FUZZUSB: Hybrid Stateful Fuzzing of USB Gadget Stacks

Kyungtae Kim[†], Taegyu Kim[§], Ertza Warraich[†], Byoungyoung Lee[¶],
Kevin R. B. Butler[‡], Antonio Bianchi[†], Dave (Jing) Tian[†]

[†]Purdue University, [§]The Pennsylvania State University, [¶]Seoul National University, [‡]University of Florida
[†]{kim1798,ewarraic,antoniob,daveti}@purdue.edu, [§]tgkim@psu.edu, [¶]byoungyoung@snu.ac.kr, [‡]butler@ufl.edu

2022 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-1316-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/SP46214.2022.9833593

***Abstract*—**

Universal Serial Bus (USB) is the de facto protocol supported by peripherals and mobile devices, such as USB thumb drives and smart phones. For many devices, USB Type-C ports are the primary interface for charging, file transfer, audio, video, etc. Accordingly, attackers have exploited different vulnerabilities within USB stacks, compromising host machines via BadUSB attacks or jailbreaking iPhones from USB connections. While there exist fuzzing frameworks dedicated to USB vulnerability discovery, all of them focus on USB host stacks and ignore USB gadget stacks, which enable all the features within modern peripherals and smart devices.

In this paper, we propose FUZZUSB, the first fuzzing framework for the USB gadget stack within commodity OS kernels, leveraging static analysis, symbolic execution, and stateful fuzzing. FUZZUSB combines static analysis and symbolic execution to extract internal state machines from USB gadget drivers, and uses them to achieve state-guided fuzzing through multi-channel inputs. We have implemented FUZZUSB upon the *syzkaller* kernel fuzzer and applied it to the most recent mainline Linux, Android, and FreeBSD kernels. As a result, we have found 34 previously unknown bugs within the Linux and Android kernels, and opened 8 CVEs. Furthermore, compared to the baseline, FUZZUSB has also demonstrated different improvements, including 3× higher code coverage, 50× improved bug-finding efficiency for Linux USB gadget stacks, 2× higher code coverage for FreeBSD USB gadget stacks, and reproducing known bugs that could not be detected by the baseline fuzzers. We believe FUZZUSB provides developers a powerful tool to thwart USB-related vulnerabilities within modern devices and complete the current USB fuzzing scope.

I. INTRODUCTION

Universal Serial Bus (USB) [1] is the de facto protocol for a wide range of peripheral devices and smart devices such as mice, keyboards, external Flash/SSD storages, GPU docks, smart phones, tablets, etc. With the increasing adoption of USB Type-C [2], we have seen laptops, smart phones, and watches with only USB Type-C ports equipped for charging, file transfer, audio, video, etc. While the prevalence and versatility of USB have made our daily life convenient, it has also attracted attackers seeking to exploit vulnerabilities within the USB ecosystem. Traditional USB attacks use USB storage devices to carry malware and break into the air-gap environment, e.g., Stuxnet [3]. Modern USB attacks target flaws within the USB specifications and stack implementations, including injecting malicious USB functionality into USB device firmware to compromise host machines in BadUSB attacks [4] and exploiting USB connections to unlock screens in Android [5] or jailbreak iPhones [6].

Meanwhile, *fuzzing* has become a popular runtime testing method, given its effective bug-finding capabilities. Fuzzing has been used successfully in various domains [7–13] to reveal large numbers of bugs. For instance, a number of fuzzers have been proposed for the USB domain, ranging from hardware fuzzers to pure software-based fuzzers [14–19]. These tools have already helped detect real-world USB vulnerabilities. For example, the state-of-the-art kernel fuzzer *syzkaller* has been extended to support USB fuzzing, and found over 100 bugs within the Linux kernel USB subsystem [20].

Currently, all existing USB fuzzing efforts assume that threats stem from malicious USB peripherals, and therefore focus on defending host machines from peripheral-based attacks by fuzzing the USB host stack. However, modern devices such as smart phones and tablets often contain another USB stack – the *USB gadget stack*, which is used to support well-known functionalities including charging, mass storage, tethering, MIDI, PTP/MTP, etc. For instance, the Linux USB gadget stack is used to support billions of embedded systems and Android devices. Unfortunately, no existing fuzzing framework is able to detect vulnerabilities within the USB gadget stack.

In addition, existing works do not fully consider the statefulness and limit input space of the USB protocol (i.e., single-channel fuzzing for USB host stacks). As a result, existing works either suffer from detecting only shallow bugs without being able to find deeper bugs (e.g., data communication only occurring after USB enumeration) [17, 18] or require significant manual efforts to enable stateful fuzzing [16]. For this reason, most of the USB bugs (e.g., reported by syzkaller) are within driver initialization functions rather than getting into the core USB logic [14, 20]. While syzkaller can fuzz many different USB host drivers (by testing different vendor IDs and product IDs), it does not try to explore the different states a specific driver can reach. Consequently, the inputs it produces are unlikely to reach deep code locations related to a driver's core logic.

To address the limitations of existing USB fuzzing approaches, in this paper, we propose FUZZUSB, the first *stateful* USB fuzzer targeting the Linux and FreeBSD USB gadget

DOI 10.1109/SP46214.2022.00037

stacks. Unlike typical USB host fuzzers that mainly accept mutation inputs from USB peripherals, FUZZUSB enables multi-channel input mutations to reach different parts of USB gadget code based on the statefulness of USB communications. Furthermore, we consider the stateful behavior of different USB gadget functionality, For that, FUZZUSB leverages static analysis and symbolic execution to extract internal state machines of each USB gadget driver and uses them to guide fuzzing, unlike the previous work which simply relies on selective symbolic execution to guide fuzzing [12]. Furthermore, FUZZUSB allows for state coverage and transition coverage as feedback, in addition to the classic code coverage, as well as different mutation rules to support different fuzzing strategies.

We have implemented FUZZUSB by rearchitecting and customizing syzkaller [16], evaluated it on the latest Linux, Android, and FreeBSD kernels, and discovered *34 previously unknown vulnerabilities with 8 CVEs assigned*. Compared to the baseline gadget fuzzer we implemented following a straw-man approach without multi-channel input mutation or state awareness, FUZZUSB exhibits multiple improvements of USB fuzzing, including 3× higher code coverage, 50× improved bug-finding efficiency for Linux gadget stacks, 2× improved bug-finding efficiency for FreeBSD gadget stacks, and reproducing known bugs that could not be detected by the baseline fuzzers. We have reported all our findings to the corresponding parties, and open-source FUZZUSB to facilitate USB security research in the community [21].

The key contributions of this paper are as follows.

- Unlike exiting USB fuzzing tools, FUZZUSB is the first USB fuzzing framework targeting the USB gadget stack, supporting both multi-channel input mutations and state-guided fuzzing.
- We combine both static analysis and symbolic execution to design an algorithm to extract internal state machines from different USB gadget drivers automatically, which are used by FUZZUSB to achieve stateful fuzzing.
- We have applied FUZZUSB to the most recent Linux kernel and Android kernels and discovered *34 previously unknown vulnerabilities with 8 CVEs assigned*. Compared with the baseline, FUZZUSB has demonstrated multiple improvements of USB fuzzing, including 3× higher code coverage, 50× improved bug-finding efficiency for Linux gadget stacks, 2× improved bug-finding efficiency for FreeBSD gadget stacks, and reproducing known bugs that could not be detected by the baseline fuzzers.

II. BACKGROUND

USB is a master-slave protocol where a USB host connects with at least one USB peripheral. A host usually refers to a desktop or a laptop in the master role and controlling connected peripherals. A peripheral could be a USB keyboard, a USB thumb drive, or even an Android phone, acting in the slave role within the USB communication and accepting a host's commands. Although supporting the same USB specifications, a USB stack implementation within a host is different from the one within a peripheral. The former is called the USB host stack, and the latter is named the USB gadget stack.[1]

Compared to the typical USB peripherals, USB smart devices (such as smartphones and tablets) are usually equipped with USB device controllers having either the On-The-Go (OTG) [22] or the Dual-Role-Device (DRD) [23] capabilities. These features enable them to behave as both a USB host and a USB peripheral depending on different usage scenarios. For instance, when a USB keyboard is connected to an Android phone, the Android phone acts as a USB host. However, when the same Android phone connects to a laptop to transfers pictures, the phone acts as a USB peripheral. Accordingly, these USB smart devices contain two different USB stack implementations to achieve different roles during USB communication.

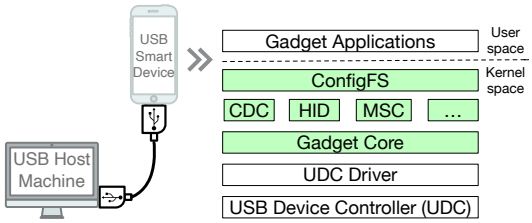


Fig. 1: USB gadget stack within a USB smart device, e.g., Android phones, connected with a USB host machine, e.g., a desktop. The USB gadget stack includes the gadget core, different USB gadget drivers such as CDC, HID, MSC, etc., and the corresponding ConfigFS filesystem.

A. USB Gadget Stack

Figure 1 shows an example of a smart device using the USB gadget stack within the Linux kernel. At the bottom is the USB device controller (UDC) providing the USB physical layer supporting either OTG or DRD. A corresponding UDC kernel driver, e.g., `fsl_usb2`, is usually needed to communicate with UDC, e.g., Freescale Highspeed USB Dual-Role Device Controller. The gadget core sits right above the UDC driver exposing APIs to a variety of USB gadget drivers, which implements different USB functionality instantiated via the UDC hardware. Example gadget drivers include Communication Data Class (CDC) [24] (for data communication functionality, like a modem), Human Interface Device (HID) [25] (for input functionality, like a keyboard), and Mass Storage Class (MSC) [26] (for storage functionality, like a USB thumb drive). The ConfigFS [27] filesystem lies above all the USB gadget drivers allowing specific user-space applications to configure the USB smart device with one or more classes of USB functionality. For Android devices, this usually refers to the USB configuration menu to change the setting of the phone, e.g., charging or Media Transfer Protocol (MTP).

In general, between a USB host machine and a USB smart device using the USB gadget stack, the USB connection and communication workflow includes three phases. In the first

[1]It is also called as USB peripheral stack or USB device stack in certain documentations.

phase, the *configuration* phase, an owner of the USB smart device decides what USB functionality will be exposed to the host machine by running the corresponding user-space gadget application, setting the expected device information (e.g., VID/PID) by writing into ConfigFS, and triggering the corresponding gadget driver (e.g., HID) to instantiate the functionality using the UDC hardware. For instance, USB tethering will activate the CDC USB gadget driver and turn smart phone into a USB modem.

Once the USB smart device is plugged into the USB host machine, the second phase, the *enumeration* phase, starts. This phase consists of a standard procedure defined by the USB specification to retrieve the device information from the remote device in the format of "descriptors". Following the previous example, the USB host machine sends out different "GetDescriptor" USB requests to the USB smart device, which in turn responds with all the information needed for the USB host to recognize a connected USB modem.

Finally, the USB host loads the matching device driver within the host operating system to enable the USB smart device's specific USB gadget functionality in the *communication* phase. Continuing the example, the host might load the CDC USB host driver to serve the USB modem as a typical USB peripheral, allowing the host to connect to the Internet using the USB smart device.

Bugs in either USB host stacks or USB gadget stacks could have critical security implications. Since USB stacks usually run in the kernel space, any vulnerabilities within these stacks could lead to privilege escalation or arbitrary code execution, compromising the whole system. Meanwhile, USB gadget stacks expose larger attack surfaces compared to USB host stacks. For instance, USB host stacks usually only need to consider malicious inputs from USB peripherals. In contrast, a malicious app with USB permissions could reconfigure the USB gadget functionality at any time from the user space; an untrusted host machine could send out malformed USB packets to USB gadget stacks from the kernel space. The local app and the remote host could even conspire together against the USB gadget stack, as we will show later in the paper. Due to the wide adoption of mobile and IoT devices, a vulnerability within USB gadget stacks could impact the security of billions of devices in use, and thus it is usually rated as "high" severity.

B. Security Model

In this paper, we assume that all hardware within a USB smart device is trusted including the UDC. We also trust the code running in the kernel space (or privileged mode) within the USB smart device, where a USB gadget stack usually resides, such as the Linux kernel USB gadget stack. Taking an Android device as a concrete example, we trust the System-on-Chip (SoC) device and the Linux kernel running within Android.

We assume two different forms of adversaries. First, a malicious gadget application running in the user space of the USB smart device may exploit the ConfigFS interface to reach any internal vulnerabilities within the USB gadget stack (i.e., a top-down approach). Second, a malicious USB host device may send out malformed or malicious USB packets to exploit the USB gadget stack of the USB smart device (i.e., a bottom-up approach).

Following the Android example, a malicious Android app may try to exploit the USB gadget stack running inside the kernel space to achieve privilege escalation, while a malicious desktop device may send out malicious USB packets to steal secret information from the attached device without a user's permission. It is also possible for adversaries to leverage both exploitation directions together to enable more sophisticated attacks, as shown in the PoC exploit in Appendix C.

Technique	Fuzz Target	Fuzz Scope	Fuzz Chan	Device Dep	State Mach
FaceDancer [19]	host	E only	device	HW	N
POTUS [15]	host	C only	host	SW	N
vUSBf [17]	host	E only	device	SW	N
umap2 [18]	host	E only	device	Hybrid	N
syzkaller [16]	host	E (C partially)	device	SW	N
USBFuzz [14]	host	E (C partially)	device	SW	N
FUZZUSB	gadget	E+C+CC	host+device	SW	Y

TABLE I: State-of-the-art USB fuzzers. E and C represent the enumeration and communication phases respectively while CC denotes the configuration phase. HW represents the need for dedicated hardware; SW relies on software only; Hybrid means supporting both HW and SW.

III. MOTIVATION

Modern peripherals and smart devices have been a popular target of recent attacks. Within the large attack surface of these devices, the USB gadget stack is a valuable and high-profile target for the following reasons. First, since the USB gadget stack supports versatile USB features as discussed in §II, it offers prevalence and large attack surfaces — ranging from charging and storage to MIDI, etc. Second, the USB gadget stack usually runs with high privilege (i.e., the kernel privilege). Lastly, such attacks can be carried out with zero privilege on the victim device. In other words, unlike typical privilege escalation attacks, attacks targeting a USB gadget stack usually do not assume a specific prior requirement, such as providing the right passcode or installing a custom malicious application. Instead, simply connecting a USB cable to the USB smart device satisfies all the attack requirements.

Surprisingly, despite the prevalence and large attack surface of the USB gadget stack, we found that little attention has been paid to this area especially in terms of fuzzing. Table I summarizes existing USB fuzzing tools, all of which focus on USB host stack fuzzing and assume a malicious USB peripheral trying to exploit vulnerabilities within a host using a USB connection. These fuzzers usually have limited coverage in handling the three phases of USB workflows. In particular, they usually target one single phase of USB communication (e.g., enumeration), mutating the input used by a single channel. In addition, some fuzzers, such as FaceDancer [19] and umap2 [18], require dedicated programmable USB hardware to generate malformed USB packets, limiting their scalability.

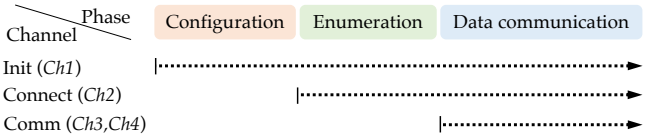


Fig. 2: Usage of the different USB gadget communication channels in a standard USB connection.

A naive approach to fuzz USB gadget stacks would be using existing fuzzers (e.g., usb-syzkaller) and switching the fuzzing direction from gadget-to-host to host-to-gadget.[2]. However, such an approach is unlikely to achieve significant code coverage or find bugs efficiently. In fact, as we will show in the rest of this section, to achieve both a wide breadth (e.g., fuzzing different phases) and a high depth (e.g., reaching deep USB core-logic code) in USB gadget stacks, a fuzzer has to tackle fundamental challenges that are specific to both the protocol design and the stack implementation.

Compared with the existing works, to the best of our knowledge, FUZZUSB is the first USB gadget stack *stateful* fuzzing framework, covering all three phases of USB connection and communication, and relying on software emulation to achieve bug reproducibility and scalability.

A. Challenge 1: Multi-channel Inputs

As mentioned earlier and shown in Figure 2, there are three phases involved in the lifetime of a USB gadget driver: configuration, enumeration, and communication. Accordingly, each phase has its own input space and dedicated USB endpoints for data transfer. These endpoints are the basic communication units defined by the USB specification, and they are indexed by numbers (e.g., endpoint 0). In the rest of this paper, we call them *input channels*. As shown in Figure 2, there are three input channels, *init*, *control*, and *data*, which are used in different phases.

The init channel (`Ch1`) refers to the ConfigFS interface exposed to gadget applications, providing the USB device information to instantiate the corresponding USB product (functionality), using the UDC during the configuration phase. The control channel (`Ch2`) represents the USB control transfer, using endpoint 0 during the enumeration phase. The data channel (`Ch3`, `Ch4`, etc.) groups all the possible USB data transfers using different endpoints during the communication phase, after a USB connection is established. Note that the data channel could contain multiple sub-channels for data transfer depending on the complexity of the USB functionality exposed by a device. For instance, a USB headset might contain 3 data transfer sub-channels, one for the microphone, one for the speakers, and one for volume control.

It is noteworthy that we need to carefully address these multi-channel inputs across different phases, in terms of the mutation strategy used for USB gadget fuzzing. For example, mutating the data channel during the enumeration phase would only waste computing resources due to the inactivity of the communication phase. Similarly, even though both the init and control channels are available during the communication phase, mutating the control channel to trigger a new USB enumeration process might not be desired for fuzzing, since it breaks the current USB connection while mutating the init channel could trigger race conditions among different parts of gadget driver code as we will see later. As a result, *when to fuzz each channel* and *how to mutate the input targeting each channel* determine the effectiveness and efficiency of fuzzing the USB gadget stack.

[2]We will compare FUZZUSB with this naive approach as the baseline in our evaluation.

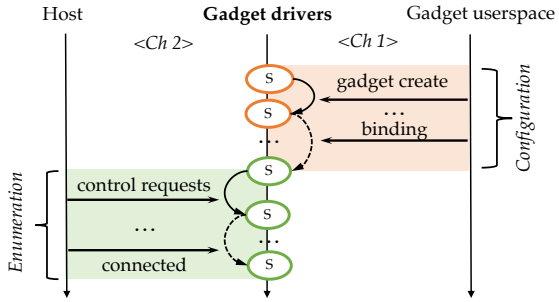


Fig. 3: Standard message exchange during the configuration and enumeration phases. Circles in color represent feasible gadget states along with their transitions.

Unfortunately, existing fuzzing techniques hardly tackle these essential issues. As shown in Table I, state-of-the-art USB host stack fuzzing only considers single-channel input, e.g., the device channel, since USB host stacks are usually not configurable from the user space, and the system call interface is too generic to fuzz USB host stacks directly. Similarly, USB host stack fuzzers focus on a specific phase, e.g., the enumeration phase, aiming at triggering bugs from different USB device drivers rather than detecting deep bugs within a driver. As we will discuss in §IV, we maximize the effectiveness and efficiency of USB gadget stack fuzzing by considering multi-channel inputs using a proper mutation strategy, based on the current phase and state of the driver.

B. Challenge 2: Statefulness

In addition to the multiple phases of the USB communication, each USB gadget driver also implements a fine-grained state machine internally. Figure 3 illustrates the standard message exchanges from the perspective of gadget drivers during the configuration and enumeration phases. To configure the functionality of a USB gadget, a user-mode configuration process (e.g., a gadget-specific application) within the USB smart device delivers a sequence of setup data to the driver in an ordered way up until binding. In the following enumeration, the USB host communicates with the gadget driver and maintains states internally via exchanging different messages (USB requests and responses) in a specific order. While each gadget driver follows the same state changes within the first two phases, different gadgets implement different state machines for the communication phase.

```
1  /* accept a SCSI command */
2  int get_next_cmd(struct fsg_common *com) {
3    ...
4    // wait for SCSI command from the host
5    recv_data_from_host (com->buf, ..) // Ch3: transition point
6    ...
7    // check validity of the received SCSI command
8    if (com->buf->cmd_size != BULK_CB_WRAP_LEN ||
9        com->buf->signature != BULK_CB_SIGN)
10     return -1;
11   ...
12 }
13 /* handle the SCSI command */
14 int do_scsi_cmd(struct fsg_common *com) {
15   ...
16   // process received SCSI command
17   if (com->buf->cmd == WRITE) {
18     // wait for actual data payload from the host
19     recv_data_from_host (com->buf, ..) // Ch3: transition point
20     ...
21     // write the received data into backing store
22     kernel_write(buf)
23   }
24   else if (com->buf->cmd == READ) {
25     // read the data from backing store
26     kernel_read (buf)
27     ...
28     // send the data to the host, and expect valid ack
29     send_data_to_host (buf, ..) // Ch3: transition point
30     if (fail to receive ack)
31       return -1;
32   }
33   ...
34 }
35 /* SCSI command handling function in mass storage gadget */
36 while (...) {
37   if (get_next_cmd(com))
38     continue;
39   if (do_scsi_cmd(com))
40     continue;
41   ...
42 }
```

Fig. 4: Simplified code used by the mass storage gadget.

Example: mass storage gadget driver. Figure 4 presents a simplified example of Small Computer System Interface (SCSI) communication, within the mass storage class (MSC) gadget driver. In this example, the gadget enters a loop and waits for a SCSI command from the host (line 35). Once received, it performs validity checks in `get_next_cmd()`, and then it starts to process the command message (line 37). Depending on the received SCSI command, the gadget might need to receive the following payload from the host, which could be data to be stored into the storage system afterward (line 17-23), or send data from the storage to the host (line 24-30).

Figure 5 depicts the corresponding state machine. According to the state machine, a sequence of state transitions can lead the exploration to deeper code locations (in the example, lines 22 and 29). To visit all the states present in the state machine, the host needs to feed different state-specific values to trigger the different state transitions. Without considering such a statefulness of the gadget driver, random input from the host side is unlikely to reach further states, causing the execution to terminate early (e.g., line 10). As a result, *how to extract the internal state machines within each gadget driver correctly and in a scalable fashion* and *how to leverage these state machines to achieve stateful fuzzing* determine the effectiveness and efficiency of fuzzing the USB gadget stack.

Unfortunately, existing fuzzing techniques hardly tackle the

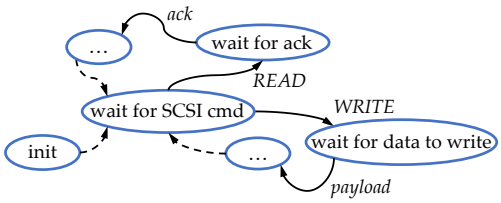


Fig. 5: Partial gadget state machine for the SCSI communication.

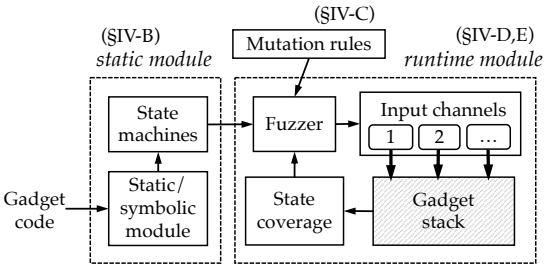


Fig. 6: Architecture of FUZZUSB.

statefulness of USB device drivers. As shown in Table I, most of the existing solutions do not consider the internal state machines within USB device drivers, due to either covering only one phase (e.g., enumeration) or having incomplete support for the communication phase. For the same reason, the state-of-the-art USB fuzzer [16], although having reported more than 100 bugs, can only find shallow bugs. In fact, most of the bugs it found are located in the driver initialization code [14, 20]. For instance, syzkaller is mostly based on pre-defined templates, which are manually written by domain experts replicating USB communications. This is a common approach to addressing stateful communication. However, manually written templates are neither generic nor scalable to cover a variety of internal state machines implemented by USB gadget stacks. As we will discuss in §IV, we maximize the effectiveness and efficiency of USB gadget stack fuzzing by guiding the fuzzer using new state-coverage and transition-coverage feedbacks, besides the typical code-coverage. To provide such new feedbacks, FUZZUSB automatically extracts internal state machines using static analysis and symbolic execution from each gadget driver. Note that the stateful fuzzing technique developed here is generic and can be applied to USB host stack fuzzing as well.

IV. DESIGN

To tackle the challenges of USB gadget fuzzing discussed in §III, we design FUZZUSB to support multi-channel input mutations and state-guided fuzzing targeting the USB gadget stack. In this section, we outline the overall design and workflow of FUZZUSB, and explain its key components in detail.

A. Approach Overview

Figure 6 depicts an overview of FUZZUSB. The inputs of FUZZUSB are USB gadget drivers within OS kernels (e.g., Linux kernel). FUZZUSB conducts both static and symbolic

analysis to extract internal state machines automatically from the source code of the gadget drivers (§IV-B). Static analysis not only finds program locations controlling the internal state machines of the drivers, but also facilitates the path prioritization of symbolic execution to avoid path explosion. After this step, symbolic execution follows the execution paths prioritized by the static analysis and extracts the internal state machines representing the functionality and behavior of the gadget drivers.

In its runtime module, FUZZUSB takes the extracted state machines and mutation rules as inputs to generate state-aware fuzzing inputs. The mutation rules guide a fuzzer to mutate inputs towards desired state transitions, and provide the flexibility to tune the mutation strategy as needed (§IV-C). During fuzzing, FUZZUSB further orchestrates the mutation generation and distribution to the multiple input channels (§IV-D). Meanwhile, coverage information, i.e., both the code coverage and the state coverage, is fed back to FUZZUSB driving mutations to generate the next-round fuzzing input (§IV-E).

B. Building State Machines

To realize state-guided fuzzing, we adopt the *finite state machine* (FSM) approach as the state representation. Instead of relying on system call dependencies (e.g., `open`-then-`write`) to infer the driver state (as implemented in related work [8–10, 28, 29]), we use a state machine, explicitly representing the possible internal states of the target programs (i.e., gadget code). This approach allows a fuzzer to know available states and transitions ahead of time and trigger state changes directly. In particular, our state machine helps the fuzzer figure out *what to fuzz in a given state* and *how to transition from one state to another*. To this end, we first introduce the definition of states and transitions in the context of USB gadget drivers. Next, we explain how we extract internal state machines from USB gadget drivers automatically by combining static analysis and symbolic execution.

1) States and Transitions: As discussed in §III-B, the effectiveness of stateful fuzzing depends on how well and faithfully we extract states and transitions from gadget code, which contains multiple `receive` and `send` operations reading from and writing to channels. We define *a new state for each code line in which one of these I/O operations (either receive or send) is executed*. Considering the example in Figure 4, they have three states, corresponding to the I/O operations happening at lines 5, 19, and 29. We then consider the gadget to be in a specific state based on the last I/O operation performed. Moreover, every time a new I/O operation is performed, we consider the gadget state transitioning from the current state to the state corresponding to the latest I/O operation executed. For instance, in Figure 4, after line 5 is executed, the gadget will transition to the next state. If line 19 is later executed, the gadget will transition to another state.

We call such an I/O operation a *transition point*. Based on our observation, transition points are identifiable in a generic way as they rely on standard USB gadget APIs (e.g., `usb_ep_queue()`) provided by the underlying USB gadget core subsystem in the OS kernels [30]. Moreover, all the gadgets follow the standard USB protocol for the first two communication phases (i.e., configuration and enumeration), thus sharing the same states and transitions. In the data communication phase, however, different gadget drivers implement different functionality (e.g., mass storage in §III), resulting in different states and transitions, and essentially different state machines.

2) State Machine Construction: Based on the insight above, we construct a state machine for each gadget, as described in Algorithm 1. At a high level, we follow a two-stage program analysis technique, including *static* and *symbolic analysis*, to obtain input values that trigger specific state transitions during fuzzing.

Static analysis. As the first step, FUZZUSB shortlists a number of code paths changing states, aiming at scaling the symbolic execution in the following step. To this aim, FUZZUSB performs static inter-procedural backward slicing [31]. In the gadget code, we keep track of all of the transition function calls, which are standard Linux USB gadget APIs (e.g., `usb_ep_queue()`). Given a USB gadget driver, we find out where a transition function is used, i.e., transition points (line 2 in Algorithm 1). From each transition point, FUZZUSB performs slicing in a backward direction along with the data- and control- dependent paths, until reaching the entry points of the gadget, which are usually dispatcher functions for input channels (line 3). We repeat this for all transition points, optimizing and leaving only execution paths that can lead to transitions. The result of the sliced driver is used as the target code for the next step of the analysis.

Symbolic analysis. To identify concrete fuzzing input values triggering transitions across states, we employ symbolic execution on the sliced gadget code. Our symbolic analysis attempts to obtain concrete input values to reach the transition points from either entry points or different transition points. Since gadget drivers may have multiple entry points due to multiple input channels, we carry out symbolic execution per entry point (lines 5-19). We first symbolically taint memory buffers that are controllable by input, then iterate over instructions from an entry point function. Meanwhile, we keep collecting path constraints and updating symbolic states. When reaching a transition point, we retrieve concrete values by solving the collected symbolic constraints (line 13). We use these values as a concrete input to trigger a transition between the current and next state according to the state machine of the gadget. After refreshing (removing) the constraints that have been used, we restart the symbolic execution from that transition point as a new starting point of the execution (line 18). We repeat the process every time the execution encounters a new transition point and terminate at the end of the entry point function.

In Figure 4, for example, the symbolic execution starts over when reaching a transition point at line 5 in `get_next_cmd()`, with new symbolic-tainted memory buffer `com->buf`. Along the path, the execution can reach another transition point

Path	Start	End	Data	Channel
1	line 5	line 19	WRITE (cmd)	Ch3
2	line 19	-	Payload	Ch3
3	line 5	line 29	READ (cmd)	Ch3
4	line 29	-	Ack	Ch3

TABLE II: Result example of the symbolic analysis.

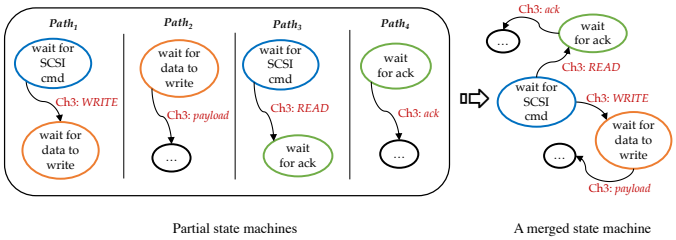


Fig. 7: State merging for mass storage gadget.

at line 19 in `do_scsi_cmd()`. We obtain the concrete values by solving the constraints `com->buf->cmd == WRITE`, which can lead to a state transition from the starting point (line 5) to another transition point (line 19). Table II summarizes the result of symbolic analysis for the example code, listing all feasible paths leading to possible state transitions.

State merging. Putting together all the collected results from the prior analysis, we finalize the state machine construction. We now have a set of *partial* state machines (i.e., S_p in Algorithm 1) from the symbolic execution results — for each execution of symbolic analysis, its start point and end point become *an entry state* and *an end state* respectively in a new partial state machine (lines 23-27 in Algorithm 1). At the same time, the concrete input values would become a transition trigger between the two states (line 28). By merging the same states (i.e., the end state of a state machine merges with the corresponding entry state of another), we connect all the partial state machines together. Repeating this procedure, we build a definitive state machine of the gadget (i.e., `S`).

Figure 7 illustrates the state merging for the motivating example in Figure 4. In each partial state machine, both states correspond to transition points, and the transition values stem from constraint solving. We connect all the partial state machines via merging the overlapping states while maintaining their transition values, building a final state machine.

In Figure 8, we take the `loopback` gadget as another case. In the example, the gadget simply sends the received data back to the host, then waits for another data. Using the generated state machines, we can first pinpoint both transition and entry points explicitly in the code. Next, we directly track the relationship between two relevant transitions and retrieve two states accordingly. Finally, FUZZUSB generates a finite state machine, i.e., State$_1$–($Ch3$:data-IN)–> State$_2$ –($Ch3$:data-OUT)–> State$_1$ .... Note that in each state, the host needs to feed appropriate inputs in the right order to transition to the next state. In this way, our static analysis makes the state machine construction automated and more scalable.

Algorithm 1 State Machine Construction.

Input: G - target gadget code
Input: TF - transition function

1: **function** STATICANAL (G, TF)
2: EP, TP ← Scanning (G) // scan all entry and transition points
3: G_s ← Slicing (G, EP, TP)
4: **return** EP, TP, G_s
5: **function** SYMBOLICANAL (EP, TP, G_s)
6: VP ← {} // initialize list of visited points
7: S_p ← {} // initialize a set of partial state machines
8: W ← {} // initialize worklist
9: W.push(EP)
10: **while** W ≠ ∅ **do**
11: P_{start} ← W.pop()
12: // do symbolic execution until termination or TP
13: list of {P_{end}, V} ← SymExec(P_{start}, TP)
14: **for each** {P_{end}, V} **in** list of {P_{end}, V} **do**
15: S_p = S_p ∪ {P_{start}, P_{end}, V}
16: **if** P_{end} ∉ VP **then**
17: VP = VP ∪ P_{end}
18: W.push(P_{end})
19: **return** S_p
20: **function** STATEMERGE (S_p)
21: VP ← {}
22: S ← {} // initialize a state machine
23: **for each** {P_{start}, P_{end}, V} **in** S_p **do**
24: **for each** P **in** {P_{start}, P_{end}} **do**
25: **if** P ∉ VP **then**
26: VP = VP ∪ P
27: AddState(P, S)
28: AddTransition(V, P_{start}, P_{end}, S)
29: **return** S
30: // entry point: global function calls
31: EP, TP, G_s ← STATICANAL (G, TF)
32: S_p ← SYMBOLICANAL (EP, TP, G_s)
33: S ← STATEMERGE (S_p)
34: **return** S
Output: S - a gadget state machine

```
1  /* entry point for data-IN */
2  int entry_IN() {
3      // request the host for data-OUT
4      request_to_host (OUT, ..) // Ch3: transition point
5  }
6  /* entry point for data-OUT*/
7  int entry_OUT() {
8      // request the host for data-IN
9      request_to_host (IN, ..) // Ch3: transition point
10 }
```

Fig. 8: Simplied `loopback` gadget code.

C. Mutation Rules

Although a state transition guided by the state machine could lead to an actual state transition at runtime as intended, such a transition may fail in reality due to non-deterministic factors, such as interrupts, and states of uncontrollable global objects inside the kernel. We tackle this issue by allowing users to define additional options to respond to such failures using the mutation rules.

To facilitate a state-guided fuzzing, FUZZUSB provides a list of mutation rules to establish a detailed strategy for state-guided mutation, such as dictating state transition towards a targeted destination or accommodating a specific gadget testing requirement. Table III outlines the rules for stateful fuzzing, i.e., 1) *transition interval*, 2) *coverage guidance*, 3) *next state target*,

Rule	Description
R1: *Transition interval*	period of time between two transitions
R2: *Coverage guidance*	state-, or transition-coverage
R3: *Next state target*	what to choose for next state
R4: *Transition failure handling*	reconnect or try again

TABLE III: Mutation rules.

	Cur	Next	Transition
T_1:	S_2	S_3	`Ch4`: write-keycode
T_2:	S_3	S_4	`Ch3`: read-keycode
T_3:	S_2	S_5	`Ch4`: read-keycode
		...	

Mutation Rule
R1: 3 sec interval
R2: state-coverage
R3: unexplored first
R4: reconnect

TABLE IV: A simplified state machine for the HID gadget (left) along with the mutation rule (right). T_X and S_X denote unique transitions and states. **Cur** represents the current state, and **Next** means the next state to be transitioned. **Transition** specifies input values (along with a relevant input channel) needed to trigger the corresponding transition.

and 4) *transition failure handling*. **R1** specifies the transition interval between two connected states. Users can set a time value if they want to fuzz USB gadget stacks in each state for the given amount of time *before* moving forward to a next state. **R2** determines the method of coverage guidance, i.e., either state- or transition-based coverage, as we will explain later in §IV-E. **R3** prioritizes the next target state (or transition, depending on **R2**) out of multiple candidates (if there are multiple states connected to the current state), allowing random choice or unexplored states first. As a transition may fail for reasons, **R4** defines how to address such a transition failure, e.g., trying again for a given period of time or closing the entire connection immediately and starting over. Note that it is possible to adjust the granularity (fine- or coarse-grained) of the rules or extend them for other purposes, such as supporting certain algorithms for state exploration (e.g., DFS/BFS), as we will discuss in §VII. At runtime, the defined rules will be passed to the fuzzer determining state-aware input mutation.

D. Multi-channel Fuzzing

As mentioned in §III-A, gadget drivers accept multiple inputs from different channels at the same time; thus, if a fuzzer does not consider the roles of different channels (e.g., random input fuzzers), it is unlikely to achieve high efficiency. FUZZUSB orchestrates the entire execution of the fuzzing campaign based on the gadget state — not only steering input mutation but also distributing inputs into appropriate input channels. To this end, we rely on gadget state machines (§IV-B) along with mutation rules (§IV-C). In addition to mutation for state changes, FUZZUSB exercises a generic mutation guided by code coverage information within a state. To be effective, we leverage the results of solved constraints (from prior symbolic analysis) as input sources, which can help the fuzzer tackle strong branch conditions [9, 12, 32].

To elaborate on how our state-aware mutation operates, we step through the input generation procedure using the given state machine and mutation rule presented in Table IV. While state machines determine gadget-specific transitions in each state, mutation rules provide gadget-independent transition strategies. When reaching a new state — for instance, S_2 in the current state (**Cur**) —, the fuzzer decides when to trigger a transition (towards the next state) depending on the mutation rule **R1**. Accordingly, a transition will occur in three seconds, and the fuzzer will ask for the next state by referring to **R3** (the next target) and **R2** (coverage guidance). Suppose S_5 has been explored already while S_3 has not yet. Then, the fuzzer takes T_1 for the transition to trigger and chooses S_3 (in **Next**) as the next state. Subsequently, it retrieves the corresponding transition values for the input (i.e., `write-keycode` in **Transition**) from T_1 and feeds them into the designated channel `Ch4`. If the transition fails, the fuzzer reacts based on the transition failure option described in **R4** and takes action (i.e., reconnect) accordingly. While staying within a state, the fuzzer performs a mutation guided by code coverage, similar to a general coverage-guided fuzzer.

E. State Coverage vs. Transition Coverage

To maximize the benefit from the stateful approach, we define two coverage metrics with respect to gadget states, *state coverage* and *transition coverage*, which are similar to block coverage and edge coverage used for code coverage representation. While the state coverage aims at visiting all presented states in state machines regardless of their transitions, the transition coverage attempts to reach all unique state transitions. The state coverage generally works well for unidirectional state machines. Since our state machines are directional, two transitions with opposite directions but connecting the same two states are considered distinct. The transition coverage fits better for our case. Nevertheless, as either metric can work for its own purpose, FUZZUSB allows users to choose a preferred one in the mutation rule.

V. IMPLEMENTATION

We implemented FUZZUSB prototype atop the *syzkaller* kernel fuzzer [16]. We customized its components to make it suitable for gadget stack fuzzing while piggybacking on the underlying functionalities of syzkaller, such as code coverage guided mutation. Specifically, we extended the mutation engine to deploy our state-aware mutation along with state machines and support multi-channel input distribution. To enable and tune the init channel `Ch1`, we employ ConfigFS interface [27], allowing user-space code to configure various USB gadgets from the gadget side. To feed inputs from the host, we devise a dummy host driver on top of the `usbtest` kernel module and use the `dummy_udc` module for the software bridge between the host and device with the absence of a physical connection. Our state machine construction, i.e., static slicing and symbolic execution, relies on `dg` llvm slicer [33] with slicing criteria upon transition points and KLEE [34] (with Z3 solver [35]), which are all based on LLVM [36]. We customize KCOV [37] to collect code coverage from kernel threads regardless of the corresponding user-space applications, while vulnerability detection operates through KASan [38], UBSan [39], and

Vendor	Platform	Device	Android kernel
Google	ACK*	TBD†	5.4 (Android 12)
Samsung	SM-G981U	Galaxy S20	4.9
LG	LMV600AM	V60 ThinQ	4.19
Huawei	ELS-AN10	P40 Pro	4.9

∗ Android Common Kernels
† At the time of submission, this device's name has not been determined yet.

TABLE V: Specification of Android kernels used for the experiment.

Kmemleak [40] enabled within the kernel. Table X summarizes our efforts of modifying the tools used in FUZZUSB.

VI. EVALUATION

In this section, we evaluate FUZZUSB from different angles. First of all, we show that FUZZUSB can find previously-unknown bugs in USB gadget drivers, and we present a case study about FUZZUSB's findings (§VI-A). Then, we evaluate how the multi-channel and statefulness features of FUZZUSB contribute to its ability to cover USB gadget driver code (§VI-B). Lastly, we describe the fuzzing experimental setup, efficiency, and effectiveness for FreeBSD USB gadget fuzzing (§VI-C). Note that we also evaluate the efficiency in finding bugs (§A) and the state machine construction in Appendix (§B).

Experimental setup. We perform all our evaluations on a machine with an Intel Xeon E5-4655 2.50GHz CPU and 512 GB RAM running Ubuntu 16.04 LTS with Linux kernel 4.4.0. We run total 32 virtual machines with KVM on this platform to benefit from parallel fuzzing executions.

Mutation rules. We choose the best mutation ruleset that yields the highest coverage growth for our testing dataset. To obtain the best ruleset, we first performed a preliminary experiment. Specifically, we tested all the 16 different rule combinations by applying two options to each of the 4 mutation rules defined in Table III. Then, we used the best ruleset as the default in FUZZUSB for the rest of the evaluation. We describe the details in §VI-B.

Target kernels and gadgets. We base our evaluation on various gadgets from the latest Linux kernel versions (at the time of the experiment), ranging from v5.5 to v5.8. We also test Android gadgets (see §III) coming from different OEM vendors (as shown in Table V), each of which has its unique (or customized) vendor-specific gadgets. In total, our evaluation is based on 28 gadgets (§B). After porting the Android gadgets to the corresponding mainline kernels, we test all these gadgets on a single testing platform. To run fuzzing in a single Linux testing platform, we incorporated the Android USB gadgets into our fuzzing system. Because Android kernels depend completely on the Linux kernel, we are able to test all the Android gadgets in a QEMU-based virtualized environment. Furthermore, we extended FUZZUSB to cover USB gadget stacks in FreeBSD. Of 10 gadgets in the mainline FreeBSD kernel, we used 7 gadgets that implement the callbacks for UDC drivers in our virtualized fuzzing environment.

Three gadget fuzzers. Since no USB gadget fuzzer is available in the wild, we built a baseline fuzzer for comparison, `G-fuzzer`, which is also built on top of syzkaller. Moreover, to highlight the two main features of FUZZUSB, i.e., multi-channel and stateful fuzzing, we incrementally enable each feature in `FuzzUSB-SL` (SL: stateless) and FUZZUSB. Table VI summarizes the specifications of the three fuzzers.

`G-fuzzer` is featured with code-coverage guidance and is aware of the main interface to connect with gadget code, e.g., a USB host, but agnostic to advanced mutation strategies, such as multi-channel and stateful fuzzing. `G-fuzzer` represents a minimum engineering effort to turn syzkaller into a USB gadget fuzzer. Note that in terms of fuzzing scope and capability, `G-fuzzer` is at the same level of syzkaller [16] and USBFuzz [14], but it focuses on the USB gadget stack instead of the host stack. `FuzzUSB-SL` is capable of multi-channel input mutations, but it is still state-agnostic. Lastly, in FUZZUSB, all the features described in this paper are enabled.

Fuzzer	Interface -aware	Code coverage -guided	Multi-channel inputs	State -guided
G-fuzzer	✓	✓		
FUZZUSB-SL	✓	✓	✓	
FUZZUSB	✓	✓	✓	✓

TABLE VI: Specification of baseline gadget fuzzers.

A. Bug Discovery

1) New Bug Finding: Based on the above testing environment, we ran FUZZUSB extensively and discovered *34 previously unknown bugs* (listed in Table VII). While 9 of them were found within the Android gadgets, the remaining 21 bugs arose from the Linux gadgets. Although the majority are gadget bugs, we also discovered 4 USB host bugs as a by-product of the fuzzing. We have reported all the findings to the corresponding parties, among which 27 were confirmed by the community, 9 patched already, and 8 CVEs assigned. The bugs detected stem from various memory errors, such as use-after-free, null-pointer-deref, memory leakage, etc. These bugs could affect the kernel in a severe way and lead to exploitations, ranging from DoS attacks (§C-A) to control-flow attacks (§C-B). Among the Android gadget bugs detected, we notice that similar bugs could be detected from different OEM vendors. For example, we discovered an accessory gadget bug from the Google Android and then found a similar bug in the Samsung Android gadget. This could happen when OEMs derive their codebase from Google AOSP inheriting similar bugs from the upstream code without big changes.

2) Case Study: Human Interface Device (HID) devices, such as a keyboard and mouse, are used to interact with humans. The corresponding HID gadget facilitates HID-specific communications over USB. Figure 9 showcases simplified buggy code found in the HID gadget, where an error arises due to the race condition on the shared object `hidg`. Specifically, `f_hidg_read()` waits for data from the host (line 6). After it receives data from `Ch3`, a use-after-free crash may occur when accessing `hidg` (line 9) because `hidg` may have been

#	Bug type	Gadget	Crash module	Kernel	Status	CVE
1	Use-after-free	(host)	usb_hcd_unlink_urb (hcd.c)	linux-5.5	Patched	CVE-2020-12464
2	Slab-out-of-bounds	common	gadget_dev_desc_UDC_store (configfs.c)	linux-5.6	Patched	CVE-2020-13143
3	Integer overflow	(host)	k_ascii (keyboard.c)	linux-5.6	Patched	CVE-2020-13974
4	Memory leak	(host)	usbtest_probe (usbtest.c)	linux-5.6	Patched	CVE-2020-15393
5	Use-after-free	printer	printer_ioctl (f_printer.c)	linux-5.6	Patched	CVE-2020-27784
6	Use-after-free	hid	f_hidg_poll (f_hidg.c)	linux-5.6	Confirmed	
7	Use-after-free	mass	config_item_get (item.c)	linux-5.6	Confirmed	
8	Null-ptr-deref	acm	tty_wakeup (tty_io.c)	linux-5.6	Confirmed	
9	Memory leak	(host)	scsi_init_io (scsi_lib.c)	linux-5.5	Reported	
10	Use-after-free	printer	printer_read (f_printer.c)	linux-5.6	Patched	
11	Memory leak	loopback	usb_copy_descriptors (config.c)	linux-5.6	Confirmed	
12	Memory leak	hid	hidg_set_alt (f_hidg.c)	linux-5.6	Confirmed	
13	Use-after-free	serial	gs_flush_chars (u_serial.c)	linux-5.7	Confirmed	
14	Kernel panic	hid	usb_ep_queue (core.c)	linux-5.6	Confirmed	
15	Memory leak	uac1	u_audio_start_playback (u_audio.c)	linux-5.8	Confirmed	
16	Use-after-free	serial	tty_init_dev (tty_io.c)	linux-5.8	Confirmed	
17	Use-after-free	mass	do_set_interface (f_mass_storage.c)	linux-5.8	Confirmed	
18	Deadlock	tcm	tcm_alloc (f_tcm.c)	linux-5.8	Reported	
19	Use-after-free	hid	f_hidg_read (f_hidg.c)	linux-5.8	Confirmed	
20	Kernel panic	mass	usb_composite_setup_continue (composite.c)	linux-5.8	Confirmed	
21	Memory leak	serial	gs_start_io (u_serial.c)	linux-5.8	Confirmed	
22	Null-ptr-deref	ecm	gether_disconnect (u_ether.c)	linux-5.8	Confirmed	
23	Null-ptr-deref	common	__configfs_open_file (file.c)	linux-5.8	Reported	
24	Null-ptr-deref	ecm	ecm_opts_ifname_show (f_ecm.c)	linux-5.8	Reported	
25	Use-after-free	serial	check_tty_count (tty_io.c)	linux-5.8	Reported	
26	Use-after-free	conn	conn_gadget_read (f_conn_gadget.c)	android-4.9 (S)	Patched	
27	Use-after-free	mtp	mtp_read (f_mtp.c)	android-4.9 (S)	Confirmed	
28	Use-after-free	accessory	acc_read (f_acc.c)	android-5.4 (G)	Patched	CVE-2021-0936
29	Memory leak	cdev	usb_cser_set_alt (f_cdev.c)	android-4.19 (S)	Reported	
30	Use-after-free	laf	laf_read (f_laf.c)	android-4.19 (L)	Patched	CVE-2021-26689
31	Use-after-free	accessory	acc_read (f_acc.c)	android-4.9 (S)	Confirmed	
32	Use-after-free	mtp	mtp_read (f_mtp.c)	android-4.19 (L)	Confirmed	
33	Use-after-free	cdev	f_cdev_open (f_cdev.c)	android-4.9 (S)	Patched	CVE-2021-30313
34	Null-ptr-deref	laf	laf_release (f_laf.c)	android-4.19 (L)	Reported	

TABLE VII: List of previously unknown bugs discovered by FUZZUSB.

deallocated by `hidg_free()` without checking its validity (line 16).

Note that to trigger this bug, a fuzzer needs to not only consider inputs from multiple channels but also understand the statefulness of the communication. Figure 10 illustrates a sequence of state transitions leading up to the bug. Since the buggy point resides deeply in the state machine, to reach this location, the fuzzer needs to follow a specific transition path, producing the right inputs for the different channels. In particular, when the code is in the state (`wait-data`), without the input from `Ch1` and `Ch3`, a simple sequence of fuzzing inputs in a limited channel cannot trigger the bug because the code will not receive data from the host, but instead return at line 7. Unlike other code coverage guided fuzzers, FUZZUSB is able to follow all the needed steps to fully explore the code's internal states, revealing deep bugs much more efficiently.

```
1  /* function from Ch4 */
2  ssize_t f_hidg_read(struct file *file, ...)
3  {
4      struct f_hidg *hidg = file->private_data;
5      ...
6      if (recv_data_from_host ()) // wait for data from Ch3
7         goto fail;
8      ...
9      list = list_entry(&hidg->complete, ...) // error!!
10     ...
11 }
12 /* function from Ch1 */
13 void hidg_free(struct usb_function *f)
14 {
15     struct f_hidg *hidg = func_to_hidg(f);
16     kfree(hidg);
17     ...
18 }
```

Fig. 9: Vulnerable HID gadget code.

B. Efficiency

Apart from its ability to find previously unknown vulnerabilities, we evaluate FUZZUSB in terms of code coverage. As mentioned earlier in §VI, we first try to obtain the best mutation ruleset, which we will rely on for the rest of the evaluation. In this experiment, we run FUZZUSB using different mutation rules and compare the achieved coverage. Considering the rule choices available and various testing scenarios, we use a total of 16 different rule combinations as follows: *nonstop transition* (A) and *3 sec interval transition* (B) for **R1**, *transition-coverage* (C) and *transition-coverage* (D) for **R2**, *unexplored state selection* (E) and *random state selection* (F) for **R3**, and *reconnection* (G) and *5 times retrials* (H) for **R4**[3]. Figure 11 presents the per-rule average code coverage for 28 gadgets. We ran the tests for 24 hours for each combination with every gadget. The experimental results show that coverage does not benefit much

[3]We did initial experiments ahead by changing (and incrementing) thresholds. As a result, 5 times retry was enough for state transitions to complete when transitions failed, and 3 sec transition interval was the smallest we could observe some difference from nonstop transitions.

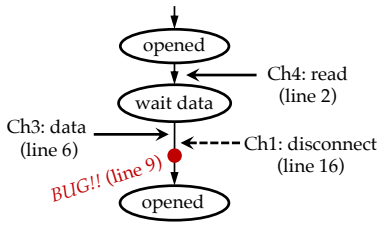


Fig. 10: The state transitions of the vulnerable HID gadget in Figure 9 necessary to reach the bug.

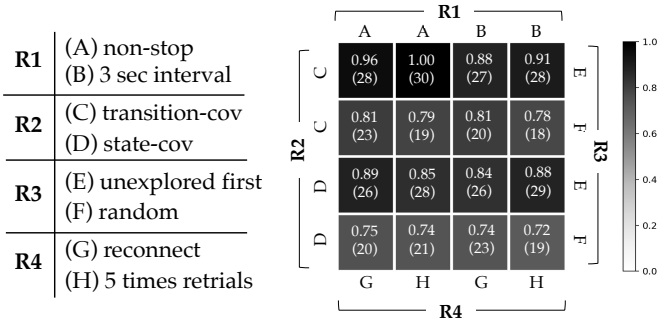
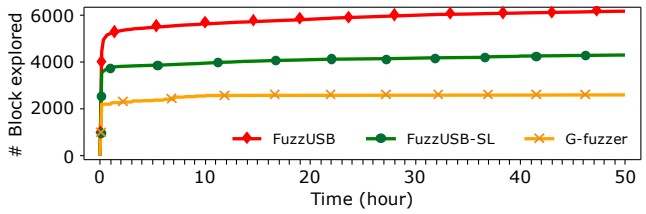
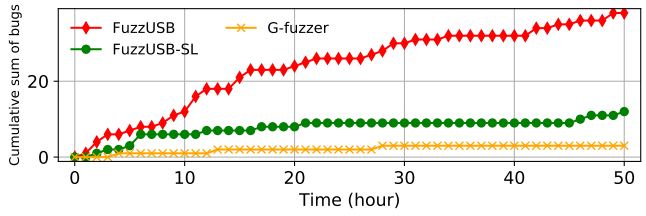


Fig. 11: The colormap for the coverage and the identified bug count (in parenthesis) under 16 mutation rules. While showing mutation rules **R1** - **R4** in each side, **A** - **H** indicate the concrete options used for each rule.



(a) Code coverage.



(b) Cumulative sum of bugs found.

Fig. 12: Coverage and bug-finding results running FUZZUSB for 50 hours.

from random transition **R3**(F) in comparison with unexplored first approach **R3**(E) due to the frequently revisiting of the explored states by random transitions. Meanwhile, transition coverage **R2**(C) presents a slightly better result than state-coverage **R2**(D), as taking bi-directional transitions provides more chances to explore more paths. Based on these results, we choose the mutation ruleset that yields the best performance in coverage, i.e., showing 1.0 scaled coverage in the table — *nonstop transition* (**R1**(A)), *transition-coverage* (**R2**(C)), *unexplored first* (**R3**(E)) and *5 times transition retrials* (**R4**(H)). We use this ruleset for the rest of our evaluation.

We ran FUZZUSB using 28 gadgets and measured the achieved code coverage over 50 hours. We compared it with the result from the baseline fuzzers, `G-fuzzer`, and `FuzzUSB-SL`. Figure 12a shows the average results after three runs. Compared with the baseline `G-fuzzer`, FUZZUSB achieved significant coverage improvements. FUZZUSB also outperforms `FuzzUSB-SL`, implicating that state-aware input generation is crucial in improving the code coverage in addition to multi-channel fuzzing. In short, full-fledged FUZZUSB showed three times and two times of improvement in the code coverage when compared to the general code-coverage guided fuzzers, `G-fuzzer`, and `FuzzUSB-SL`, respectively.

Figure 13 presents per-gadget code coverage after 50-hour runs. As expected, FUZZUSB showed better coverage for all the cases in comparison with `FuzzUSB-SL` and `G-fuzzer`. Note that the coverage difference is much higher in the `mass_-storage` gadget because of the reason that we discussed in §III — its core functionality (e.g., handling SCSI) relies on state-aware inputs, resulting in poor performance if inputs remain state-agnostic. Unlike FUZZUSB and `FuzzUSB-SL`, the coverage of `G-fuzzer` showed varying results per gadget. The reason is that most of the coverage achieved by `G-fuzzer` was derived by executing the first two initialization phases of a gadget driver (i.e., configuration and enumeration), which are shared among gadgets.

We designed another experiment to better highlight the contribution of FUZZUSB to code coverage of stateful fuzzing. In this experiment, we trigger state transitions every 30 seconds, while, in between, we resort to code coverage only to guide mutations. Figure 14 demonstrates the code coverage improvement rate for the first 5 minutes of FUZZUSB's run. The experiment relies on 3 gadgets that represent unique behaviors with different USB device classes (e.g., Ethernet and storage). As expected, the higher rate of coverage growth happens every 30 seconds, just after we enable the state-transition-based mutations. In particular, higher growth rates are observed up to the early stage of the data communication phase (near 150 seconds), meaning that state transitions are essential in these phases.

In addition, we investigate the effectiveness of the multi-channel fuzzing, breaking down the overall coverage by inputs from different channels. Figure 15 shows per-channel coverage results with the same gadget set used in Figure 14, listing coverage from the host, gadget, and both channels (i.e., `combination`). Although the figure solely illustrates the partial result in the first 3 hours of the experiment (to particularly highlight the most coverage change), we ensure that each experiment lasted for 24 hours and the rest of the (unplotted) results yielded the same trend. As shown in the figure, for all these gadgets, the individual channels have their unique coverage contribution, and this implies that fuzzing multiple channels is essential to achieve more coverage than single-channel fuzzing. Overall, the gadget channel presents a

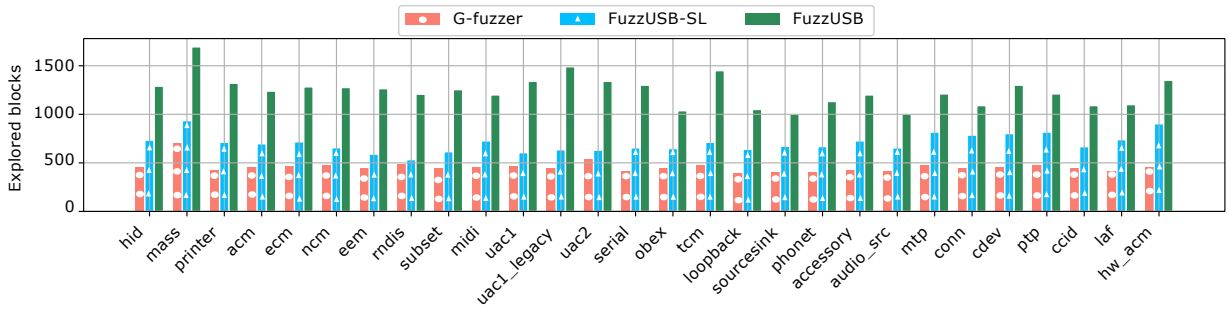


Fig. 13: Per-gadget code coverage after 50-hour runs.

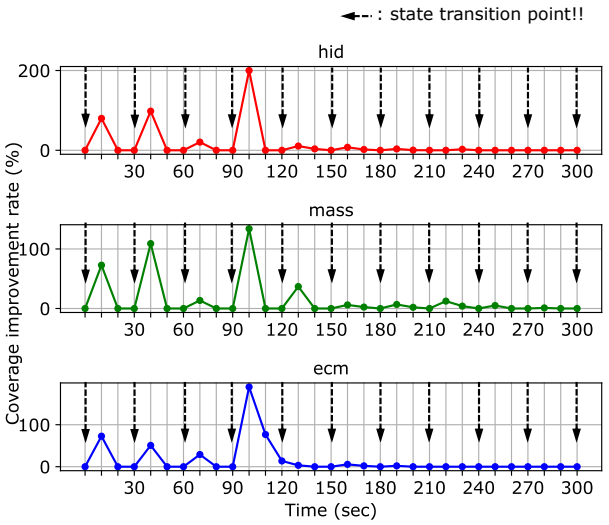


Fig. 14: Coverage improvement rate within the first five minutes of execution for three representative gadgets. To highlight coverage contribution by state changes, we only allow state transition every 30 seconds.

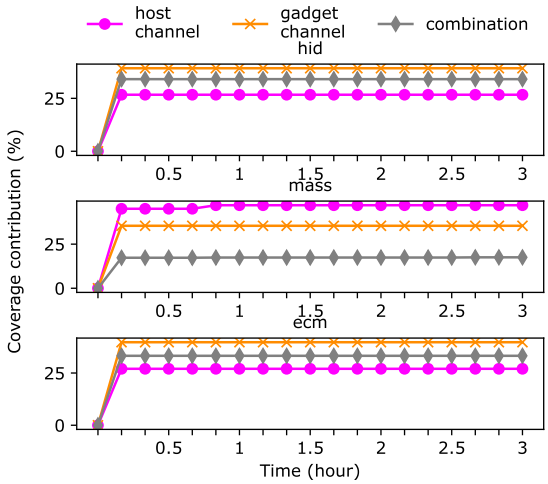


Fig. 15: Coverage breakdown by the type of channels. While presenting unique coverage contribution from `host` and `gadget` channels, `combination` represents the coverage from the both channels.

higher contribution at an earlier time, because it is in charge of early initialization steps, e.g., gadget configuration. Note that the host channel has more impact on `mass_storage` gadget than the others. This happens because the `mass_storage` gadget heavily interacts with the host by following the SCSI protocol.

C. FreeBSD USB Gadget Stack

To show the generality and flexibility of FUZZUSB, we extend FUZZUSB to examine the USB gadget stack of FreeBSD 14, the latest version at the time of the experiment. Note that FreeBSD is used for not only desktop and server machines but also embedded and IoT devices, supporting multiple USB gadget functionalities [44–46].

Experimental setup. Unlike Linux and Android, FreeBSD USB stacks lack a software-emulated bridge between the host and gadget stacks. Thus, we emulated the hardware bridge from scratch to handle underlying UDC hardware, and established a virtual connection from the USB host. Based on this, we applied our fuzzing system equipped with our stateful input mutation engine to FreeBSD gadgets. Using 7 gadgets, we carried out experiments under the same mutation ruleset used in §VI-B. Note that there is no explicit user-side input channel because FreeBSD gadget drivers are self-contained without user-specified configurations (e.g., device setup inputs). Hence, we focus on host channel fuzzing guided by the extracted gadget state machines. We achieve this by adjusting the algorithm to recognize the FreeBSD-specific transition and gadget entry functions, such as `usb_request_callback()`. For comparison, we designed a basic FreeBSD gadget fuzzer as the baseline using typical code coverage instead of stateful fuzzing on par with G-FUZZER.

Evaluation. Based on the experimental setup, we evaluate several aspects of FUZZUSB on FreeBSD. In particular, we examine the bug-finding efficiency and coverage. Similar to §A, we measured the detection time of a given bug (crash) compared with the baseline. Since there are no available bug reports for the FreeBSD gadgets in testing [47], we introduced a previously-known memory corruption security bug by reverting a commit fixing it (i.e., one safety check against invalid memory access [43]). As shown in Table VIII, FUZZUSB led the executions to the bug quickly while the baseline cannot discover

ID	Bug type	Gadget	Kernel	Detection time		
				G-fuzzer	FuzzUSB-SL	FuzzUSB
CVE-2019-14763 [41]	Deadlock	hid	linux-4.16	✗	2.51 mins	2.24 mins
CVE-2018-20961 [42]	Double-free	midi	linux-4.16	✗	4 days	3.5 hours
commit-9c847ff [43]	Invalid access	eem	FreeBSD	✗	N/A	1.18 mins

TABLE VIII: Detection time for previously-known USB gadget bugs (✗: detection failure).

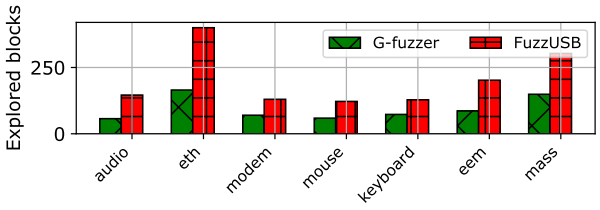


Fig. 16: Per-gadget coverage in FreeBSD after 50-hour run.

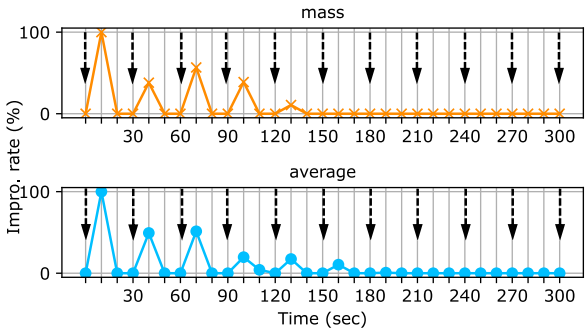


Fig. 17: Coverage improvement rate for 7 FreeBSD gadgets on average. Similar to Figure 14, a state transition is triggered every 30 seconds for 5 minutes.

the bug due to stateless fuzzing that unlikely reaches deep code paths. Furthermore, we carry out fuzzing campaigns for 50 hours and collect execution coverage. The result in Figure 16 shows that FuzzUSB outperformed the baseline coverage. Specifically, it achieves 2.2× improvement of coverage. The performance improvement is larger in large codebase gadgets (e.g., eth). Such results imply the advantage of the state-aware fuzzing of FuzzUSB. To highlight the improvement contributed by our stateful input mutation, similar to Figure 14, we demonstrate the coverage improvement rate during the first 5 minutes of the execution. Figure 17 depicts the mass gadget as an example and the average result of our targeted 7 FreeBSD USB gadgets. As expected, each state change directly contributed to a higher coverage increase, especially during the early time of the execution.

Takeaway. As demonstrated by the evaluation above, we can apply FuzzUSB to other operating system kernels for USB gadget stack fuzzing, provided that a software bridge is available to establish a virtual USB connection. The fact that no USB gadget bugs are ever reported, and our fuzzing did not find any USB gadget bugs either in FreeBSD may suggest some efficient ways to reduce the attack surface of USB gadget stacks, that are, 1) limiting the number of gadgets, 2) simplifying the implementation of each gadget, and 3) disabling user-space configurations.

VII. Discussion

Bug reproducibility. Among our new bug findings, we observe a number of them being race conditions bugs. As a result, we notice that around 30% of our findings could be reproduced deterministically, while the rest could not be reproduced due to the non-deterministic nature of race conditions. This is also a known issue of syzkaller when reproduced programs fail to reproduce bugs. One possible way to increase the bug reproducibility is to record and replay the runtime state of a gadget driver within the kernel, which requires a unified design and implementation of a gadget state.

Optimal mutation strategy. As mentioned in §VI, we used the fixed rules for the stateful mutation in FuzzUSB by default, assuming that they can help maximize the performance in coverage extension. However, the optimal mutation strategy might be gadget-dependent and need to be changed over time depending on the status of a gadget. For example, complex gadgets with a large codebase, such as `mass_storage`, should be extensively targeted and fuzzed in the data communication phase to be more effective. We leave such exploration for our future work.

Android USB gadget fuzzing. To fuzz Android-specific gadget drivers, we manually looked into Android kernels from different Android OEMs and ported the unique drivers into the corresponding mainline kernels. Ideally, fuzzing an Android kernel within QEMU directly without any change would be the best way. Unfortunately, due to hardware diversity, we could not find one emulation environment supporting all different Android kernels. For the same reason, existing Android USB fuzzing still relies on physical Android devices [48].

USB gadget stack coverage. Although we have been focusing on USB gadget drivers, such as CDC, HID, and MSC, FuzzUSB covers the whole USB gadget stack fuzzing because of the connections between USB gadget drivers and other layers as shown in Figure 1 except UDC drivers. To fuzz UDC drivers in a scalable fashion, we need to emulate the physical layer of UDC hardware in QEMU, which unfortunately only provides the basic HCD hardware (e.g., xHCI) evaluation. Consequently, the current UDC driver fuzzing requires the corresponding hardware, such as a RaspPi Zero or an Android phone [20].

USB gadget stacks in other OSes. The USB gadget stack in the Linux kernel might be the most widely used due to

Android and Embedded Linux adoption. Beside Linux gadgets, we additionally evaluate FUZZUSB with the USB gadget stack in the FreeBSD kernel to present the generality of our approach. Hence, we believe FUZZUSB is also applicable to other open-source gadget stacks, such as Zephyr [49] and Mbed [50], which usually have limited functionalities, e.g., supporting a limited number of USB classes, although maybe non-trivial, requiring an emulation environment for the corresponding OS, e.g., QEMU, a USB host environment provided by the target OS or another OS, e.g., Linux, a virtual USB connection between the USB host and the corresponding USB gadget stack, and a means to collect code coverage from within the target OS.

Imprecise static analysis. Static analysis is known to potentially cause imprecise analysis results [51]. Specifically, false positives can be introduced by over-approximation of static analysis encompassing infeasible indirect call targets. In contrast, false negatives can be introduced by under-approximation caused by missing indirect call targets. Fortunately, we did not observe any specific problems caused by imprecise static analysis in our experiments, as its analysis scope is bounded within explicit gadget entry and transition points. Meanwhile, we could mitigate the aforementioned limitations by leveraging existing techniques aiming at reducing the false positives [52, 53] or false negatives [54]. Moreover, symbolic analysis can compensate for such limitations. For example, as our path exploration analysis is achieved through symbolic execution, we can handle indirect branches as long as the corresponding function pointers are symbolically tainted. Although we may miss untainted indirect branches, they can be ignored safely because such branches are not part of state machines controlled by fuzzing inputs.

VIII. RELATED WORK

USB vulnerability detection. Despite the consistent efforts to reveal USB vulnerabilities for many years [14–19, 55–57], USB still has been shown to be vulnerable to various attacks. Relying on software testing technique, one side of the efforts [18, 19, 48] takes a hardware-based approach, leveraging dedicated hardware boards to support essential functions of USB devices. Another large body of works [14–17, 55, 57] examines USB stacks using software emulated USB devices without physical peripheral devices. Either way, as discussed in §III, existing USB fuzzers are not suited for fuzzing USB gadget stacks as they are originally designed to test USB host drivers. In particular, they do not much take the key features of USB protocols into account: multiple input channels and statefulness. Thus, their fuzzing efficiency is limited significantly. FirmUSB [56] exercises symbolic execution with USB domain knowledge and finds security bugs in USB device firmware. However, it cannot tackle the statefulness and multiple channel challenges either, and suffers from other issues, such as path explosion.

Stateful fuzzing. Modern fuzzers often increase the efficiency by taking the statefulness of programs under test [8–10, 58–63]. Commonly, OS kernel fuzzers [8–10, 58] try to resolve the dependencies of system calls (and APIs), to generate better test cases that fit into the input format specific to the target domain. Despite such efforts to understand stateful kernel execution, it cannot represent the accurate internal states of the target system, thereby making stateful fuzzing less efficient. Known network protocol fuzzers [60–62] or mobile application fuzzers [59] also rely on stateful communication with explicit state machines. However, they require either network traces or mobile communication logs in order to infer their state machines which cannot accurately represent actual state machines of the target system. More importantly, the aforementioned stateful fuzzers must be customized significantly to fuzz USB software stacks with multi-channel inputs. A recent work [63] devises stateful fuzzing by taking advantage of good human oracle rules to explore untested state space, but its mutation fully relies on manual annotations. Consistent with their claim, we leverage USB states as an oracle specialized in USB gadget fuzzing, which existing works have not accomplished. Furthermore, our oracles require in-depth analysis of complex USB software stacks to define input spaces and automatically build state machines.

IX. CONCLUSION

In this paper, we present FUZZUSB, the first USB fuzzing technique for a USB gadget system. FUZZUSB achieves state-guided fuzzing upon gadget-specific state machines, which effectively addresses the multi-channel and stateful nature of USB communication. FUZZUSB found 34 previously unknown vulnerabilities with security impacts in the latest Linux and Android kernels, and outperformed the baseline fuzzers with 3× higher code coverage, 50× improved bug-finding efficiency for Linux USB gadget stacks, 2× higher code coverage for FreeBSD USB gadget stacks, and reproducing known bugs that the baseline fuzzers could not detect.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for the valuable comments and suggestions. This work was supported in part by ONR under grants N00014-20-1-2128 and N00014-20-1-2205, and NSF under grant CNS-1815883. This material is also based on research sponsored by AFRL under contract number AFRL FA8650-19-1-1741. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] "Usb 3.2 specifications," USB Implementers Forum https://www.usb.org/document-library/usb-32-specification-released-september-22-2017-and-ecns, 2017.

[2] "Usb type-c® cable and connector specification," USB Implementers Forum https://www.usb.org/usb-type-cr-cable-and-connector-specification, 2019.

[3] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *IEEE Security & Privacy*, vol. 9, no. 3, pp. 49–51, 2011.

[4] D. Kierznowski, "Badusb 2.0: Usb man in the middle attacks," *Retrieved from RoyalHolloway*, 2016.
[5] D. J. Tian, G. Hernandez, J. I. Choi, V. Frost, C. Raules, P. Traynor, H. Vijayakumar, L. Harrison, A. Rahmati, M. Grace *et al.*, "Attention spanned: Comprehensive vulnerability analysis of {AT} commands within the android ecosystem," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 273–290.
[6] BlackHat, "Mactans: Injecting malware into ios devices via malicious chargers," https://media.blackhat.com/us-13/US-13-Lau-Mactans-Injecting-Malware-into-iOS-Devices-via-Malicious-Chargers-WP.pdf.
[7] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kafl: Hardware-assisted feedback fuzzing for os kernels," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.
[8] S. Pailoor, A. Aday, and S. Jana, "Moonshine: Optimizing os fuzzer seed selection with trace distillation," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, USA, Aug. 2018.
[9] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "Hfl: Hybrid fuzzing on the linux kernel," in *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2020.
[10] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.
[11] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.
[12] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.
[13] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.
[14] H. Peng and M. Payer, "Usbfuzz: A framework for fuzzing usb drivers by device emulation," in *25th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 397–414.
[15] J. Patrick-Evans, L. Cavallaro, and J. Kinder, "Potus: Probing off-the-shelf usb drivers with symbolic fault injection," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.
[16] D. Vyukov, "Syzkaller," https://github.com/google/syzkaller, 2015.
[17] S. Schumilo, R. Spenneberg, and H. Schwartke, "Don't trust your usb! how to find bugs in usb device drivers," *Blackhat Europe*, 2014.
[18] NCCGroup, "Umap2," https://github.com/nccgroup/umap2.
[19] T. Goodspeed and S. Bratus, "Facedancer usb: Exploiting the magic school bus," in *Proceedings of the REcon 2012 Conference*, 2012.
[20] Syzkaller, "Linux kernel usb bugs found by syzkaller," https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs_usb.md.
[21] Https://github.com/purseclab/fuzzusb.
[22] "Usb on the go and embedded host," USB Implementers Forum https://www.usb.org/usb-on-the-go.
[23] J. Hyde, "Usb multi-role device design by example," *Comissioned by Cypress Semiconductors*, 2003.
[24] "Class definitions for communication devices (1.2)," USB Implementers Forum https://www.usb.org/document-library/class-definitions-communication-devices-12.
[25] "Usb human interface device (hid) information," USB Implementers Forum https://www.usb.org/hid.
[26] "Mass storage class specification overview (1.4)," USB Implementers Forum https://www.usb.org/document-library/mass-storage-class-specification-overview-14.
[27] J. Becker, "Configfs," https://www.kernel.org/doc/Documentation/filesystems/configfs/configfs.txt.
[28] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.
[29] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data race fuzzing for kernel file systems," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.
[30] "Usb gadget api for linux," The Linux Kernel https://www.kernel.org/doc/html/v4.17/driver-api/usb/gadget.html.
[31] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.
[32] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "Qsym: a practical concolic execution engine tailored for hybrid fuzzing," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, USA, Aug. 2018.
[33] M. Chalupa, "dg," https://github.com/mchalupa/dg, 2016.
[34] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.
[35] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.
[36] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.
[37] "Kcov," https://www.kernel.org/doc/html/v4.15/dev-tools/kcov.html, 2018.

[38] "Kernel address sanitizer," https://github.com/google/kasan/wiki, 2018.
[39] "Undefined behavior sanitizer," https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2018.
[40] "Kernel memory leak detector," https://www.kernel.org/doc/html/v4.10/dev-tools/kmemleak.html, 2018.
[41] MITRE, "CVE-2019-14763." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-14763, 2019.
[42] ——, "CVE-2018-20961." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-20961, 2018.
[43] FreeBSD, https://cgit.freebsd.org/src/commit/?id=9c847ffd743b4f68af56c5069da401bd1831efcb, 2020.
[44] T. F. Project, "Freebsd," https://www.freebsd.org/.
[45] TechRadar, "Security holes put 100 million iot devices at risk," https://www.techradar.com/news/security-holes-put-100-million-iot-devices-at-risk.
[46] S. I. E. Inc., "Open source software used in playstation 3," https://doc.dl.playstation.net/doc/ps3-oss/index.html.
[47] F. bug report, https://bugs.freebsd.org/bugzilla/, 2020.
[48] Syzkaller, "Syzkaller for android device," https://android.googlesource.com/platform/external/syzkaller/+/HEAD/docs/linux/setup_linux-host_android-device_arm64-kernel.md.
[49] L. F. Projects, "Zephyr rtos," https://www.zephyrproject.org/.
[50] ARM, "Mbed os," https://os.mbed.com/mbed-os/.
[51] T. Kim, V. Kumar, J. Rhee, J. Chen, K. Kim, C. H. Kim, D. Xu, and D. J. Tian, "Pasan: Detecting peripheral access concurrency bugs within bare-metal embedded applications," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 249–266.
[52] K. Lu and H. Hu, "Where does it go? refining indirect-call targets with multi-layer type analysis," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1867–1881.
[53] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang, "Pex: A permission check analysis framework for linux kernel," in *Proceedings of 28th USENIX Security Symposium (USENIX Security)*, 2019.
[54] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "Dr.checker: A soundy analysis for linux kernel drivers," in *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, 2017.
[55] D. Song, F. Hetzelt, J. Kim, B. B. Kang, J.-P. Seifert, and M. Franz, "Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 2541–2557.
[56] G. Hernandez, F. Fowze, D. Tian, T. Yavuz, and K. R. Butler, "Firmusb: Vetting usb device firmware using domain informed symbolic execution," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2245–2262.
[57] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz, "Periscope: An effective probing and fuzzing framework for the hardware-os boundary," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.
[58] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim, "Finding semantic bugs in file systems with an extensible fuzzing framework," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.
[59] E. B. Yi, H. Zhang, K. Xu, A. Maji, and S. Bagchi, "Vulcan: Lessons in reliability of wear os ecosystem through state-aware fuzzing," in *Proceedings of the 18th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2020.
[60] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, "Pulsar: Stateful black-box fuzzing of proprietary network protocols," in *Proceedings of the International Conference on Security and Privacy in Communication Systems (SecureComm)*. Springer, 2015, pp. 330–347.
[61] R. Ma, T. Zhu, C. Hu, C. Shan, and X. Zhao, "Sulleyex: A fuzzer for stateful network protocol," in *Proceedings of the International Conference on Network and System Security (NSS)*. Springer, 2017, pp. 359–372.
[62] J. De Ruiter and E. Poll, "Protocol state fuzzing of tls implementations," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 193–206.
[63] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring deep state spaces via fuzzing," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.
[64] Y. Chen and X. Xing, "Slake: facilitating slab manipulation for exploiting vulnerabilities in the linux kernel," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1707–1722.
[65] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "{FUZE}: Towards facilitating exploit generation for kernel use-after-free vulnerabilities," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 781–797.

APPENDIX A
EFFICIENCY OF BUG FINDING

In this section, we evaluate how effective FUZZUSB is in finding bugs. We performed the following two experiments: 1) measure the number of vulnerabilities identified by FUZZUSB within a limited period of time; and 2) measure the time taken to identify previously known CVE vulnerabilities. While running each experiment, we also run other gadget fuzzers to compare the time-to-detection.

First, we examine how many bugs that FUZZUSB can discover within a limited period of time. Using the kernel version Linux-5.5, we ran FUZZUSB along with the two gadget fuzzers for 50 hours. Removing duplicate crashes (using syzkaller crash-hashing functionality), we recorded the number of uniquely identified bugs for three running times, then obtained the average results as presented in Figure 12b.

Compared with FuzzUSB-SL and G-fuzzer, FUZZUSB not only found more bugs (about 40) in total but also found the first bugs faster. Given the total number of bugs detected by G-fuzzer and FuzzUSB-SL in 50 hours, FUZZUSB showed the same bug-finding capability after 1 hour and 10 hours, respectively.

We carried out an additional experiment based on known CVEs to measure the time taken to detect. Specifically, we run FUZZUSB to detect CVE-2019-14763 and CVE-2018-20961. Table VIII summarizes the results of this experiment. FUZZUSB found these two bugs faster than FuzzUSB-SL. In addition, we noticed that G-fuzzer is unable to find any of these two bugs, since it cannot go deeper in the gadget code due to the lack of multi-channel fuzzing capability.

To summarize, in terms of bug-finding efficiency, FUZZUSB shows 50 times better performance than code-coverage fuzzing G-fuzzer, and achieves 5 times higher efficiency than FuzzUSB-SL.

Kernel	Gadget	# States/Trans	Analysis time (sec)		
			Static	Symbolic	Total
Linux & Android	hid	15 / 28	1.75	7.30	10.07
	mass	15 / 42	11.70	35.22	48.22
	printer	15 / 28	2.73	7.28	11.34
	acm	14 / 26	1.68	10.11	13.66
	ecm	14 / 26	1.69	13.28	16.67
	ncm	14 / 26	1.71	13.34	16.28
	eem	14 / 26	1.62	12.50	15.49
	rndis	14 / 26	1.43	14.29	17.53
	subset	14 / 26	1.68	12.21	15.20
	midi	14 / 26	1.24	15.19	17.40
	uac1	14 / 26	1.77	16.36	19.24
	uac1_leg	13 / 24	1.76	14.21	17.19
	uac2	13 / 24	1.86	15.92	19.41
	serial	14 / 26	1.16	9.17	11.72
	obex	14 / 26	1.30	7.72	9.89
	tcm	15 / 30	4.33	18.50	23.87
	loopback	12 / 24	0.79	6.90	9.38
	sourcesink	14 / 26	1.33	9.11	11.79
	phonet	14 / 26	1.26	10.27	13.30
Android specific	accessory	14 / 26	1.71	11.44	15.25
	audio_src	12 / 22	1.15	8.98	11.69
	mtp	14 / 26	2.11	11.12	14.53
	conn	14 / 26	2.22	12.24	15.45
	cdev	14 / 26	1.09	8.51	11.48
	ptp	14 / 26	2.09	11.97	16.16
	ccid	16 / 30	2.33	14.80	18.74
	laf	14 / 26	1.50	9.76	13.15
	hw_acm	14 / 26	1.74	9.19	12.17

TABLE IX: Performance of state machine construction.

APPENDIX B
OVERHEAD OF BUILDING STATE MACHINES

In this section, we examine the building process of state machines. Table IX summarizes the result of our state machine construction. In the table, the third column indicates the number of states as well as transitions for each gadget, and the last three columns present the time needed by the analyses to build the state machines. The overhead of building state machines is 15.93s on average, including static and symbolic analysis as well as state merging.

Our observation found that the mass_storage gadget has much more transitions than other gadgets because it needs to fully deal with a layered communication protocol (e.g., SCSI) over the USB channel. To tackle all protocol commands, we leverage corresponding transition inputs obtained from the symbolic execution, which increases the total transition numbers. Accordingly, to fully handle every command, mass_storage includes large functionalities with a large codebase. For this reason, building the state machine for this gadget driver requires more time. In contrast, the loopback gadget provides simpler functionalities — it receives messages from Ch3, and sends them back through that channel, resulting in a smaller number of states.

```
1  /* function from Ch2 */
2  int hidg_set_alt(struct usb_function *f, ...)
3  {
4      if (hidg->in_ep != NULL) {
5          ...
6          // allocation point
7          req_in = hidg_alloc_ep_req(hidg->in_ep, ...);
8          if (!req_in) {
9              // error handling
10         }
11     }
12     ...
13     if (hidg->in_ep != NULL) {
14         hidg->req = req_in; // missing point
15         ...
16     }
17 }
18 /* function from Ch1 */
19 void hidg_disable() {
20     ...
21     usb_ep_disable(hidg->in_ep);
22     ...
23 }
```

Fig. 18: A victim gadget code.

```
1  /* exploit function from Ch2 */
2  void Ch2_exploit(arg)
3  {
4      while (1)
5        feed_input(Ch2, arg)
6  }
7  /* exploit function from Ch1 */
8  void Ch1_exploit(arg)
9  {
10     while (1)
11       feed_input(Ch1, arg)
12 }
13 create_thread (Ch2_exploit, reset_connect);
14 create_thread (Ch1_exploit, disable_connect);
```

Fig. 19: Feasible PoC exploit against Figure 18.

APPENDIX C
ADDITIONAL CASE STUDIES

A. DoS attack

Aside from §C-B, we present an additional example of exploitation. In Figure 18, the buggy function hidg_set_alt tries to reset the connection. For a new connection, the function allocates a kernel object req_in to hold data from the host (line 7). Since the assignment of req_in into a global instance at line 14 is away from its initial allocation at line 7, during such time window, the object can leak by the not-taken branch at line 13 if unexpected disconnect function hidg_disable nullifies the instance in_ep (line 21). Figure 19 outlines the corresponding PoC code. The idea behind the PoC is to realize

Component	Language	Lines of Code
FSM construction	Python,C++	1,564
Fuzzer	Go	630
State manager	Python	844
Host backend	C	1,381

TABLE X: Lines of code composing FUZZUSB.

such a race condition between the reconnection (`Ch2`) and disconnection (`Ch1`), in the hope that we meet the condition of the memory leak above. Using two separate threads, we extensively feed inputs from the two channels `Ch1` and `Ch2`, to trigger reconnection and disconnection simultaneously. After 10-hour running of the exploit code, we observed the kernel memory space fills up (as a result of frequent memory leaks), causing the denial-of-service (DoS). Such a DoS exploit is crucial, for instance, when providing services in the cloud system. Note that FUZZUSB not only improves the capability of revealing such a bug with our multi-channel mutation but also facilitates exploit generation by providing the infrastructure of the multi-channel input distribution, which are unlikely done by existing USB fuzzers.

B. Control-flow attack

To demonstrate the high severity of our findings, we demonstrate an *arbitrary code execution* on the buggy HID gadget shown in Figure 9. Referring to the reported attack scenario against *use-after-free* bugs [64, 65], we successfully diverted the control flow of the HID driver, and achieved illegitimate code execution. The exploit works by reallocating the freed memory (i.e., `hidg` at line 16) and putting a compromised value into the function pointer in `hidg`, leading to an illegal control flow transfer when the HID driver accesses its dangling pointer. This example is fully exploitable because the vulnerable object `hidg` contains abundant function and data pointers (e.g., `unbind`), and attackers can populate them with user-supplied data via system calls (e.g., `sendmsg`).

A successful exploitation is challenging, because it needs to reallocate the memory region where the vulnerable object (e.g., `hidg`) was occupied in a short time window between its deallocation and the dangling pointer access. In this example, however, the attacker could easily manipulate such a time window, because the dangling pointer occurrence and dereference could be controlled by system calls, i.e., `close()` and `read()`, respectively. The PoC allows attackers to take control of the gadget, diverting its control flow.