam).

Leveraging the USBFuzz framework, we applied coverage-guided fuzzing, the state-of-art fuzzing technique, on a broad range of USB drivers in the Linux kernel. In nine recent—already extensively fuzzed—versions of the Linux kernel, we found 16 new memory bugs of high security impact and 20 previous bugs in our ground truth study. Reusing the seeds generated when fuzzing the Linux drivers, we leveraged USBFuzz to fuzz USB drivers on FreeBSD, MacOS, and Windows. So far, we have found one bug in FreeBSD, three bugs (two causing an unplanned restart, one freezing the system) in MacOS and four bugs (resulting in Blue Screens of Death) in Windows. We applied USBFuzz to a specific USB webcam driver, and discovered one bug in the Linux host controller driver. Lastly we found a new bug in a Linux USB camera driver. In total, we discovered 26 new and 20 existing bugs. The main contributions of this paper are as follows:

1. Design and implementation of USBFuzz, a portable, modular and flexible framework to fuzz USB drivers in OS kernels. USBFuzz is customizable to fuzz USB drivers in different kernels, applying coverage-guided fuzzing or dumb fuzzing based on the target OS with different focus. Our prototype supports Linux, FreeBSD, MacOS, and Windows.

2. Design and implementation of a driver-focused coverage collection mechanism for the Linux kernel, allowing the coverage collection across interrupt contexts.

3. In our evaluation, we found 26 new bugs across Linux, FreeBSD, MacOS, and Windows. The discovery of bugs in FreeBSD, Windows, and MacOS highlights the power of our cross-pollination efforts and demonstrates the portability of USBFuzz.

2 Background

The USB architecture implements a complex but flexible communication protocol that has different security risks when hosts communicate with untrusted devices. Fuzzing is a common technique to find security vulnerabilities in software, but existing state-of-the-art fuzzers are not geared towards finding flaws in drivers of peripheral devices.

2.1 USB Architecture

Universal Serial Bus (USB) was introduced as an industry standard to connect commodity computing devices and their peripheral devices. Since its inception, several generations of the USB standard (1.x, 2.0, 3.x) have been implemented with increasing bandwidth to accommodate a wider range of applications. There are over 10,000 different USB devices [54].

USB follows a master-slave architecture, divided into a single host side and potentially many device sides. The device

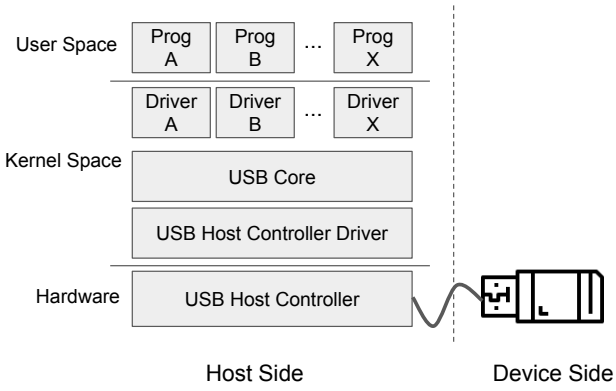


Figure 1: USB architecture

side acts as the slave, and implements its own functionality. The host side, conversely, acts as the master, and manages every device connected to it. All data communication must be initiated by the host, and devices are not permitted to transmit data unless requested by the host.

The most prominent feature of the USB architecture is that it allows a single host to manage different types of devices. The USB standard defines a set of requests that every USB device must respond to, among which the most important are the device descriptor (containing the vendor and product IDs) and the configuration descriptor (containing the device's functionality definition and communication requirements), so that the host-side software can use different drivers to serve different devices according to these descriptors.

The host side adopts a layered architecture with a hardware-based host controller (see Figure 1). The host controller provides physical interfaces (using a root hub component), and supports multiplexing device access, and the host controller driver provides a hardware-independent abstraction layer for accessing the physical interfaces. The *USB core* layer, built on top of the host controller driver, is responsible for choosing appropriate drivers for connected devices and provides core routines to communicate with USB devices. Drivers for individual USB devices (located on top of the USB core) first initialize the device based on the provided descriptors, then interface with other subsystems of the host OS. Userspace programs use APIs provided by various kernel subsystems to communicate with the USB devices.

USB drivers consist of two parts: (i) probe routine to initialize the driver and (ii) function routines to interface with other subsystems (e.g, sound, network, or storage) and deregister the driver when the device is unplugged. Existing USB fuzzers focus exclusively on the probe routines, ignoring other function routines, because probe functions are invoked automatically when the device is plugged in, while other function routines are usually driven by userspace programs.

2.2 USB Security Risks

USB exposes kernel access from externally-connected peripherals, and therefore poses an attack surface. In the past years, several USB-based attacks have been devised to compromise the security of a computer system. We classify the existing USB-based attacks below.

C1. Attacks on implicit trust. As a hardware interface, both OSes and the USB standard implicitly assume that the device is trustworthy. A wide range of USB-based attacks [9, 36, 61] reprogram the device firmware. The reprogrammed devices look like regular USB thumb drives, but perform additional tasks like keylogging (BadUSB [27]) or injecting keystrokes and mouse movements, thus allowing installation of malware, exfiltrating sensitive information (USB Rubber Ducky [6]), installing backdoors, or overriding DNS settings (USB-Driveby [23]).

C2. Electrical attacks. Here, the attacker uses the power bus in the USB cable to send a high voltage to the host, causing physical damage to the hardware components of the host computer. USBKiller [65] is the best known attack falling into this category.

C3. Attacks on software vulnerabilities. The attacker leverages a vulnerability in the USB stack or device drivers. As an example, Listing 1 highlights a Linux kernel vulnerability reported in CVE-2016-2384 [37] where a malicious USB-MIDI [2] device with incorrect endpoints can trigger a double-free bug (one in line 7, and the other in line 18 when the containing object (`chip->card`) is freed).

Memory bugs similar to Listing 1 can be disastrous and may allow an adversary to gain control of the host system, because device drivers run in privileged mode (either in the kernel space or as a privileged process). An exploit for the above vulnerability allows full adversary-controlled code execution [29]. Since devices connected to USB may function as any arbitrary device from the perspective of the host system, the USB interface exposes attacker-controlled input to any service or subsystem of the kernel that is connected through a USB driver. Similar exploits target the storage system of Windows [31].

These security risks are rooted in a basic assumption: hardware is difficult to modify and can be trusted. On one hand, as USB connects hardware devices to computer systems, security issues were neither part of the design of the USB standard nor host side software implementation, making attacks on the trust model (C1) and electrical attacks (C2) possible. On the other hand, device driver developers tend to make assumptions regarding the data read from the device side, e.g., the

```
1  // in snd_usbmidi_create
2  if (quirk && quirk->type == QUIRK_MIDI_MIDIMAN)
3    err = snd_usbmidi_create_endpoints_midiman(
       umidi, &endpoints[0]);
4  else
5    err = snd_usbmidi_create_endpoints(umidi,
       endpoints);
6  if (err < 0) {
7    snd_usbmidi_free(umidi);
8    return err;
9  }
10 // in usb_audio_probe, snd_usb_create_quirk
       calls snd_usbmidi_create
11 err = snd_usb_create_quirk(chip, intf, &
       usb_audio_driver, quirk);
12 if (err < 0)
13   goto __error;
14 //...
15 __error:
16   if (chip)
17     if (!chip->num\_interfaces)
18       snd_card_free(chip->card);
```

Listing 1: CVE-2016-2384 [37] vulnerability

descriptors are always legitimate. This assumption results in the problem that unexpected data read from the device side may be improperly handled. Even if the developers try to handle unexpected values, as recently disclosed bugs demonstrate [15], code is often not well tested due to the difficulty in providing exhaustive unexpected data during development.[1] In other words, when a device driver is written, the programmer can speculate about unexpected inputs, but it is infeasible to create arbitrary hardware that provides such faulty inputs. This results in poorly-tested error-handling code paths.

However, recent research has fundamentally changed this basic assumption. Some USB device firmware is vulnerable, allowing attackers to control the device and messages sent on the bus. In addition, with the adoption of recent technologies such as Wireless USB [70] and USBIP [46], the USB interface is exposed to networked devices, turning USB-based attacks into much easier network attacks. Finally, reprogrammable USB devices (e.g., FaceDancer [13]) allow the implementation of arbitrary USB devices in software.

2.3 Fuzzing the USB Interface

Given the security risks, there have been several fuzzing tools targeting the USB interface. This section briefly analyzes these existing fuzzing tools and serves to motivate our work.

The first generation of USB fuzzers targets the device level. vUSBf [49] uses a networked USB interface (usbredir [43]), and umap2 [18] uses programmable hardware (FaceDancer [13]) to inject random hardware input into the host USB stack. Though easily portable to other OSes, they are dumb fuzzers and cannot leverage coverage information to guide their input mutation, rendering them inefficient.

The recent usb-fuzzer [14] (an extension of the kernel fuzzer syzkaller [16]) injects fuzz inputs into the IO stack of the Linux kernel using a custom software-implemented host controller combined with a coverage-guided fuzzing technique. The adoption of coverage-guided fuzzing has led to the discovery of many bugs in the USB stack of the Linux kernel [14]. However, usb-fuzzer is tightly coupled with the Linux kernel, making it hard to port to other OSes.

All existing USB fuzzers focus exclusively on the probe routines of drivers, not supporting fuzzing of the remaining function routines. The status-quo of existing USB fuzzers motivates us to build a flexible and modular USB fuzzing framework that is portable to different environments and easily customizable to apply coverage-guided fuzzing or dumb fuzzing (in kernels where coverage collection is not yet supported), and allows fuzzing a broad range of probe routines or focusing on the function routines of a specific driver.

3 Threat Model

Our threat model consists of an adversary that attacks a computer system through the USB interface, leveraging a software vulnerability in the host software stack to achieve goals such as privilege escalation, code execution, or denial of service. Attacks are launched by sending prepared byte sequences over the USB bus, either attaching a malicious USB device to a physical USB interface or hijacking a connection to a networked USB interface (e.g., in USBIP [46] or usbredir [43]).

4 USBFuzz Design

Device drivers handle inputs both from the device side and from the kernel. The kernel is generally trusted but the device may provide malicious inputs. The goal of USBFuzz is to find bugs in USB drivers by repeatedly testing them using random inputs generated by our fuzzer, instead of the input read from the device side. The key challenge is how to feed the fuzzer generated inputs to the driver code. Before presenting our approach, we discuss the existing approaches along with their respective drawbacks.

Approach I: using dedicated hardware. A straightforward solution is to use dedicated hardware which returns customizable data to drivers when requested. For USB devices, FaceDancer [13] is readily available and used by umap2 [18]. This approach follows the data paths in real hardware and thus covers the complete code paths and generates reproducible inputs. However, there are several drawbacks in such a hardware-based approach. First, dedicated hardware parts incur hardware costs. While $85 for a single FaceDancer

[1] Special hardware that provides unexpected data from the USB device side exists (e.g., Ellisys USB Explorer [12]), however it is either not used because of its cost, or the drivers are not sufficiently tested.

is not prohibitively expensive, fuzzing campaigns often run on 10s to 1000s of cores, resulting in substantial hardware cost. Similarly, connecting physical devices to fuzzing clusters in a server center results in additional complexity. Second, hardware-based approaches do not scale as one device can only fuzz one target at a time. Hardware costs and lack of scalability together render this approach expensive. Finally, this approach is hard to automate as hardware operations (e.g., attaching and detaching a device to and from a target system) are required for each test iteration.

Approach II: data injection in IO stack. This approach modifies the kernel to inject fuzz data to drivers at a certain layer of the IO stack. For example, usb-fuzzer in syzkaller [16] injects fuzz data into the USB stack through a software host controller (`dummy hcd`), replacing the driver for the hardware host controller. PeriScope [50] injects fuzzer generated input to drivers by modifying MMIO and DMA interfaces.

Compared to hardware-based approaches, this approach is cheap, scalable, and can be automated to accommodate fuzzing. However, this solution struggles with portability as its implementation is tightly coupled to a given kernel layer (and sometimes kernel version). In addition, it requires deep understanding of the hardware specification and its implementation in the kernel. As input is injected at a specific layer of the IO stack, it cannot test code paths end-to-end, and thus may miss bugs in untested code paths (as we show in § 6.4).

Design Goals. After evaluating the above approaches, we present the following design goals:

G1. Low Cost: The solution should be cost-effective and hardware-independent.

G2. Portability: The solution should be portable to test other OS and platforms, avoiding deep coupling with a specific kernel version.

G3. Minimal Required Knowledge: The interaction between the driver, the USB device, and the rest of the system is complex and may be different from device to device. The solution should require minimal knowledge of the USB standard and the device.

USBFuzz's approach. At a high-level, USBFuzz leverages an emulated USB device to feed random input to device drivers. The target kernel (hosting the tested device drivers) runs in a virtual machine (VM) and the emulated USB device is integrated into the VM. The hypervisor in the VM transparently sends read/write requests from the drivers of the guest kernel to the emulated device (and not to real hardware) without any changes to the USB system in the target kernel. The emulated USB device, on the other hand, responds to kernel IO requests using the fuzzer-generated input, instead of following the specification of a device.

As a software-based solution, an emulated device does not incur any hardware cost and is highly scalable, as we can

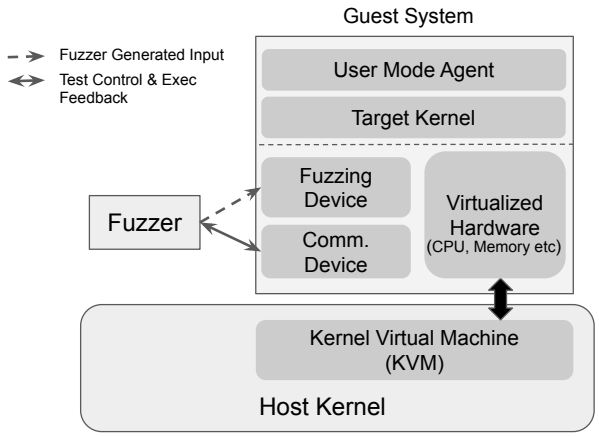


Figure 2: Overview of USBFuzz

easily run multiple instances of a virtual machine to fuzz multiple instances of a target kernel in parallel, satisfying **G1**—low cost. Because our solution implements an emulated hardware device, it is decoupled from a specific kernel or version. One implementation of the emulated device can be used to provide random input to device drivers running on different kernels on different platforms, satisfying **G2**—portability. As this solution works at the device level, no knowledge of the software layers in the kernel is required. In addition, based on mature emulators such as QEMU, a developer only needs to understand the data communication protocol, satisfying **G3**—minimal required knowledge.

Based on these goals, we designed USBFuzz, a modular framework to fuzz USB device drivers. Figure 2 illustrates the overall design of USBFuzz. The following list summarizes high level functionalities of its main components.

Fuzzer: The fuzzer runs as a userspace process on the host OS. This component performs the following tasks: (i) mutating the data fed to device drivers in the target kernel; and (ii) monitoring and controlling test execution.

Guest System: The guest system is a virtual machine that runs a target kernel containing the device drivers to test. It provides support for executing the guest code, emulating the fuzzing device as well as the supporting communication device.

Target Kernel: The target kernel contains the code (importantly, device drivers) and runs inside the guest system. The drivers in the kernel are tested when they process the data read from the emulated fuzzing device.

Fuzzing Device: The fuzzing device is an emulated USB device in the guest system. It is connected through the emulated USB interface to the guest system. However, instead of providing data according to the hardware specification, it forwards the fuzzer-generated data to the

host when the target kernel performs IO operations on it (shown in § 4.1).

Communication Device: The communication device is an emulated device in the guest system intended to facilitate communication between the guest system and the fuzzer component. It shares a memory region and provides synchronization channels between the fuzzer component and the guest system. The shared memory region also shares coverage information in coverage-guided fuzzing (shown in § 4.2).

User Mode Agent: This userspace program runs as a daemon process in the guest system. It monitors the execution of tests (shown in § 4.3). Optionally, it can be customized to perform additional operations on the fuzzing device to trigger function routines of drivers during focused fuzzing (demonstrated in § 6.4).

The modular design of USBFuzz, in combination with the emulated fuzzing device, allows fuzzing USB device drivers on different OSes and applying different fuzzing techniques with flexible configuration based on the target system, e.g., coverage-guided fuzzing to leverage feedback, or dumb fuzzing without any feedback to explore certain provided USB traces (dumb fuzzing is useful when coverage information is not available). In this work, we applied coverage-guided fuzzing to the Linux kernel (discussed in § 4.4), and dumb fuzzing to FreeBSD, MacOS, and Windows using cross-pollination seeded by inputs generated from fuzzing Linux.

4.1 Providing Fuzzed Hardware Input

Our input generation component extends AFL, one of the most popular mutational coverage-guided fuzzing engines. AFL [72] uses a file to communicate the fuzzer generated input with the target program. The fuzzing device responds to read requests from device drivers with the contents of the file.

As mentioned in § 2.1, when a USB device is attached to a computer, the USB driver framework reads the device descriptors and configuration descriptors and uses the appropriate driver to interact with it. However, depending on the implementation of the USB stack, the device descriptor and configuration descriptor may be read multiple times (e.g., the Linux kernel reads the device descriptor both before and after setting the address of the USB device). To improve fuzzing efficiency and considering that throughput is relatively low compared to simple user space fuzzing (see § 6.3), these two requests are handled separately: they are loaded (either from a separate file or the fuzzer generated file) once when the fuzzing device is initialized and our framework responds with the same descriptors when requested. All other requests are served with bytes from the current position of the fuzzer generated file until no data is available, in which case, the device responds with no data. Note that as we are fuzzing the device drivers using data read from the device side, write operations to the device are ignored.

This design allows either *broad* fuzzing or *focused* fuzzing. By allowing the fuzzer to mutate the device and configuration descriptors (loading them from the fuzzer generated file), we can fuzz the common USB driver framework and drivers for a wide range of devices (broad fuzzing); by fixing the device and configuration descriptor to some specific device or class of devices (loading them from a separate configuration file), we can focus on fuzzing of a single driver (focused fuzzing). This flexibility enables different scenarios, e.g., it allows bug hunting in the USB driver framework and all deployed USB device drivers, or it can be used to test the driver of a specific USB device during the development phase. We demonstrate *focused fuzzing* on a USB webcam driver in § 6.4.

4.2 Fuzzer – Guest System Communication

Like all existing fuzzers, the fuzzer component in USBFuzz needs to communicate with the target code to exert control over tests, reap coverage information, and so forth. As shown in Figure 2, the fuzzer component runs outside the guest system and cannot gain information about the target system directly. The communication device is intended to facilitate the communication between the fuzzer and the guest system.

In a coverage-guided fuzzer, coverage information needs to be passed from the guest system to the fuzzer. To avoid repeated memory copy operations, we map the bitmap, which is a memory area in the fuzzer process, to the guest system using a QEMU communication device. After the guest system is fully initialized, the bitmap is mapped to the virtual memory space of the target kernel, to which the instrumented code in the target kernel can write the coverage information. As it is also a shared memory area in the fuzzer process, the coverage information is immediately accessible by the fuzzer component, avoiding memory copy operations.

In addition, the fuzzer component needs to synchronize with the user mode agent running in the guest system (see § 4.3) in each fuzz test iteration. To avoid heavy-weight IPC operations, a control channel is added to the communication device to facilitate the synchronization between the user mode agent and the fuzzer component.

4.3 Test Execution and Monitoring

Existing kernel fuzzers execute tests using the process abstraction of the target kernel. They follow an iterative pattern where, for each test, a process is created, executed, monitored, and the fuzzer then waits for the termination of the process to detect the end of the test. In USBFuzz, as tests are performed using the fuzzing device, in each iteration, a test starts with virtually attaching the (emulated) fuzzing device to the guest system. The kernel then receives a request for the new USB device that is handled by the low-end part of the kernel device

management which loads the necessary drivers and initializes the device state. However, without support from the kernel through, e.g., process abstractions similar to the `exit` system call, it is challenging to monitor the execution status (e.g., whether a kernel bug is triggered or not) of the kernel during its interaction with the device.

In USBFuzz, we follow an empirical approach to monitor the execution of a test by the kernel: by checking the kernel's logged messages. For example, when a USB device is attached to the guest system, if the kernel is able to handle the inputs from the device, the kernel will log messages containing a set of keywords indicating the success or failure of the interaction with the device. Otherwise, if the kernel cannot handle the inputs from the device, the kernel will freeze or indicate that a bug was triggered. The USBFuzz user mode agent component monitors the execution status of a test by scanning kernel logs from inside the virtualized target system, synchronizing its status with the fuzzer component so that it records bug triggering inputs and continues to the next iteration.

To avoid repeatedly booting the guest system for each iteration, USBFuzz provides a persistent fuzzing technique, similar to other kernel fuzzers (syzkaller [16], TriforceAFL [19], trinity [22], or kAFL [48]), where a running target kernel is reused for multiple tests until it freezes, in which case, the fuzzer automatically restarts the kernel.

4.4 Coverage-Guided Fuzzing on Linux

So far, the USBFuzz framework provides basic support for fuzzing USB device drivers on different OSes. However, to enable coverage-guided fuzzing, the system must collect execution coverage. A coverage-guided fuzzer keeps track of code coverage exercised by test inputs and mutates interesting inputs which trigger new code paths.

Coverage collection is challenging for driver code in kernel space. On one hand, inputs from the device side may trigger code executions in different contexts, because drivers may contain code running in interrupts and kernel threads. On the other hand, due to the kernel performing multitasking, code executed in a single thread may be preempted by other unrelated code execution triggered by timer interrupts or task scheduling. To the best of our knowledge, the Linux kernel only supports coverage collection by means of static instrumentation through `kcov` [67]. However, `kcov` coverage collection is limited to a single process, ignoring interrupt contexts and kernel threads. Extending the static instrumentation of `kcov`, we devised an AFL-style edge coverage scheme to collect coverage in USB device drivers of the Linux kernel. To collect coverage across different contexts, (i) the previous executed block is saved in the context of each thread of code execution (interrupts or kernel threads), so that edge transitions are not mangled by preempted flows of code execution; and (ii) instrumentation is limited to related code: USB core, host controller drivers, and USB drivers.

5 Implementation Details

The implementation of the USBFuzz framework extends several open source components including QEMU [4, 57] (where we implement the communication device and the emulated USB device), AFL [72] (which we modify to target USB devices by collecting coverage information from our virtualized kernel and interacting with our User Mode Agent), and `kcov` [67] (which we extend to track edge coverage across the full USB stack, including interrupt contexts). We implement the user mode agent from scratch. The workflow of the whole system, illustrating the interaction among the components, is presented in Figure 3. The implementation details of individual components are discussed in the following sections.

When the fuzzer starts, it allocates a memory area for the bitmap and exports it as a shared memory region, with which the communication device is initialized as QEMU starts. After the target kernel is booted, the user mode agent runs and notifies the fuzzer to start testing.

In each iteration of the fuzzing loop, the fuzzer starts a test by virtually attaching the fuzzing device to the target system. With the attachment of the fuzzing device, the kernel starts its interaction with the device and loads appropriate USB drivers for it. The loaded USB driver is tested with the fuzz input as it interacts with the fuzzing device. The user mode agent monitors execution by scanning the kernel log and notifies the fuzzer of the result of the test. The fuzzer completes the test by virtually detaching the fuzzing device from the target system.

5.1 Communication Device

The communication device in USBFuzz facilitates lightweight communication between the fuzzer component and the target system, which includes sharing the bitmap area and synchronization between the user mode agent and the fuzzer component. The implementation of the communication device is built on the IVSHMEM (Inter-VM shared memory) device [56], which is an emulated PCI device in QEMU. The shared memory region from the fuzzer component is exported to the guest system as a memory area in IVSHMEM device and mapped to the virtual memory

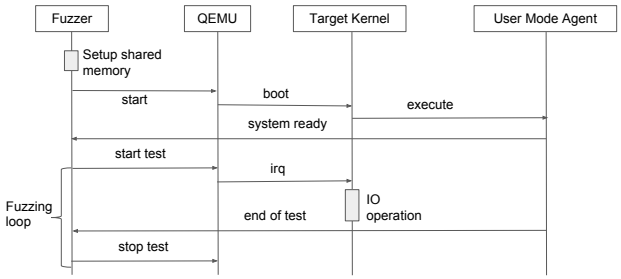


Figure 3: Workflow of USBFuzz.

space of the guest system. One register (BAR2, the Base Address Register for a memory or IO space) is used for the communication channel between the fuzzer component and the user mode agent.

5.2 Fuzzer

The fuzzer uses two pipes to communicate with the VM: a control pipe and a status pipe. The fuzzer starts a test by sending a message to the VM via the control pipe, and it receives execution status information from the VM via the status pipe.

On the VM side, two callbacks are registered for the purpose of interfacing with the fuzzer component. One callback attaches a new instance of the fuzzing device to the hypervisor with the fuzzer-generated input when a new message is received from the control pipe. When execution status information is received from the user mode agent via the communication device, the other callback detaches the fuzzing device from the hypervisor and forwards execution status information to the fuzzer via the status pipe.

5.3 Fuzzing Device

The fuzzing device is the key component in USBFuzz that enables fuzzing of the hardware input space of the kernel. It is implemented as an emulated USB device in the QEMU device emulation framework and mimics an attacker-controlled malicious device in real-world scenarios.

Hypervisors intercept all device read/write requests from the guest kernel. Every read/write operation from the kernel of the guest OS is dispatched to a registered function in the emulated device implementation, which performs actions and returns data to the kernel following the hardware specification.

The fuzzing device is implemented by registering "read" functions which forward the fuzzer-generated data to the kernel. To be more specific, the bytes read by device drivers are mapped sequentially to the fuzzer-generated input, except the device and configuration descriptors, which are handled separately (as mentioned in § 4.1).

5.4 User Mode Agent

The user mode agent is designed to be run as a daemon process in the guest OS and is automatically started when the target OS boots up. It monitors the execution status of tests based on the kernel log and passes information to the fuzzer via the communication device. After initialization, it notifies the fuzzer that the target kernel is ready to be tested.

On Linux and FreeBSD, our user mode agent component monitors the kernel log file (`/dev/kmsg` in Linux, `/dev/klog` in FreeBSD), and scans it for error messages indicating a kernel bug or end of a test. If either event is detected, it notifies the fuzzer—using the device file exported to user space by the communication device driver—to stop the current iteration and proceed to the next one. The set of error messages is borrowed from the report package [44] of syzkaller. On Windows and MacOS, due to the lack of a clear signal from the kernel when devices are attached/detached, our user mode agent uses a fixed timeout (1 second on MacOS and 5 seconds on Windows) to let the device properly initialize.

5.5 Adapting Linux kcov

To apply coverage-guided fuzzing on USB drivers for the Linux kernel, we use static instrumentation to collect coverage from the target kernel. The implementation is adapted from `kcov` [67] which is already supported by the Linux kernel with the following modifications to accommodate our design.

```
1 index = (hash(IP) ^ hash(prev_loc))%BITMAP_SIZE;
2 bitmap[index] ++;
3 prev_loc = IP;
```

Listing 2: Instrumentation used in USBFuzz

USBFuzz implements an AFL-style [72] edge coverage scheme by extending `kcov`. Our modification supports multiple paths of execution across multiple threads and interrupt handlers, untangling non-determinism. We save the previous block whenever non-determinism happens. For processes, we save `prev_loc` (see Listing 2) in the `struct task` (the data structure for the process control block in the Linux kernel), and for interrupt handlers we save `prev_loc` on the stack. Whenever non-determinism happens, the current previous location is spilled (in the `struct task` for kernel threads, or on the stack for interrupt handlers) and set to a well-defined location in the coverage map, untangling non-determinism to specific locations. When execution resumes, the spilled `prev_loc` is restored. Note that this careful design allows us to keep track of the execution of interrupts (and nested interrupts) and separates their coverage without polluting the coverage map through false updates.

The instrumented code is modified to write the coverage information to the memory area of the communication device, instead of the per-process buffer. The Linux build system is modified to limit the instrumentation to only code of interest. In our evaluation, we restrict coverage tracking to anything related to the USB subsystem, including drivers for both host controllers and devices.

6 Evaluation

We evaluate various aspects of USBFuzz. First, we perform an extensive evaluation of our coverage-guided fuzzing implementation on the USB framework and its device drivers (broad fuzzing) in the Linux kernel. § 6.1 presents the discovered bugs, and § 6.3 presents the performance analysis. Second, we compare USBFuzz to the usb-fuzzer extension

of syzkaller based on code coverage and bug discovery capabilities (§ 6.2). In § 6.4, we demonstrate the flexibility of USBFuzz by fuzzing (i) USB drivers in FreeBSD, MacOS, and Windows (broad fuzzing); and (ii) a webcam driver (focused fuzzing). Finally, we showcase one of the discovered bugs in the USB core framework of the Linux kernel (§ 6.5).

Hardware and Software Environment. We execute our evaluation on a small cluster in which each of the four nodes runs Ubuntu 16.04 LTS with a KVM hypervisor. Each node is equipped with 32 GB of memory and an Intel i7-6700K processor with Intel VT [20] support.

Guest OS Preparation. To evaluate FreeBSD, Windows, and MacOS, we use VM images with unmodified kernels and a user mode agent component running in userspace. When evaluating Linux, the target kernel is built with the following customization: (i) we adapt `kcov` as mentioned in § 5.5; (ii) we configure all USB drivers as built-in; (iii) we enable kernel address sanitizer (KASAN) [25, 26] to improve bug detection capability. At runtime, to detect abnormal behavior triggered by the tests, we configure the kernel to panic in case of "oops" or print warnings by customizing kernel parameters [62].

Seed Preparation. To start fuzzing, we create a set of USB device descriptors as seeds. We leverage the set of expected identifiers (of devices, vendors, products, and protocols) and matching rules of supported devices that syzkaller [16] extracted from the Linux kernel [64]. A script converts the data into a set of files containing device and configuration descriptors as fuzzing seeds.

6.1 Bug Finding

To show the ability of USBFuzz to find bugs, we ran USBFuzz on 9 recent versions of the Linux kernel: v4.14.81, v4.15, v4.16, v4.17, v4.18.19, v4.19, v4.19.1, v4.19.2, and v4.20-rc2 (the latest version at the time of evaluation). Each version was fuzzed with four instances for roughly four weeks (reaching, on average, approximately 2.8 million executions) using our small fuzzing cluster.

Table 1 summarizes all of the bugs USBFuzz found in our evaluation. In total, 47 unique bugs were found. Of these 47 bugs, 36 are memory bugs detected by KASAN [25], including double-free (2), NULL pointer dereference (8), general protection error (6), out-of-bounds memory access (6), and use-after-free (14). 16 of these memory bugs are new and have never been reported. The remaining 20 memory bugs were reported before, and so we used them as part of our ground truth testing. Memory bugs detected by KASAN are serious and may potentially be used to launch attacks. For example, NULL pointer dereference bugs lead to a crash, resulting in denial of service. Other types of memory violations such as use-after-free, out-of-bounds read/write, and double frees can be used to compromise the system through a code execution attack or to leak information. We discuss one of our discovered memory bugs and analyze its security impact in detail in our case study in § 6.5.

Type	Bug Symptom	#
Memory Bugs (36)	double-free	2
	NULL pointer dereference	8
	general protection	6
	slab-out-of-bounds access	6
	use-after-free access	14
Unexpected state reached (11)	WARNING	9
	BUG	2

Table 1: Bug Classification

The remaining 11 bugs (WARNING, BUG) are caused by execution of (potentially) dangerous statements (e.g., assertion errors) in the kernel, which usually represent unexpected kernel states, a situation that developers may be aware of but that is not yet properly handled. The impact of such bugs is hard to evaluate in general without a case-by-case study. However, providing a witness of such bugs enables developers to reproduce these bugs and to assess their impact.

Bug Disclosure. We are working with the Linux and Android security teams on disclosing and fixing all discovered vulnerabilities, focusing first on the memory bugs. Table 2 shows the 11 **new** *memory bugs* that we fixed so far. These new bugs were dispersed in different USB subsystems (USB Core, USB Sound, or USB Network) or individual device drivers. From these 11 new bugs, we have received 10 CVEs. The remaining bugs fall into two classes: those still under embargo/being disclosed and those that were concurrently found and reported by other researchers. Note that our approach of also supplying patches for the discovered bugs reduces the burden on the kernel developers when fixing the reported vulnerabilities.

6.2 Comparison with syzkaller

Due to challenges in porting the kernel-internal components of syzkaller, we had to use a version of the Linux kernel that is supported by syzkaller. We settled on version v5.5.0 [17], as it is maintained by the syzkaller developers. In this version, many of the reported USB vulnerabilities had already been fixed. Note that USBFuzz does not require any kernel components and supports all recent Linux kernels, simplifying porting and maintenance. In this syzkaller comparison we evaluate coverage and bug finding effectiveness, running five 3-day campaigns of both USBFuzz and syzkaller.

Bug Finding. In this heavily patched version of the Linux kernel, USBFuzz found 1 bug in each run within the first day and syzkaller found 3 different bugs (2 runs found 2, 3 runs found 3). The bug USBFuzz found is a new bug that triggers a *BUG_ON* statement in a USB camera driver [32]. The bugs found by syzkaller trigger WARNING statements in different USB drivers.

Kernel bug summary	Kernel Subsystem	Confirmed Version	Fixed
KASAN: SOOB Read in __usb_get_extra_descriptor	USB Core	4.14.81 - 4.20-rc2	✓
KASAN: UAF Write in usb_audio_probe	USB Sound	4.14.81 - 4.20-rc2	✓
KASAN: SOOB Read in build_audio_procunit	USB Sound	4.14.81 - 4.20-rc2	✓
KASAN: SOOB Read in parse_audio_input_terminal	USB Sound	4.14.81 - 4.18	✓
KASAN: SOOB Read in parse_audio_mixer_unit	USB Sound	4.14.81 - 4.20-rc2	✓
KASAN: SOOB Read in create_composite_quirks	USB Sound	4.14.81 - 4.20-rc2	✓
KASAN: SOOB Write in check_input_term	USB Sound	4.14.81 - 4.20-rc2	✓
KASAN: SOOB Read in hso_get_config_data	USB Network	4.14.81 - 4.20-rc2	✓
KASAN: NULL deref in ath{6kl,10k}_usb_alloc_urb_from_pipe	Device Driver	4.14.81 - 4.20-rc2	✓
KASAN: SOOB Read in lan78xx_probe	Device Driver	4.14.81 - 4.17	✓
KASAN: double free in rsi_91x_deinit	Device Driver	4.17 - 4.20-rc2	✓

Table 2: USBFuzz's new memory bugs in 9 recent Linux kernels (SOOB: slab-out-of-bounds, UAF: use-after-free) that we fixed.

	Line (%)	Function (%)	Branch (%)
syzkaller	18,039 (4.5)	1,324 (5.6)	7,259 (3.2)
USBFuzz	10,325 (2.5)	813 (3.5)	4,564 (2.0)

Table 3: Comparison of line, function, and branch coverage in the Linux kernel between syzkaller and USBFuzz. The results are shown as the average of 5 runs.

Code Coverage. We collected accumulated code coverage in the USB related code (including the USB core framework, host controller drivers, gadget subsystem, and other device drivers) by replaying inputs generated from both fuzzers. The line, function, and branch coverage of 5 runs are shown in Table 3. Overall, syzkaller outperforms USBFuzz on maximizing code coverage. We attribute the better coverage to the manual analysis of the kernel code and custom tailoring the individual generated USB messages to the different USB drivers and protocols. The manual effort results in messages adhering more closely to the standard [55]—at a high engineering cost.

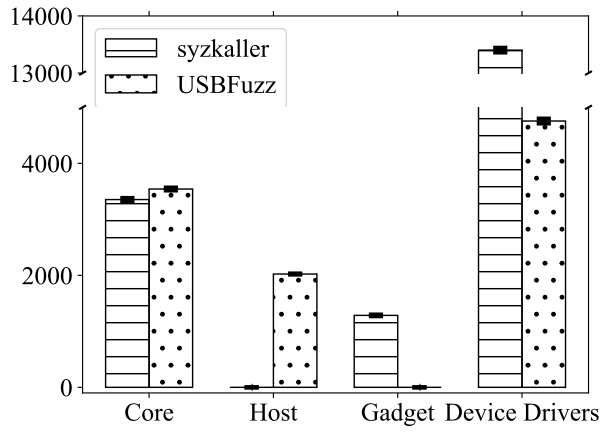


Figure 4: Comparison of line coverage between syzkaller and USBFuzz in USB Core, host controller drivers, gadget subsystem, and other device drivers.

Table 3 shows that both syzkaller and USBFuzz only triggered limited code coverage. There are three reasons: (i) some drivers are not tested at all; (ii) some code (function routines) can be triggered only by operations from userspace, and are thus not covered; (iii) some host controller drivers can only be covered with a specific emulated host controller.

Figure 4 demonstrates the differences between USBFuzz and syzkaller. First, syzkaller triggered zero coverage in the host controller drivers. This is because syzkaller uses a USB gadget and a software host controller (dummy HCD) while USBFuzz leverages an emulated USB device to feed fuzzer generated inputs to drivers. Though syzkaller may find bugs in the USB gadget subsystem, which is only used in embedded systems as firmware of USB devices and not deployed on PCs, it cannot find bugs in host controller drivers. We show a bug found in XHCI driver in our extended evaluation in § 6.4.

Syzkaller achieves better overall coverage for device drivers due to the large amount of individual test cases that are fine-tuned. These syzkaller test cases can be reused for focused, per device fuzzing in USBFuzz to extend coverage. USBFuzz achieves better coverage in USB core, which contains common routines for handling data from the device side. This is caused by the difference in the input generation engines of the two fuzzers. As a generational fuzzer, syzkaller's input generation engine always generates valid values for some data fields, thus prohibiting it from finding bugs triggered by inputs that violate the expected values in these fields. USBFuzz, on the other hand, generates inputs triggering such code paths. Note that the driver in which USBFuzz found a bug was previously tested by syzkaller. However, as the inputs it generated are well-formed, the bug was missed. We show an example of this in § 6.5.

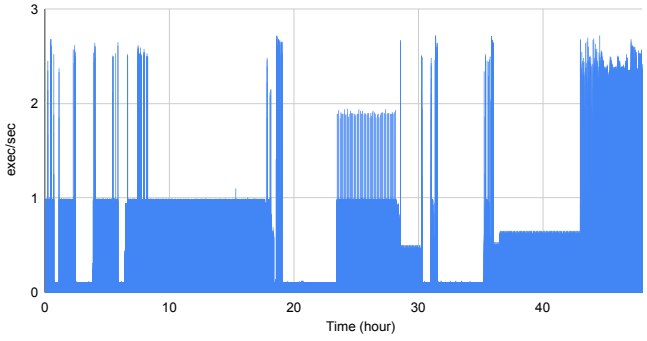
In summary, syzkaller leverages manual engineering to improve input generation for specific targets but misses bugs that are not standard compliant or outside of where the input

is fed into the system. USBFuzz follows an out-of-the box approach where data is fed into the unmodified subsystem, allowing it to trigger broader bugs. These two systems are therefore complementary and find different types of bugs and should be used concurrently. As future work, we want to test the combination of the input generation engines, sharing seeds between the two.

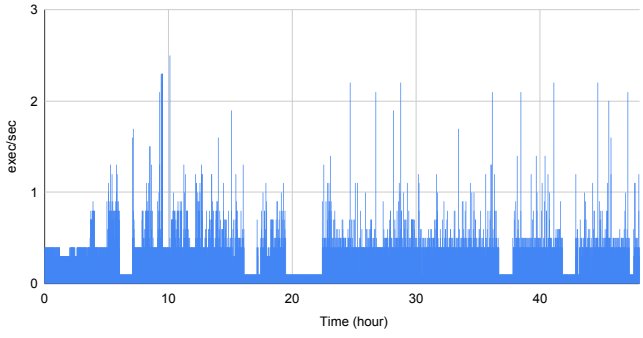
6.3 Performance Analysis

To assess the performance of USBFuzz we evaluate execution speed and analyse time spent in different fuzzing phases.

Fuzzing Throughput. Figure 5(a) shows the execution speed of USBFuzz in a sampled period of 50 hours while running on Linux 4.16. The figure demonstrates that USBFuzz achieves a fuzzing throughput ranging from 0.1–2.6 exec/sec, much lower than that of userspace fuzzers where the same hardware setup achieves up to thousands of executions per second. Note the low fuzzing throughput in this scenario is mostly not caused by USBFuzz, because tests on USB drivers run much longer than userspace programs. E.g., our experiment with physical USB devices shows that it takes more than 4 seconds to fully recognize a USB flash drive on a physical machine. A similar throughput (0.1–2.5 exec/sec) is observed in syzkaller and shown in Figure 5(b).



(a) A sample of execution speed of USBFuzz



(b) A sample of execution speed of syzkaller

Figure 5: Comparison of execution speed between USBFuzz (0.1–2.6 exec/sec) and syzkaller (0.1- 2.5 exec/sec).

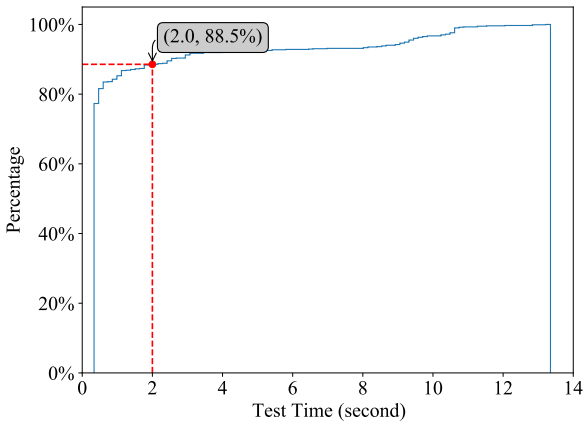


Figure 6: Cumulative distribution of test run time, collected by tracing the inputs generated by USBFuzz.

Overhead Breakdown. To quantify the time spent for each executed test, and to evaluate possible improvements in fuzzing throughput, we performed an in-depth investigation on the time spent at each stage of a test. As mentioned in § 5, a test is divided into 3 stages, (i) virtually attaching the fuzzing device to the VM; (ii) test execution; and (iii) detaching the fuzzing device. We measure the time used for attaching/detaching, and the time used in running a test when device drivers perform IO operations. The result is shown in Figure 7. The blue line and red line show the time used in the attach/detach operations (added together) and the time used in tests respectively. From Figure 7, the time used in these attach/detach operations remains stable at about 0.22 second, while the time used by tests varies from test to test, ranging from 0.2 to more than 10 seconds.

Manual investigation on the test cases shows that the time a test takes depends on the quality of input. If the input fails the first check on the sanity of the device descriptor, it finishes very quickly. If the emulated device passes initial sanity checks and is bound to a driver, the execution time of a test depends on the driver implementation. Typically longer tests trigger more complex code paths in device drivers. Figure 6 depicts the runtime distribution of tests generated by USBFuzz. It shows that about 11% of the generated tests last longer than 2 seconds.

We also evaluated the overhead caused by the user mode agent component. We measured the time used to run tests on a base system with the user mode agent running and that without user mode agent, a comparison shows that the difference is roughly 0.01 second, which is negligible compared to the overall test execution time.

Though the overhead of attach/detach operations is negligible for long tests, it accounts for about 50% of the total execution time of short tests. As the emulated device is allo-

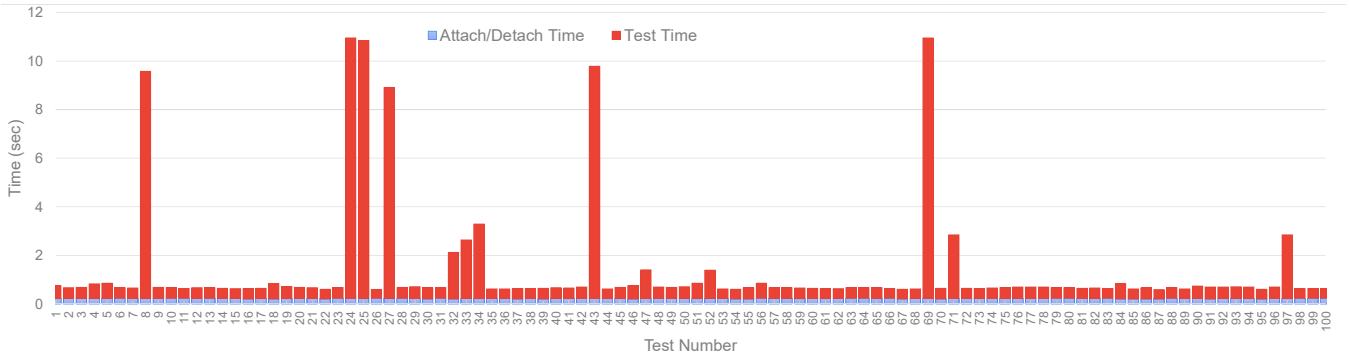


Figure 7: Execution Time Breakdown of 100 randomly chosen tests. The axes denote test number and execution time. Blue and red bars represent time used in attaching/detaching the emulated device to the VM and the time spent in testing respectively.

cated/deallocated before/after the test in each iteration, this overhead can be reduced by caching the emulated device and performing only necessary initialization operations. We leave this optimization as future work.

6.4 USBFuzz Flexibility

To demonstrate the benefit of portability and flexibility of the USBFuzz framework, we performed two extended evaluations: (i) fuzzing FreeBSD, MacOS, and Windows; (ii) focused fuzzing a USB webcam driver.

Fuzzing FreeBSD, MacOS, and Windows. Leveraging the portability of a device emulation-based solution to feed fuzzer-generated inputs to device drivers, we extended our evaluation to FreeBSD 12 (the latest release), MacOS 10.15 Catalina (the latest release) and Windows (both version 8 and 10, with most recent security updates installed). After porting the user mode agent and the device driver of the communication device we apply dumb fuzzing on these OSes.

Fuzzing drivers on these OSes is more challenging than the Linux kernel due to the lack of support infrastructure. These OSes support neither KASAN, other sanitizers, nor coverage-based collection of executions. The lack of a memory-based sanitizer means our fuzzer only discovers bugs that trigger exceptions, and misses all bugs that silently corrupt memory. Because we cannot collect coverage information, our fuzzer cannot detect seeds that trigger new inputs.

To alleviate the second concern, the lack of coverage-guided optimization, we experiment with cross-pollination. To seed our dumb fuzzer, we reuse the inputs generated during our Linux kernel fuzzing campaign.

USBFuzz found three bugs (two resulting unplanned restart and one resulting system freeze) on MacOS, and four bugs on Windows (resulting in a Blue Screen of Death, confirmed on both Window 8 and Windows 10) during the first day of evaluation. Additionally, one bug was found in a USB Bluetooth dongle driver on FreeBSD in two weeks. In this bug, the driver is trying to add an object to a finalized container.

Focused fuzzing on the LifeCam VX-800 driver. So far, we let the fuzzer create emulated USB peripherals as part of the input generation process. Here we want to show the capability of USBFuzz of fuzzing focusing on a specific device. We extract the device and configuration descriptor from a real LifeCam VX-800 [34] webcam (with the lsusb [11] utility) and let USBFuzz create a fake USB device based on that information, enabling the Linux kernel to detect and bind a video driver to it.

We extended the user mode agent to receive a picture from the webcam with streamer [63][2] using the emulated device. After fuzzing this targeted device for a few days with randomly generated inputs, we found another bug in the XHCI [68] driver of the Linux kernel. The buggy input triggers an infinite loop in the driver, in which the driver code keeps allocating memory in each iteration until the system runs out of memory.

USBFuzz Flexibility. The bugs found in the FreeBSD, MacOS and Windows, and XHCI driver demonstrate the advantage of USBFuzz compared to syzkaller's usb-fuzzer. As the implementation of usb-fuzzer only depends on the Linux kernel, it cannot be ported other OSes without a full reimplementation. Moreover, as usb-fuzzer injects fuzzer-generated inputs via a software host controller (dummy HCD [51]), it is unable to trigger bugs in drivers of physical host controllers.

6.5 Case Study

In this section, we discuss a new bug USBFuzz discovered in the USB core framework of the Linux kernel. In the USB standard, to enable extensions, a device is allowed to define other customized descriptors in addition to the standard descriptors. As the length of each descriptor varies, the USB standard defines the first two bytes of a descriptor to represent the length and type of a descriptor (as shown by `usb_descriptor_header` in Listing 3). All descriptors must follow the same format. For example, an OTG (USB On-The-

[2] We execute the `streamer -f jpeg -o output.jpeg` command.

Go, a recent extension which allows a USB device to act as a host [69]) descriptor (shown as `usb_otg_desciptor` in Listing 3) has three bytes and thus a correct OTG descriptor must start with a `0x03` byte.

Descriptors are read from the device, and therefore, cannot be trusted and must be sanitized. In the Linux kernel, `__usb_get_extra_descriptor` is one of the functions used by the USB core driver to parse the customized descriptors. Listing 3 shows that the code simply scans the data (`buffer` argument) read from the device side. To match descriptors for a given type (`type` argument) it returns the first match.

When handling maliciously crafted descriptors, this implementation is vulnerable. By providing a descriptor that is shorter than its actual length, the attacker can trigger an out-of-bounds memory access. E.g., a two byte (invalid) OTG descriptor with the third byte missing will be accepted by `__usb_get_extra_descriptor` and treated as valid. If the missing field is accessed (e.g., the read of `bmAttributes` at line 30), an out-of-bounds memory access occurs.

Depending on how the missing fields are accessed, this vulnerability may be exploited in different ways. For example, reading the missing fields may allow information leakage. Similarly, writing to the missing fields corrupts memory, enabling more involved exploits (e.g., denial-of-service or code execution). Although our fuzzer only triggered an out-of-bounds read, an out-of-bounds write may also be possible.

```
1  struct usb_descriptor_header {
2    __u8  bLength;
3    __u8  bDescriptorType;
4  } __attribute__ ((packed));
5  struct usb_otg_descriptor {
6    __u8  bLength;
7    __u8  bDescriptorType;
8    __u8  bmAttributes;
9  } __attribute__ ((packed));
10 int __usb_get_extra_descriptor(char *buffer,
       unsigned size, char type, void **ptr) {
11   struct usb_descriptor_header *header;
12   while (size >= sizeof(struct
       usb_descriptor_header)) {
13     header = (struct usb_descriptor_header *)
       buffer;
14     if (header->bLength < 2) {
15       printk("%s: bogus descriptor...\n", ...)
16     }
17     if (header->bDescriptorType == type) {
18       *ptr = header;
19       return 0;
20     }
21     buffer += header->bLength;
22     size -= header->bLength;
23   }
24   return -1;
25 }
26 static int usb_enumerate_device_otg(struct
       usb_device *udev) {
27   // ......
28   struct usb_otg_descriptor *desc = NULL;
29   err=__usb_get_extra_descriptor(udev->
       rawdescriptors[0], le16_to_cpu(udev->config
       [0].desc.wTotalLength), USB_DT_OTG, (void
       **) &desc);
30   if (err||!(desc->bmAttributes & USB_OTG_HNP))
31     return 0;
32   // ......
33 }
```

Listing 3: Out-of-bounds vulnerability in the Linux USB core framework. The two byte descriptor (`0x02`, `USB_DT_OTG`) is accepted by `__usb_get_extra_descriptor` as three byte `usb_otg_descriptor`. Triggering an out-of-bounds access when the missing field `bmAttributes` is accessed at line 30.

6.6 Fuzzing other peripheral interfaces

Peripheral interfaces represent a challenging attack surface. USBFuzz is extensible to other peripheral interfaces supported by QEMU. To add support for a new peripheral interface in USBFuzz, an analyst needs to: (i) implement a fuzzing device for the interface and adapt its reading operations to forward fuzzer generated data to the driver under test; (ii) adapt the fuzzer to start/stop a test by attaching/detaching the new fuzzing device to the VM; and (iii) adapt the user mode agent component to detect the end of tests based on the kernel log.

The SD card [3] is an interface that is well supported by QEMU and exposes a similar security threat as USB. SD cards are common on many commodity PCs and embedded devices. We extended USBFuzz to implement SD card driver fuzzing. The implementation required few code changes: 1,000 LoC to implement the fuzzing device, 10 LoC to adapt the fuzzer, and 20 LoC to adapt the user-mode agent.

After adapting, we fuzzed the SD card interface for 72 hours. As the SD protocol is much simpler than USB (with fixed commands and lengths), and there are only a limited number of drivers, we did not discover any bugs after running several fuzzing campaigns on Linux and Windows.

7 Related Work

In this section, we discuss related work that aims at securing/protecting host OS from malicious devices.

Defense Mechanisms. As an alternative to securing kernel by finding and fixing bugs, defense mechanisms stop active exploitation. For example, Cinch [1] protects the kernel by running the device drivers in an isolated virtualization environment, sandboxing potentially buggy kernel drivers and sanitizing the interaction between kernel and driver. SUD [5] protects the kernel from vulnerable device drivers by isolating the driver code in userspace processes and confining its interactions with the device using IOMMU. Rule-based au-

Tools	Cov	Data Inj	HD Dep	Portability
TTWE	✗	Device	✓	✓
vUSBf	✗	Device	✗	✓
umap2	✗	Device	✓	✓
usb-fuzzer	✓	API	✗	✗
USBFuzz	✓	Device	✗	✓

Table 4: A comparison of USBFuzz with related tools. The "Cov" column shows support for coverage-guided fuzzing. The "Data Inj" column indicates how data is injected to drivers: through the device interface (Device) or a modified API at a certain software layer (API). The "HD Dep" and "Portability" columns denote hardware dependency and portability across different platforms.

thorization policies (e.g., USBGuard [45]) or USB Firewalls (e.g., LBM [59] and USBFILTER [60]) work by blocking known malicious data packets from the device side.

Cinch [1] and SUD [5] rely heavily on hardware support (e.g., virtualization and IOMMU modules). Though their effectiveness has been demonstrated, they are not used due to their inherent limitations and complexities. Rule-based authorization policies or USB Firewalls may either restrict access to only known devices, or drop known malicious packets, thus they can defend against known attacks but potentially miss unknown attacks. These mitigations protect the target system against exploitation but do not address the underlying vulnerabilities. USBFuzz secures the target systems by discovering vulnerabilities, allowing developers to fix them.

Testing Device Drivers. We categorize existing device driver fuzzing work along several dimensions: support for coverage-guided fuzzing, how to inject fuzzed device data into tested drivers, and hardware dependency and portability across platforms. Support of coverage-guided fuzzing influences the effectiveness of bug finding, and the approach to inject device data into target drivers determines the portability. Hardware dependency incurs additional hardware costs.

Table 4 summarizes related work. Tools such as TTWE [66] and umap2 [18] depend on physical devices and do not support coverage-guided fuzzing. While eliminating hardware dependency through an emulated device interface for data injection, vUSBf [49] does not support coverage-guided fuzzing. usb-fuzzer [14] (a syzkaller [16] extension) supports coverage-guided fuzzing, and passes the fuzzer generated inputs to device drivers through extended system calls. However, its implementation depends on modifications to modules (the gadgetfs [42] and dummy-hcd [51] modules) in the USB stack of the Linux kernel, and is thus not portable. In contrast, USBFuzz is portable across different platforms and integrates coverage feedback (whenever the kernel exports it).

Sylvester Keil et al. proposed a fuzzer for WiFi drivers based on an emulated device [24]. While they also emulate a device, their system does not support coverage-guided fuzzing. They focus on emulating the functions of a single WiFi chip (the Atheros AR5212 [28]). As the hardware and firmware are closed source, they reverse engineered the necessary components. USBFuzz, in comparison, does not require reverse engineering of firmware and supports all USB drivers in the kernel. In concurrent work, PeriScope [50] proposes to apply coverage-guided fuzzing on WiFi drivers by modifying DMA and MMIO APIs in the kernel. IoTFuzzer [7] targets memory vulnerabilities in the firmware of IoT devices. These tools either have additional dependencies on physical devices, or cannot leverage coverage feedback to guide their fuzzing. Additionally, the AVATAR [71] platform enables dynamic analysis of drivers by orchestrating the execution of an emulator with the real hardware.

Symbolic Execution. The S2E [8] platform adds selective symbolic execution support to QEMU. Several tools extend S2E to analyze device drivers by converting the data read from the device side into symbolic values (e.g, SymDrive [47] and DDT [30]). Potus [38] similarly uses symbolic execution to inject faulty data into USB device drivers.

Like our approach, symbolic execution eliminates hardware dependencies. However, it is limited by high overhead and scalability due to path explosion and constraint solving cost. Further, Potus is controlled by operations from userspace, thus *probe* routines are out of scope. In contrast, USBFuzz follows a dynamic approach, avoiding these limitations and targets both *probe* routines and function routines.

8 Conclusion

The USB interface represents an attack surface, through which software vulnerabilities in the host OS can be exploited. Existing USB fuzzers are inefficient (e.g., dumb fuzzers like vUSBf), not portable (e.g., syzkaller usb-fuzzer), and only reach probe functions of drivers. We propose USBFuzz, a flexible and modular framework to fuzz USB drivers in OS kernels. USBFuzz is portable to fuzz USB drivers on different OSes, leveraging coverage-guided fuzzing on Linux and dumb fuzzing on other kernels where coverage collection is not yet supported. USBFuzz enables broad fuzzing (targeting the full USB subsystem and a wide range of USB drivers) and focused fuzzing on a specific device's driver.

Based on the USBFuzz framework, we applied coverage-guided fuzzing (the state-of-art fuzzing technique) on the Linux kernel USB stack and drivers. In a preliminary evaluation on nine recent versions of the Linux kernel, we found 16 new memory bugs in kernels which have been extensively fuzzed. Reusing the generated seeds from the Linux campaign, we leverage USBFuzz for dumb fuzzing on USB drivers in the FreeBSD, MacOS and Windows. To date we have found one bug in FreeBSD, three bugs on MacOS and four bugs on Windows. Last, focusing on a USB webcam driver, we performed focused fuzzing and found another bug in the XHCI driver of the Linux kernel. So far we have fixed 11

new bugs and received 10 CVEs. USBFuzz is available at https://github.com/HexHive/USBFuzz.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their insightful comments. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868). This material is based upon work supported by ONR under Award No. ONR award N00014-18-1-2674 and by NSF under award number CNS-1801601.

References

[1] Sebastian Angel, Riad S Wahby, Max Howald, Joshua B Leners, Michael Spilo, Zhen Sun, Andrew J Blumberg, and Michael Walfish. Defending against Malicious Peripherals with Cinch. In *USENIX Security Symposium*, pages 397–414, 2016.

[2] MIDI Association. Basics of USB-MIDI. https://www.midi.org/articles-old/basic-of-usb, 2018.

[3] SD Association. SD Standard overview. https://www.sdcard.org/developers/overview/, 2020.

[4] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2005.

[5] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating Malicious Device Drivers in Linux. In *USENIX annual technical conference*. Boston, 2010.

[6] Hartley Brody. USB Rubber Ducky Tutorial: The Missing Quickstart Guide to Running Your First Keystroke Payload Hack. https://blog.hartleybrody.com/rubber-ducky-guide/, 2017.

[7] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *NDSS*, 2018.

[8] 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.

[9] Catalin Cimpanu. List of 29 Different Types of USB Attacks. https://www.bleepingcomputer.com/news/security/heres-a-list-of-29-different-types-of-usb-attacks/, 2019.

[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*, pages 2123–2138. ACM, 2017.

[11] die.net. lsusb(8) - linux man page. https://linux.die.net/man/8/lsusb, 2018.

[12] Ellisys. Explorer 200 - Hardware trigger. https://www.ellisys.com/products/usbex200/trigger.php.

[13] GoodFET. Goodfet-facedancer21. http://goodfet.sourceforge.net/hardware/facedancer21/, 2018.

[14] Google. External usb fuzzing for linux kernel. https://github.com/google/syzkaller/blob/master/docs/linux/external_fuzzing_usb.md, 2018.

[15] Google. Found linux kernel usb bug. https://github.com/google/syzkaller/blob/usb-fuzzer/docs/linux/found_bugs_usb.md, 2018.

[16] Google. Syzkaller - kernel fuzzer. https://github.com/google/syzkaller, 2018.

[17] Google. KASAN-Linux usb-fuzzer. https://github.com/google/kasan/tree/usb-fuzzer, 2020.

[18] NCC Group. Umap2. https://github.com/nccgroup/umap2.

[19] NCC Group. AFL/QEMU fuzzing with full-system emulation. https://github.com/nccgroup/TriforceAFL, 2018.

[20] Intel. Intel virtualization technology. https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html, 2018.

[21] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *Symposium on Security and Privacy*. IEEE, 2019.

[22] Dave Jones. Trinity: Linux system call fuzzer. https://github.com/kernelslacker/trinity, 2018.

[23] Samy Kamkar. USBdriveby: Exploiting USB in style. http://samy.pl/usbdriveby/, 2014.

[24] Sylvester Keil and Clemens Kolbitsch. Stateful fuzzing of wireless device drivers in an emulated environment. *Black Hat Japan*, 2007.

[25] Linux kernel document. The kernel address sanitizer (kasan). https://www.kernel.org/doc/html/v4.12/dev-tools/kasan.html, 2018.

[26] Linux kernel document. Kerneladdresssanitizer. https://github.com/google/kasan/wiki, 2018.

[27] David Kierznowski. BadUSB 2.0: Exploring USB Man-In-The-Middle Attacks, 2015.

[28] knowledge base. ar5212. https://whirlpool.net.au/wiki/ar5212.

[29] Andrey Konovalov. CVE-2016-2384: Exploiting a double-free in the USB-MIDI Linux kernel driver. https://xairy.github.io/blog/2016/cve-2016-2384, 2016.

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

[31] Jon Larimer. Beyond Autorun: Exploiting vulnerabilities with removable storage. https://media.blackhat.com/bh-dc-11/Larimer/BlackHat_DC_2011_Larimer_Vulnerabiliters_w-removeable_storage-wp.pdf, 2011.

[32] LXR. Linux source code. https://elixir.bootlin.com/linux/latest/source/drivers/media/mc/mc-entity.c#L666, 2020.

[33] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. Fuzzing: the state of the art. Technical report, DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION EDINBURGH (AUSTRALIA), 2012.

[34] Microsoft. Lifecam vx-800. https://www.microsoft.com/accessories/en-us/d/lifecam-vx-800, 2018.

[35] Microsoft. Microsoft security bulletin ms17-011 - critical. https://docs.microsoft.com/en-us/security-updates/SecurityBulletins/2017/ms17-011, 2018.

[36] Nir Nissim, Ran Yahalom, and Yuval Elovici. USB-based attacks. *Computers & Security*, 70:675–688, 2017.

[37] NVD. Common vulnerabilities and exposures:cve-2016-2384. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2384, 2018.

[38] James Patrick-Evans, Lorenzo Cavallaro, and Johannes Kinder. POTUS: Probing Off-The-Shelf USB Drivers with Symbolic Fault Injection. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[39] Alex Plaskett. Biting the Apple that Feeds you. https://labs.mwrinfosecurity.com/assets/BlogFiles/mwri-44con-biting-the-apple-that-feeds-you-2017-09-25.pdf, 2018.

[40] Alex Plaskett. MacOS Kernel Fuzzer. https://github.com/mwrlabs/OSXFuzz, 2018.

[41] NSA Playset. TURNIPSCHOOL NSA Playset. http://www.nsaplayset.org/turnipschool, 2019.

[42] Matt Porter. Kernel USB Gadget Configfs Interface. https://events.static.linuxfound.org/sites/events/files/slides/USB%20Gadget%20Configfs%20API_0.pdf, 2018.

[43] Spice Project. usbredir. https://www.spice-space.org/usbredir.html, 2018.

[44] Syzkaller Project. report.go. https://github.com/google/syzkaller/blob/master/pkg/report/report.go.

[45] USBGuard project. USB Guard. https://usbguard.github.io/, 2018.

[46] USB/IP Project. USB/IP PROJECT. http://usbip.sourceforge.net/.

[47] Matthew J Renzelmann, Asim Kadav, and Michael M Swift. SymDrive: testing drivers without devices. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 279–292, 2012.

[48] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for OS kernels. In *Usenix Security Symposium*, 2017.

[49] Sergej Schumilo, Ralf Spenneberg, and Hendrik Schwartke. Don't trust your usb! how to find bugs in usb device drivers. *Blackhat Europe*, 2014.

[50] 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 *2019 Network and Distributed Systems Security Symposium (NDSS)*. Internet Society, 2019.

[51] Alan Stern. Dummy/Loopback USB host and device emulator driver. https://elixir.bootlin.com/linux/v3.14/source/drivers/usb/gadget/dummy_hcd.c, 2018.

[52] syzkaller team. Found bugs by syzkaller in bsd. https://github.com/google/syzkaller/blob/master/docs/openbsd/found_bugs.md, 2018.

[53] syzkaller team. Found bugs by syzkaller in linux. https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs.md, 2018.

[54] syzkaller team. vusb ids. https://github.com/google/syzkaller/blob/usb-fuzzer/sys/linux/vusb_ids.txt, 2018.

[55] syzkaller team. vusb.txt at google/syskaller. https://github.com/google/syzkaller/blob/master/sys/linux/vusb.txt, 2018.

[56] QEMU team. Device specification for inter-vm shared memory device. https://github.com/qemu/qemu/blob/master/docs/specs/ivshmem-spec.txt, 2018.

[57] Qemu Team. Qemu: the fast! processor emulator. https://www.qemu.org/, 2018.

[58] Dave Jing Tian, Adam Bates, and Kevin Butler. Defending against malicious USB firmware with GoodUSB. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 261–270, 2015.

[59] Dave Jing Tian, Grant Hernandez, Joseph I Choi, Vanessa Frost, Peter C Johnson, and Kevin RB Butler. LBM: A Security Framework for Peripherals within the Linux Kernel. In *LBM: A Security Framework for Peripherals within the Linux Kernel*, page 0, 2019.

[60] Dave Jing Tian, Nolen Scaife, Adam Bates, Kevin Butler, and Patrick Traynor. Making USB Great Again with USBFILTER. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 415–430, 2016.

[61] Jing Tian, Nolen Scaife, Deepak Kumar, Michael Bailey, Adam Bates, and Kevin Butler. SoK:" Plug & Pray" Today–Understanding USB Insecurity in Versions 1 Through C. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 1032–1047. IEEE, 2018.

[62] Linus Torvalds. Kernel parameters. https://github.com/torvalds/linux/blob/master/Documentation/admin-guide/kernel-parameters.txt, 2018.

[63] Ubuntu. Package: streamer (3.103-3build1). https://packages.ubuntu.com/xenial/streamer, 2019.

[64] Linux USB. A list of usb id's. http://www.linux-usb.org/usb.ids, 2019.

[65] usbkill.org. Official usb killer site. https://usbkill.com/, 2019.

[66] Rijnard Van Tonder and Herman A Engelbrecht. Lowering the USB Fuzzing Barrier by Transparent Two-Way Emulation. In *WOOT*, 2014.

[67] Dmitry Vyukov. kernel: add kcov code coverage. https://lwn.net/Articles/671640/, 2018.

[68] Wikipedia. Extensible Host Controller Interface. https://en.wikipedia.org/wiki/Extensible_Host_Controller_Interface, 2018.

[69] Wikipedia. USB On-The-Go. https://en.wikipedia.org/wiki/USB_On-The-Go, 2018.

[70] Wikipedia. Wireless USB. https://en.wikipedia.org/wiki/Wireless_USB, 2019.

[71] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *NDSS*, 2014.

[72] Michal Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/, 2017.

[73] Michal Zalewski. The bug-o-rama trophy case. http://lcamtuf.coredump.cx/afl/#bugs, 2017.

[74] Google Project Zero. Notes on Windows Uniscribe Fuzzing. https://googleprojectzero.blogspot.com/2017/04/notes-on-windows-uniscribe-fuzzing.html, 2018.