SATURN: Host-Gadget Synergistic USB Driver Fuzzing

Yiru Xu, Hao Sun, Jianzhong Liu, Yuheng Shen, and Yu Jiang✉
BNRist, School of Software, Tsinghua University, Beijing, China

***Abstract*—The Universal Serial Bus (USB) is an essential component in modern operating systems, allowing for a wide assortment of peripherals to connect conveniently to a computer. The USB stack in an operating system usually consists of the following two components: the host-side driver and the device-side gadget driver, both of which are security-critical. If any vulnerabilities in these privileged-mode drivers are exploited, a malicious or malformed device could crash the whole system. Fuzzing, a popular automated vulnerability detection technology, has been applied to testing kernel components such as drivers with varying degrees of success. However, existing works mainly focus on one side and test drivers through emulating malicious input from userspace or peripherals while neglecting intricate internal states triggered only through interaction between the two boundaries, leaving a multitude of bugs exposed.**

In this paper, we propose SATURN, a host-gadget synergistic USB driver fuzzing approach, aiming to cover the entire handling chain throughout the USB communication. To achieve this, SATURN first leverages extracted driver information to attach gadgets systematically and trigger more driver types, facilitating the transition to interactive logic. Then, SATURN performs a persistent synergistic fuzzing process through canonical operation injection on both sides to play their own important roles, significantly expanding the states explored and exposing bugs in such logic. Compared to the state-of-the-art USB fuzzers, such as Syzkaller, USBFuzz and FUZZUSB, SATURN improves the branch coverage statistics on the corresponding stack by 1.53×, 3.69× and 2.3×, respectively. In addition, SATURN found 26 previously unknown bugs, among which are 4 CVEs, including drivers on each side.

1. Introduction

The Universal Serial Bus (USB) is one of the universal standards in the modern computer system. A wide variety of peripheral devices can connect to a computer through simply plugging into the USB socket. USB devices are pervasive and play an essential role in a variety of applications such as file transfer, charging, audio, and keyboard. Their functionalities are powered by two software components: the host-side driver and the device-side driver. Apart from working with the hardware, the host-side driver performs interactions with the user program as well as makes various requests to the peripherals. The device-side driver, commonly named the *gadget*, runs on billions of embedded systems and Android devices, which receives requests from the host and responds accordingly. While unifying peripheral connections for convenience, the USB system, like any other piece of hardware and software, contains bugs and is potentially dangerous. For instance, attackers can exploit vulnerabilities within the drivers to intercept sensitive traffic, steal private data, or even take control of a computer [10], [37]. As a master-slave model, the security of USB system depends on either side. The numerous functionalities expose device drivers to a broad attack surface, and in conjunction with the drivers running under the privileged mode, lead to the possibility that simply plugging in a malicious or malformed device might cause a system-level crash.

Fuzzing is an effective vulnerability discovery approach and has been applied to assist kernel testing. There has been prior research concerning USB driver fuzzing. Syzkaller's USB fuzzing module [13] injects randomly generated data from the device in response to the host's requests, exploring any possible vulnerabilities in the host drivers. USBFuzz [22] uses an emulated USB device virtually attached and detached to the target system, feeding random inputs when drivers perform IO operations. FUZZUSB [12] extracts internal state machines from USB gadget drivers to achieve state-guided gadget fuzzing through multi-channel inputs. In practice, these methods have achieved respectable results and have detected vulnerabilities within the USB stack.

However, these approaches mainly test one side of the USB stack, which limits its effectiveness significantly. As aforementioned, the USB system consists of host drivers and gadget drivers, both of which maintain their own internal state that can be altered by inputs from outside and interactions with the opposite side. These drivers expose two external boundaries: userspace application to the host kernel and external events to the device-side logic, (e.g., keyboard press). Existing works, which test from one side, experience difficulty in exploring states that can only be reached through both sides' interaction. For instance, a printer's *"mid-print, no paper"* status requires a joint effort. Specifically, to reach this state, the userspace program first submits a print job, which is dispatched to the host driver. Then the host driver informs the printer gadget and their respective states change to the printing in progress state. During printing, the printer detects a lack of paper in the intray, the information is then sent to the gadget, which subsequently interacts with the host to enter the corresponding state. In the above process, inputs from both the user application and the peripheral device influenced the states of

✉. Yu Jiang is the corresponding author.

both drivers, leading to a state that cannot be easily reached by sending inputs only to one side, thus will be incapable of detecting bugs that can only be triggered here.

This prompts a need for more sophisticated testing techniques that can effectively find bugs in the drivers' interaction phase, as currently, the Linux kernel primarily conducts initialization checks that detect invalid USB configurations, while the data transfer, i.e. device interactions, is left unchecked [14]. Therefore, we propose to fuzz the host drivers and the gadget drivers synergistically, targeting vulnerabilities within the drivers' interaction after attachment. To achieve such goals, the challenges are two-fold:

First, we need to significantly increase the success rate of driver initialization, thus the state of such drivers can transition to those involved in their respective devices' interactions, allowing fuzzers to uncover bugs within this process. Existing methods mainly rely on a limited set of initialization information or randomness to overcome kernel sanity checks, which are not robust enough to initialize a wide variety of drivers. SATURN employs Linux's built-in gadget subsystem as the device side, which can trigger various host drivers through synthesizing different attribute configurations. However, the gadget module has a vast configuration space, and randomly generated configurations can be easily rejected by the kernel's sanity checker. Therefore, it is imperative to devise a method for systematically and efficiently attaching a wide variety of gadget devices.

Second, we need to engage in the interaction process with canonical operations, significantly expanding the states explored and exposing bugs in such interactive logic. Previous approaches such as Syzkaller [5] perform less than optimal since they solely rely on predefined system call descriptions and test one side of the USB stack, thus a large proportion of the inputs it produces are incompatible with the dynamically attached driver. Therefore, it is essential to reliably detect dynamically attached device files including both the host drivers and the corresponding gadget drivers, and inject compatible and high-quality inputs to both sides, thus invoking their interactions and exploring the states that can only be reached through such interactions.

To address the aforementioned limitations and improve fuzzing effectiveness, we propose SATURN, a novel host-gadget synergistic USB driver fuzzing approach. Unlike typical fuzzers focusing on one side, SATURN enables both security-critical sides, the host and gadget, to play their own essential roles in the overall fuzzing process. First, SATURN directly extracts the possible configuration space from the kernel and utilizes the gadget subsystem to respond to initialization requests, thus greatly increasing the success rate of driver initialization. Next, SATURN efficiently detects the device files on both sides and dynamically probes the underlying file operations, allowing SATURN to be aware of the inputs that host and gadget drivers require. Finally, SATURN performs persistent fuzzing synergistically by injecting sequences on both sides, grouping syscalls according to gadget types, and fixing parameters if necessary, thus significantly increasing the quality of the generated inputs.

We implemented SATURN and evaluated its performance on recent versions of the Linux kernel. Overall, SATURN achieves higher coverage on the corresponding stack than the host-side fuzzer Syzkaller USB fuzzing module and USBFuzz, as well as the gadget-side fuzzer FUZZUSB by 1.53×, 3.69× and 2.3× on average. Subsequently, we evaluated the specific improvements gained by either component. Regarding SATURN's driver attachment component, it successfully triggered 304 unique drivers, compared to Syzkaller's 14 drivers and USBFuzz's 44 drivers. Regarding synergistic fuzzing, we extracted and compared the coverage of common drivers among Syzkaller, USBFuzz, and SATURN, and observed that synergistic fuzzing delivers a 58% and 77% improvement, respectively. This demonstrates that SATURN is capable of initializing more drivers and effectively exploring the interaction states simultaneously through the host-gadget synergistic fuzzing approach. In terms of vulnerability discovery capabilities, SATURN found 26 previously unknown vulnerabilities, among which are 4 CVEs, including the implementations on both the host and gadget drivers. The bugs have been reported to kernel maintainers and relevant patches have been merged.

Overall, this paper makes the following contributions:

- We identify the difficulties in USB driver fuzzing, and propose a host-gadget synergistic fuzzing approach targeting to cover the full handling chain during USB communication.
- We design SATURN, consisting of a systematic gadget attachment phase and an efficient host-gadget synergistic fuzzing phase, which is capable of decreasing the difficulty involved for fuzzers to test USB drivers and increasing the attack surface.
- We implemented SATURN and evaluated its effectiveness from the perspectives of the overall and component-wise improvements and found 26 unknown vulnerabilities, among which are 4 CVEs.

2. Background and Motivation

2.1. USB Host-Gadget Model

Since its introduction in the 1990s, USB has received widespread popularity due to the hot-plugging and plug-and-play features. It has become a ubiquitous standard used to connect a diverse range of peripheral devices such as keyboards, mice, printers, digital cameras, flash memory, and hard drives to the host computer. With the advent of high-speed USB standards (1.x, 2.x, 3.x, 4.x), numerous new peripherals have been developed, and a considerable number of drivers have been integrated into the kernel codebase.

USB follows an asymmetrical master-slave model in which the host computer functions as the master, while the peripheral devices serve as the slaves. In the communication process, only the host initiates communication requests by polling and sending data to the peripheral devices, which can only wait for the host's requests. This master-slave model implies that drivers on the host and device sides differ significantly. Linux, one of the most successful operating systems,

contains both side components. The host-side driver runs on the Linux computer, and the device-side driver running on the embedded Linux peripherals is the *gadget* driver.

Some Linux-based intelligent devices can serve as both host and peripheral device at the same time, due to On-The-Go (OTG) [36] and Dual-Role-Device (DRD) [8] features. For example, an Android phone acts as a host when it charges through a USB port and as a device when it connects to a computer for file transfer, and the internal driver of the latter is implemented by the gadget stack. The gadget subsystem itself achieves various device-specific features including, but not limited to network (CDC, ACM), human-computer interaction, mass storage, and audio interfaces. Besides, the gadget subsystem provides flexible configuration options, including various USB device properties, like many function types and interface attributes, thereby supporting attaching and triggering various host and gadget drivers through synthesizing different attribute configurations.

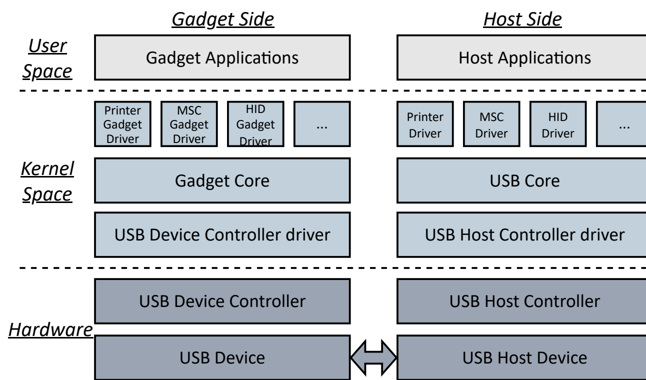

Figure 1: Host Gadget Architecture. Once the gadget device is configured and enumerated, host-side and gadget-side drivers communicate via requests originating from userspace applications, encapsulated by device drivers, and passed through the core driver, controller driver, and hardware to the recipient counterpart.

In Linux's implementation, the design of the host and gadget drivers share the same structure, as depicted in Figure 1, along with similar data structures. Directly interfacing with the hardware devices and controllers, the device/host controller drivers represent the layers communicating with the hardware through means such as registers and DMA. They offer foundational support for the subsequent device drivers. At the core layer, the gadget core driver handles driver lifecycle events by returning configuration and string descriptions, including managing bindings to hardware, disconnecting chains, etc. On the host side, the USB core driver is responsible for selecting the corresponding driver for the device, and processing information between the device and the host. The device drivers layer provides specific functionalities such as printers, Mass Storage Class (MSC), Human Interface Devices (HID), and generates/consumes protocol-specific data transfers. Finally, the top layer is comprised of applications, which allow users to interact with the kernel space for specific tasks.

When a USB gadget is connecting to a host, the system's behavior can be divided into several phases: (0) Gadget configuration stage. Users choose and configure a specific gadget through particular functions and attributes, which is the preparatory phase for connecting the gadget, unlike general USB devices that are already pre-programmed and configured. (1) Device enumeration stage. The gadget is bound to a particular USB Device Controller (i.e., a device is plugged into a port), and the host sends request messages to identify the device, following specific USB standards. This phase is handled entirely by the gadget and is transparent to the userspace. Once a gadget is fully identified, the host proceeds with driver matching and probing to initialize a specific driver. (2) Host-Gadget communication stage. Once a connection is established, both sides can send commands to each other, utilizing queues of request objects for packaging I/O buffers. Requests in any direction can impact the respective states of the host-side and gadget-side drivers.

2.2. Threat Model

USB drivers, as a medium, naturally face two types of threats: first, the threat from the OS-peripheral, for example, attackers can exploit vulnerabilities in the drivers of cloud service to take control of the devices and inject malicious inputs to attack host machines. Additionally, some malicious or malformed peripherals may send malicious data to the driver of the host, causing memory bugs like Use After Free [32], and other logical bugs, ultimately leading to system crashes. Second, the threat from the userspace-OS arises from system calls where malicious applications can manipulate files (e.g., `ioctl()`), or exploit vulnerabilities in the device drivers, leading to system crashes.

In this paper, we define the security boundary to be at two locations to assist us to identify and protect against the above threats: the userspace application to the host kernel and the device-side logic/event (i.e., keyboard press, mouse click) to the gadget subsystem. This is depicted by the upper dashed line in Figure 1. Therefore the external threat is events from the host userspace or device-side logic, as depicted in "User Space" with grey color in the diagram. Moreover, we assume that all hardware on both sides, including the device controller, is trustworthy, as depicted in "Hardware" with dark blue color. We also trust the code logic of the drivers in the kernel space, including the controller driver, core driver, and device driver, as depicted in "Kernel Space" with light blue color. All threats occur during the communication phase, where both the host-side and device-side drivers have to deal with vulnerabilities from external attack surfaces, including but not limited to applications, physical operations on hardware devices, etc.

There are many fuzzing efforts dedicated to discovering these potential threats. Some researchers explore the attack surface from OS-peripheral, who emulate the possible operation of peripherals by providing inputs to hardware I/O channels. For instance, FUZZUSB extracts internal state machines from USB gadget drivers and achieves state-guided fuzzing through multi-channel inputs from the host to the

gadget stack. On the other hand, some techniques address uncertainty from the userspace. Syzkaller injects random syscall sequences into the device files on the host side, relying on handwritten file paths and operation specifications by kernel experts. These efforts have yielded excellent results and found many vulnerabilities in the kernel drivers.

2.3. Motivating Example

In the communication of USB drivers, the host and gadget sides maintain their own state machines, which are affected by inputs from both userspace applications and device-side logic. When a command is launched from the userspace, it is processed to the host-side drivers first and prompts the transition of the host's status, then delivered to the gadget via the hardware. The gadget drivers transfer to the transaction-specific handler and respond to the host. Conversely, events from the device side affect the state of the gadget first, then the gadget drivers encapsulate messages to notify the host, leading to state changes and subsequent actions on the host side. Regardless of the input direction, they access certain pieces of the kernel logic and are routed through drivers on both sides. Additionally, the execution of requests involves two-way interaction, and the state on one side impacts the other side. However, existing works are limited to one-side data injection, thus possibly missing some parts of the communication chain, resulting in repeated exploration of already covered code and failing to expose vulnerabilities in more complex states.

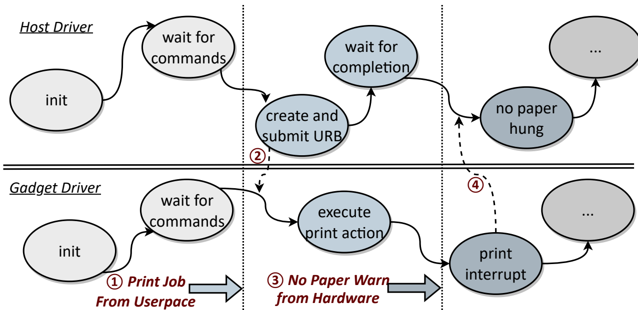

Figure 2: State machines for drivers on both sides in a printer's workflow. After successful attachment and initialization, the printer receives a job from userspace (①). The host-side driver creates and submits URBs to inform the device (②), while the gadget driver executes the print job. In this process, the state of the host impacts the gadget. In case the paper runs out, the printing on the gadget side is forced to interrupt (③), leading to the host driver entering the no paper hung status (④) and activating the subsequent processing mechanism. In this process, the state of the gadget affects the host. Moreover, the states of both sides are affected by the userspace and device side simultaneously.

As an example, consider the internal state transitions of the USB printer drivers (`usblp/gprinter`) depicted in Figure 2. Once a printer peripheral is plugged into the computer and initialized, both side drivers enter into a waiting state. Next, a print job is launched from the userspace through system call `usblp_write()`, the host-side driver starts to read the data with `copy_from_user()`, then creates and submits the URB to the device side and waits for its completion. When receiving the print request, the gadget-side driver constantly executes print action through interaction with the hardware. During the waiting process, the host driver continuously polls the status of the corresponding `IN` port. If the gadget driver detects that there is no more paper, it interrupts the print action and returns a `-ENOSPC` message to the host, which then triggers the `collect_error` statement to handle subsequent actions, such as displaying a warning message on the user's screen.

In the end, the entire system reaches a "no paper, stop working" state, which necessitates collaboration between the host and gadget. If the existing fuzzing methods test this case, the works with userspace injection repeatedly send the `usblp_write()` system call to the printer, leading to a valid working status. Other works that supply device-side inputs can only respond to requests, without the host's polling for paper status, the entire system may not reach a paper-required state at all. Like this situation, existing works experience difficulties in reaching these complex kernel statuses designed for the host and device's interactions, as they only generate inputs from one side, without adequate capture of the synergistic relationship between the two sides.

```
1  static __poll_t usblp_poll(struct file *file, struct
   ↪ poll_table_struct *wait)
2  {
3      struct usblp *usblp = file->private_data;
4      __poll_t ret = 0;
5      unsigned long flags;
6
7      poll_wait(file, &usblp->rwait, wait);
8      poll_wait(file, &usblp->wwait, wait);
9
10     mutex_lock(&usblp->mut);
11     if (!usblp->present)
12         ret|= EPOLLHUP;
13     mutex_unlock(&usblp->mut);
14
15     spin_lock_irqsave(&usblp->lock, flags);
16     if (usblp->bidir && usblp->rcomplete)
17         ret |= EPOLLIN  | EPOLLRDNORM;
18     if (usblp->no_paper || usblp->wcomplete)
19         ret |= EPOLLOUT | EPOLLWRNORM;
20     spin_unlock_irqrestore(&usblp->lock, flags);
21     return ret;
22 }
```

Listing 1: This patch fixes a long-existing bug in the USB subsystem that hangs in `poll()` if the device is disconnected [40]. This vulnerability existed in Linux for 15 years and was not discovered until 2021.

The patch depicted by Listing 1 demonstrates a vulnerability that requires interaction from both sides. The code fixes the issue by adding Lines 10-13 highlighted in grey. Specifically, the vulnerability occurs when the peripheral device is suddenly disconnected after a successful connection with the device file opened. If a userspace program subsequently invokes the `poll()` system call, the `usblp` driver hangs and cannot be awakened. The root cause of the

stall is the lack of checking of the status of the peripheral before acquiring the `usblp->lock`, and after getting the lock (Line 15), it has not been able to wait for a response from the peripheral (Lines 16-19) to unlock (Line 20), thus blocking the thread here. This straightforward but complex logic requires the combined action of both sides to trigger correctly. Furthermore, such a large kernel may have various vulnerabilities under more complex conditions, which are difficult to detect with existing methods.

Moreover, existing USB fuzzing methods face a challenge in generating high-quality inputs that can trigger effective state transitions within host drivers, which in turn limits their fuzzing performance. Specifically, they mostly emulate approaches where the host obtains data from the device and randomly feeds it back in those ways, with little consideration of their functionality. For instance, PERISCOPE [29] treats each fuzzer-generated input as a serialized sequence of memory accesses and substitutes the device drivers' read values from MMIO and DMA mappings. However, the operating system kernel strictly adheres to various general-purpose protocols with stringent input validation mechanisms, and values that do not satisfy the specifications are immediately rejected and entered into an error-handling mechanism. As a result, PERISCOPE could not discover any more bugs after testing with ten thousand inputs, and some shallow bugs were repeatedly triggered. Other works [16], [42] that emulate the device aim to pass or satisfy complex verifications during the initialization phase and present some emulation of the basic functions. They are unable to generate data that can effectively alter the overall state of the system.

```
1  static long
2  printer_ioctl(struct file *fd, unsigned int code,
3                unsigned long arg)
4  {
5      ...
6      switch (code) {
7      case GADGET_GET_PRINTER_STATUS:
8          status = (int)dev->printer_status;
9          break;
10     case GADGET_SET_PRINTER_STATUS:
11         dev->printer_status = (u8)arg;
12         break;
13     ...
14     }
15     ...
16 }
```

Listing 2: Part of `printer_ioctl()` code, where gadget module provides easy-to-use interfaces to change status.

Compared to these approaches, the gadget subsystem in the Linux kernel, which serves as the device side, can bring more complex state changes to the whole system, since it emulates various functionalities on the device side. Its comprehensive and sophisticated state transition mechanism and easy-to-use filesystem interfaces benefit us in exploring more status. For instance, in the previous example (Figure 2), setting the third parameter `arg` of the `ioctl()` system call to `PRINTER_PAPER_EMPTY` when injecting it into the gadget device file transforms the printer from a normal working state to a no paper state, as shown in Listing 2. This illustrates the gadget module's capability to uncover more complex states in the kernel.

3. Design

To address the concerns mentioned above, we propose SATURN, a USB driver fuzzing mechanism that facilitates the synergistic effects between the host and the gadget side, dedicated to covering the full handling chain during USB communication and exploring deep code logic. Figure 3 illustrates the design of SATURN, which consists of two stages: gadget attachment (Section 3.1) and synergistic fuzzing (Section 3.2). In the gadget attachment stage, SATURN directly extracts the possible configuration space from the kernel and utilizes dynamically attached gadget devices to respond to initialization requests, so as to increase the success rate of driver initialization. In the synergistic fuzzing stage, SATURN enhances its approach by probing the underlying file operations of both host and gadget drivers, grouping syscalls according to gadget types, and fixing parameters if necessary, thus significantly increasing the generated syscall sequences' quality.

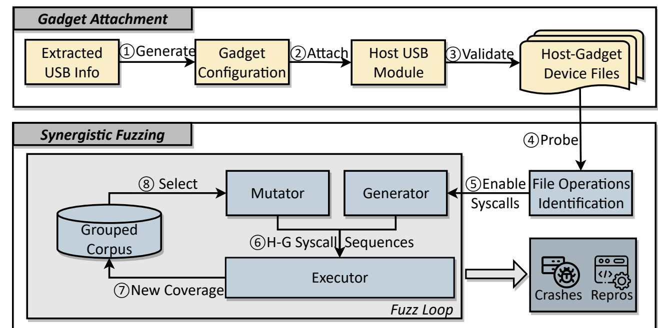

Figure 3: SATURN Overview. With the extracted USB information, SATURN configures the gadget devices (①) and attaches it to the host (②). After validating the attachment and monitoring the device files on both sides (③), SATURN utilizes a probe to identify the file operations (④), then enables proper system calls in the fuzzing loop (⑤), and effectively generates system call sequences acting on both side devices (⑥). Corpus is grouped by gadget functionality to improve fuzzing efficiency (⑦, ⑧).

3.1. Gadget Attachment

To improve the success rate of driver initialization and facilitate the transition of drivers to their respective interactions, we propose dynamically attaching gadget devices and constructing the complete attachment procedure until it can be correctly identified to pass the USB enumeration process. However, the gadget module has a vast configuration space and randomly generated data often fails to pass legitimacy checks. Therefore, SATURN utilizes the USB information in the kernel, trying to generate the valid gadget configuration. Specifically, *ConfigFS* [2] is adopted to configure gadget

devices, which provides a programmable interface to configure and create the gadget devices. Then, access to the host and gadget drivers is achieved through device files, which indicate how userspace programs access specific hardware devices in Unix-like kernels. While device file locations are dynamically generated upon successful attachments, it is essential to have a reliable method for detecting file nodes. During attachment, SATURN validates the gadget's availability and detects corresponding device files.

USB IDs Extraction. To support hot-plugging and automatic driver-matching functionalities, Linux hardcodes some device-related values and constraints into the `usb_device_id` structures. When a device tries to connect, the host-side driver core compares its attributes, such as vendor and product ID, with those listed in the drivers' USB IDs fields to determine the appropriate driver for the device. SATURN leverages this feature to dynamically set up the gadget as a specific device, allowing it to pass the USB enumeration phase and trigger the corresponding drivers. In the extraction module, SATURN leverages the iterator to traverse the USB driver bus and access these values, which are static constants listed in each specific driver's implementation and linked to the bus through the `usb_register()` function. Then SATURN constructs a callback function that obtains the `usb_device_id` structure from the USB bus at each callback and continuously dumps them in a certain format. Since most built-in drivers are registered during kernel boot, SATURN can extract and collect this information upon loading as a kernel module during runtime.

Gadget Configuration Generation. Based on the extracted USB device ID structures, SATURN generates gadget configurations, including device attributes and interface properties, capturing the available functionalities and their corresponding features. These USB device ID structures consist of candidate attribute values for matching and a bitmap field named `match_flags`, which plays a critical role in determining the fields necessary for device matching. When constructing device attributes, in addition to the strictly limited fields that are identified through bit-level exclusive operations, SATURN randomly generates extraneous fields following their protocol specifications, augmenting the diversity of the device attributes. To determine the suitable interface properties, SATURN selects the appropriate function based on the interface-related fields and the semantic information they convey. For instance, SATURN maps macro definitions to generate corresponding `interfaceClass` values for devices. The utilization of dynamic generation strategies enables SATURN to create comprehensive and precise device configurations, thereby allowing the efficient trigger of appropriate driver types.

Gadget Availability Validation. To account for potential attachment failures because of special hardware requirements, SATURN validates the availability of the gadget device. Upon configuration, the gadget is enabled and bound to a specific USB Device Controller (UDC), with the properties written to the kernel space for handling by the gadget driver during the enumeration stage. Then beginning with the host driver's probe mechanism, which involves hardware initialization, resource allocation, and other actions as per the USB protocol specifications. To confirm that the device is successfully attached, SATURN checks its availability by validating its attributes using *syfs* [17], which exports kernel data structures to the userspace. Specifically, to accomplish this, SATURN monitors all registered device classes, including host-side and gadget-side, to determine if an object matches the attached gadget. If the verification process times out, the device connection is considered to have failed.

Upon successful device attachment, SATURN locates the device files on both sides in the virtual filesystem. Since these files are actively created by the userspace program *udev* [15] rather than in the kernel space, SATURN obtains the major and minor numbers to trace their locations at runtime, which represents a particular driver and a device index, respectively. By combining device numbers, SATURN is capable of directly referencing a unique device file by monitoring the corresponding HCD/UDC, as each device is associated with a unique one. With the device numbers, SATURN backtracks their paths in the virtual filesystem to obtain the actual locations on both sides.

3.2. Synergistic Fuzzing

To expand the scope of potential vulnerabilities and explore a wider range of states during USB communication, SATURN introduces the idea of leveraging the synergistic capabilities of the host and gadget sides through system call sequences injection to produce canonical operations. SATURN proposes such a synergistic fuzzing scheme that identifies cooperative functions and control structures, thus allowing the states to evolve using canonical operations and command sequences. The process for generating these system call sequences is depicted in Figure 4. Upon successful validation of the gadget attachment, SATURN identifies the device files for both sides and attempts to construct appropriate inputs. However, each driver has its unique operations for interacting with the device file, which is crucial for fuzzing as it determines the input structures that the driver requires. To address this, SATURN utilizes a probe to dynamically identify these operations from the file operations structure, which consists of function pointers. These operations are then mapped to corresponding system calls to generate sequences, which are injected into the kernel space to trigger the fuzzing process. When the fuzzing campaign is underway, SATURN adopts a persistent fuzz loop for these dynamically attached drivers, until no new covered code appears or the timeout is reached.

File Operations Identification. Each device driver has its own concrete system call implementations with varying data types and potential values of the parameters. For instance, the v4l2 driver and mass storage driver have differing formats for the data encoded in the same `write()` method. Therefore, it is crucial to generate inputs that are valid for each driver based on their respective implementations. Traditional kernel fuzzers, such as Syzkaller and Moonshine [19], hardcode the file path and system call parameters in the pre-prepared description for pre-attached

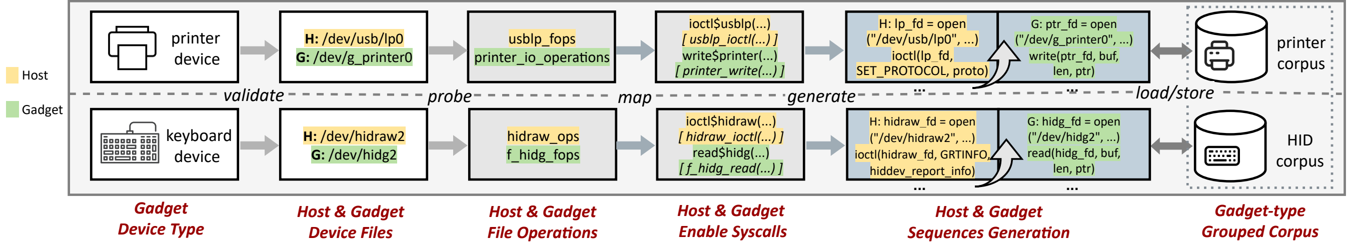

Figure 4: Example of the syscall sequence generation procedure. When a gadget is attached, SATURN detects the device files of the host side (e.g. `/dev/usb/lp0`) and gadget side (e.g. `/dev/g_printer0`). Then, SATURN utilizes the probe to identify the file operations on each side (e.g., `usblp_fops`, and `printer_io_operations`). Subsequently, SATURN maps the file operations to their corresponding syscall descriptions to generate parameters correctly. Finally, SATURN employs specific generation strategies or mutation techniques based on function-type grouped corpus to produce syscall sequences that integrate the operations performed on both the host and gadget sides.

drivers, thus constraining the input generation and improving the accuracy of test cases. However, the information encoded in existing descriptions, e.g., file paths and input structures, may be inconsistent with dynamically attached devices. Consequently, SATURN cannot rely on these static templates, and we need to identify the corresponding device file operations at runtime.

SATURN employs a probe tracing method with *kallsyms* and *kcov* module for precisely identifying these implementations. The *kallsyms* provides the symbolic tables and stack traces in the kernel, and *kcov* assists in tracing covered system calls. In practice, the implementations of these system calls are typically stored as function pointers in structures named `file_operations` for the driver to index them. To extract these structures, SATURN first opens the device file and executes a general `read()` system call. Next, the program handler is transferred to the virtual file system and SATURN follows the execution of this system call with *kcov* to obtain the program counter. Using the program counter values, SATURN filters *kallsyms* to identify the entire call chain and locate the actual implementation of the system call, which is then traced back to the file operations structure corresponding to the device driver.

A function mapping table is created to map the file operations to the respective system call descriptions, which allows fuzzers to generate corresponding sequences accordingly. Under the fuzzing campaign, SATURN probes the file operations of the attached devices on both sides and references the table to extract their corresponding system call descriptions. For instance, when attaching the host-side `usblp` driver as shown in Figure 4, SATURN probes the file operation `usblp_fops` through the device file `/dev/usb/lp0`. When generating system call sequences, SATURN matches the corresponding system call description, such as `ioctl$usblp` whose underlying implementation is `usblp_ioctl()`, and leverages the structural characteristics of their input parameters to generate inputs.

Host-Gadget Sequences Generation. SATURN triggers the interactive actions between the host and gadget drivers by injecting system call sequences containing operations on both sides into the executor, allowing the fuzzer to efficiently explore the driver's state spaces. System calls within a sequence are executed by multiple threads simultaneously, thus enabling a coordinated interaction on both sides. The initial step of the sequence generation is to obtain the file descriptor by opening the device files, which serves as an index to an entry in the process's table of open files and is referenced in subsequent system calls. Therefore, SATURN initiates the sequences with the `open()` system call, as depicted in Figure 4. Based on the system calls enabled for specific device files and existing file descriptors from the previous step, SATURN generates the corresponding operations on both sides, facilitating the exploration of driver state spaces and detection of original bugs.

To enhance the mutation effectiveness, SATURN categorizes its corpus according to the functional type of gadget. Unlike other fuzzers, which deposit test cases into the corpus for always-loaded drivers when they touch new code, SATURN targets the dynamically attached/detached driver in each cycle of the fuzzing loop. Consequently, picking a seed from the corpus may lead to irrelevant file operations for the current devices, resulting in ineffective seeds. To overcome this problem, SATURN divides the corpus into different groups by the functional type of gadget. When generating a gadget configuration, SATURN notes the function types and utilizes them as attributes for subsequent operation sequences. When synchronizing seeds, SATURN pulls from other corpus and maps them to local groups based on tags.

During the mutation phase, SATURN chooses a system call sequence randomly from the corpus that shares the same function type as the initial seed, and then alters its parameters, order, or adds and removes system calls. Although these mutated operations are related to current drivers in terms of function types, the names of the opened devices are likely to have changed. To address this issue, SATURN applies a fixup for this system call sequence based on the attributes of dynamically attached device files, replacing the old device information before handing it over to the executor.

4. Implementation

We implemented SATURN based on the state-of-the-art kernel fuzzer Syzkaller. We customized its components to

make it suitable for synergistic fuzzing while reusing the underlying functionalities, such as code coverage collection and corpus synchronization mechanisms.

Gadget Attachment. We dynamically load the built kernel module into a runtime kernel through `insmod` to extract the device driver matching information on the USB and HID buses. The gadget attach process is encapsulated into a pseudo system call in userspace with the *libusbgx* [18], a C library wrapper for the kernel USB gadget-configfs userspace API functionality. In the implementation of this pseudo system call, the first step is to detach any previously bound gadget to this USB Device Controller, as it can only support one device's functionality at a time. Then we feed the device attributes and interface properties and finally enable the gadget. Our implementation involves enabling almost all function types of gadget devices and using the DUMMY HCD/UDC module as a software connection between the host and gadget. When the fuzzing campaign is underway, a separate thread is created to verify device attachment and monitor device files, which reports the host-gadget device list to the fuzzer main thread upon detection.

Synergistic Fuzzing. We implement the aforementioned sequence generation process, including the gadget type grouped corpus which can synchronize and transfer between different instances. To precisely generate syscall sequences that satisfy the type and range constraints of system call inputs, we reuse existing system call descriptions of Syzkaller. As mentioned before, the file operations of a device file determine the way to access it and correspond to specific system calls. We probe the file operations of dynamically attached device files by extracting the executed kernel functions with *kcov*, which enables SATURN to capture an accurate execution path of the invoked system call. We construct a table to map file operations to the respective system call descriptions, which is generated during the dry-run Syzkaller by utilizing the file operations identification approach mentioned above. Then we reference the table to extract their corresponding system call descriptions, which provide precise information about the system call parameters, and utilize them to guide the generation.

5. Evaluation

In order to showcase the efficacy of SATURN, a series of experiments are conducted on recent versions of the Linux kernel. First, we demonstrate SATURN's capability to explore more execution paths and kernel space status on both the host and gadget sides respectively, by comparing its overall coverage to existing research. Second, we evaluate the contributions of each component to its ability to cover the USB driver code. Finally, we exhibited SATURN's vulnerability detection capabilities by listing previously-unknown bugs and presenting case studies on its findings. We design experiments to address the following questions:

- **RQ1:** How effective is SATURN in improving coverage on the host and gadget stacks, respectively?
- **RQ2:** What are the respective contributions of the gadget attachment and synergistic fuzzing to the overall coverage improvement?
- **RQ3:** How does SATURN perform in vulnerability detection, including host and gadget drivers?

Hardware and Software Environment. We conduct our experiments on a computer with an AMD EPYC 7742 64-Core processor, 256 GiB of memory, and running 64-bit Ubuntu 20.04.2 LTS. The compilers used to build the respective kernels are GCC 12.2 and LLVM 14.0. The experiments are configured with identical parameters in terms of QEMU configurations and other settings. Specifically, for strict control of computing resources, we start all experiments simultaneously and distributed the resources evenly, including 2 cores and 2 GiB of memory for each virtual machine. To reduce statistical errors, each experiment is repeated 10 times, and the average results are reported.

Guest OS Preparation. We base our evaluation on the latest Linux kernel version and various long-time support kernel versions, ranging from v5.5 to v6.0. The host-side relevant USB driver configurations are enabled and compiled into the kernel binary. For the gadget side, we support 13 gadget function types and their corresponding interface attributes that allow synchronous transmission, including `USBG_F_MIDI`, `USBG_F_HID` and `USBG_F_PRINTER`, and don't activate any legacy gadget device. This supports attaching and triggering different host and gadget drivers by synthesizing different attribute configurations. We also enable kcov (`CONFIG_KCOV`) to collect code coverage, Kernel Address Sanitizer (`CONFIG_KASAN`) to detect kernel memory bugs, and USB Dummy HCD/UDC (`CONFIG_USB_DUMMY_HCD`) to serve as virtual hardware. In addition, we have configured different kernel compilation options to satisfy the requirements of projects, such as `CONFIG_USB_RAW_GADGET` for Syzkaller, as well as some gadget function-related options for SATURN.

5.1. Coverage Improvement

To answer **RQ1**, and to demonstrate the capability of SATURN to initialize more drivers and effectively explore the interaction states simultaneously, we present the overall code coverage statistics in comparison with existing state-of-the-art fuzzers. The coverage collection is conducted separately on the host stack and the gadget stack, with only one-side stack being instrumented. To evaluate the effectiveness of SATURN compared to other host-side fuzzers, we select Syzkaller USB fuzzing module [13] and USBFuzz [22] as experimental subjects. We choose these two fuzzers because of their proven effectiveness in real-world scenarios, as well as the discovery of plenty of vulnerabilities through their application. For the gadget side, we opt for FUZZUSB [12], which is the first USB fuzzing method for the USB gadget system and has successfully identified 34 previously unknown vulnerabilities. To ensure fairness, we perform 10 repeated experiments without any initial seeds.

Host-side Comparison. We conduct a comparison of code coverage statistics on the USB host stack by solely in-

strumenting USB host-related code (including the USB core framework, host controller drivers and USB device drivers) with SATURN, Syzkaller, and USBFuzz. Syzkaller is capable of externally fuzzing the USB subsystem by defining a set of pseudo-syscalls that can connect USB devices and transmit or receive control and non-control messages from the device to the host. In comparison to Syzkaller, we perform experiments on four different kernel versions, including v5.10, v5.15, v5.19, and v6.0, which were either the latest or stable versions at the time of submission. USBFuzz utilizes a software-emulated USB device to provide random device data to drivers when they perform IO operations and patches the kernel's assembly code to facilitate coverage collection, which has changed considerably with the newer kernels. To ensure consistency with USBFuzz, we utilize Linux v5.5 in our experiments [21]. We sample coverage statistics every 10 seconds during each campaign and compute the average value of each fuzzer's sampled data over its 10 executions.

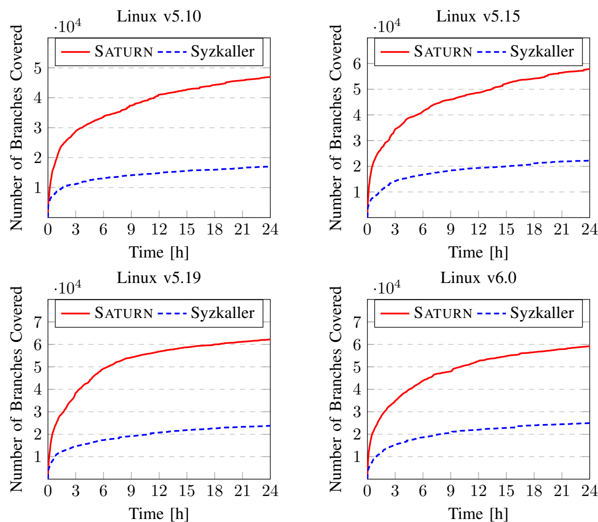

Figure 5: Coverage comparison between SATURN and Syzkaller over 24 hours. In all four versions, SATURN can achieve higher coverage statistics with less time.

Figure 5 presents a comparison between Syzkaller and SATURN in terms of their coverage statistics, demonstrating that SATURN can achieve higher coverage in the same amount of time. At the beginning of each experiment, SATURN has significantly higher coverage improvement than Syzkaller since it handles the enumeration process entirely in userspace, resulting in a lower success rate of device attachments. This leads to Syzkaller repeatedly encountering matching and probing logic before proper initialization. While both tools register significant coverage escalation within the first six hours, there is a noticeable deceleration thereafter. Nevertheless, SATURN's coverage grows at a faster rate than Syzkaller, indicating that the interaction process reaches the kernel state more quickly.

Table 1 presents the coverage improvement statistics achieved by SATURN in comparison with Syzkaller and USBFuzz. In terms of the comparison with USBFuzz, SATURN achieves 3.69× coverage improvement on Linux v5.5,

TABLE 1: Branch coverage statistics of SATURN, USBFuzz and Syzkaller on the respective kernels.

Version	Syzkaller	USBFuzz	SATURN	Improvement
5.5	20602	10382	48728	1.37× / 3.69×
5.10	17018	-	46950	1.76×
5.15	22186	-	57842	1.61×
5.19	23789	-	62172	1.61×
6.0	24939	-	59107	1.37×
Average	21707	10382	54959	1.53× / 3.69×

following the experiment settings of USBFuzz. As the limitations of the USBFuzz implementation mentioned above, the coverage statistics are not available for the other kernel versions tested with Syzkaller and SATURN. We perform a modular analysis of the code they cover and observe that USBFuzz can cover more on the host controller drivers, as it leverages an emulated device to feed generated inputs to drivers while SATURN directly utilizes the dummy HCD driver. However, SATURN can cover more code on the device drivers, and the USB core that contains common routines for handling data from the device side.

The increase in coverage statistics indicates that SATURN is capable of exploring more kernel states, which can be attributed to two factors. First, the gadget attachment mechanism enables the kernel to trigger more host-side drivers, resulting in potentially hitting a larger part of the kernel code. Furthermore, the synergistic fuzzing strategy significantly boosts overall efficiency. Through the file operation probing method, SATURN precisely generates the system call sequences that satisfy the type checks and constraints for dynamically attached device files. Under such circumstances, SATURN achieves efficient fuzzing by overcoming many complex sanity checks within the kernel and emulating the interactive communication process, thereby testing the more profound kernel logic.

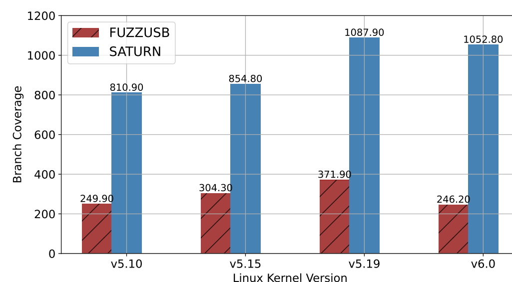

Figure 6: Coverage comparison of USB gadget stack between SATURN and FUZZUSB over 24 hours. In all four versions, SATURN can achieve higher coverage statistics.

Gadget-side Comparison with FUZZUSB. Here we break down the detailed coverage statistic to further evaluate the effectiveness of the USB gadget stack. Specifically, FUZZUSB is the first USB fuzzing technique for a USB gadget system, which achieves state-guided fuzzing upon gadget-specific state machines. We compile the Linux kernel with only the gadget code instrumented partially and compare their branch coverage. Figure 6 shows the overall cov-

erage statistic after ten runs of 24-hours experiments. As we can infer from the figure, SATURN outperforms FUZZUSB after 24 hours, which has about 2.3× coverage improvement on average and about 2.2×, 1.8×, 1.9×, and 3.3× coverage improvement on four different kernel versions compared with FUZZUSB separately, on the gadget side alone. The significant improvement in coverage statistics achieved by SATURN can primarily be attributed to its synergistic fuzzing mechanism. Both host-side driver and device logic operations affect the state transition of the gadget driver, therefore sending input in one direction as what FUZZUSB adopts cannot completely cover the state machine inside the entire gadget. In contrast, SATURN's interactive testing approach yields a more complex state transition mechanism, leading to the detection of more state space in the gadget driver.

We further explore the logic of the uncovered code in a systematic way. Generally speaking, a USB driver's lifecycle can be generalized into eight states: register, probe, bind, active, suspend, resume, unbind, and unregister. SATURN targets the device interaction relevant states (active/suspend/resume) and precisely generates syscalls to facilitate state transition, whereas the driver initialization relevant states (probe/bind/unbind) are covered during gadget attachment. Therefore, SATURN covers all aforementioned stages, but since we do not target non-interaction stages, their coverage statistics will be relatively less improved.

5.2. Contribution of Each Component

Both the greater number of drivers successfully triggered and improved coverage per driver contribute to coverage improvements. We present the aggregated statistics above as SATURN aims to initialize more drivers and effectively explore the interaction simultaneously. To address **RQ2**, we try to break down the overall improvement into two parts: one from attaching more drivers (gadget attachment) and the other from fuzzing existing drivers supported by other fuzzers (synergistic fuzzing) and evaluate the effectiveness of each component to present their respective contributions.

Effectiveness of Gadget Attachment. The goal of the experiment is to evaluate how many types of corresponding host-side drivers can be triggered by our proposed approach, as well as the success rate of the attachment. The former is crucial because driver initialization is a prerequisite for the communication phase and enables the state of such drivers can transition to those involved in their respective devices' interactions. The attachment success rate is a key performance indicator, as it is closely related to testing efficiency. SATURN utilizes the same types of gadget-side drivers with FUZZUSB, so we compare the host side in this aspect. We conduct the experiments by continuously trying to attach devices at a given time interval without any communication process. Specifically, we dry ran Syzkaller, USBFuzz and SATURN by only enabling the device attachment process in 24 hours, and the parallelism of experiments is restricted, ensuring that only one device was trying to attach at the same time. *sysfs* is adopted to validate the host driver's availability, as mentioned in the Section 3.1.

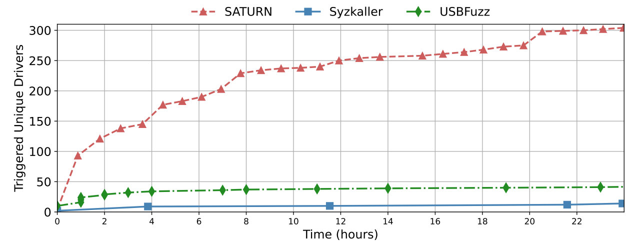

Figure 7: Number of successfully triggered host-side driver types after dry run device attachment process over 24 hours.

As shown in Figure 7, SATURN attempted to attach devices 16756 times and successfully initialized 13670 times with a success rate of 81.58%, triggering 304 unique drivers. In comparison, Syzkaller attempted 15548 device attachments with a 38.91% success rate and triggered 14 drivers, while USBFuzz initialized 44 drivers with an unavailable success rate as it utilized binary files as seeds. The range of host-side drivers triggered by SATURN allows it to explore diverse modules in the subsequent communication process. Subsequently, we manually analyzed some failed situations and found that they were primarily due to protocol-specific data format or port requirements not being specifically implemented in the Linux gadget module.

To further illustrate the coverage improvement gained due to the gadget attachment, we measured the coverage achieved during the dry-run attachment process. SATURN covers 19060 branches with 304 triggered drivers compared to Syzkaller's 17565 with 14 drivers during driver attachment (we experienced difficulties in separating coverage collection for USBFuzz's different USB stages as it utilizes binary files as seeds). The 9% improvement is due to running similar code regardless of successful attachment. Then we deduce that the interaction stage for Syzkaller and SATURN covers 3037, and 29668 branches, respectively. Here, SATURN's coverage of the uniquely attached drivers is more than 23000. This demonstrates the attachment component's contribution since the last figure is purely made possible by attaching significantly more drivers than the comparison.

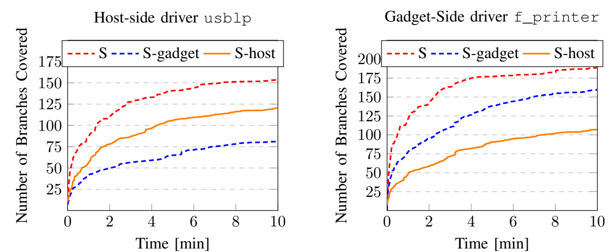

Figure 8: Coverage comparison of *S* (SATURN), *S-host* (injection from the host), *S-gadget* (injection from the gadget). SATURN achieves a high coverage improvement for both host-side and gadget-side drivers.

Effectiveness of Synergistic Fuzzing. First, we conducted an evaluation comparing the one-side injection with synergistic testing. To accurately collect the code covered

by the host-side and device-side drivers, we take the printer as an instance, with the class driver `usblp` on the host side and driver `f_printer` on the gadget side. SATURN is denoted as *S*, *S-gadget* and *S-host* indicate input only from the device and host side, respectively.

The experiment results are shown in Figure 8. The synergistic fuzzing strategy improves coverage by 27.57% and 68.57% for the host-side driver `usblp`, 76.82% and 18.37% for the device-side driver `f_printer`, respectively, compared to one-side injections. We analyzed the results and found that some generic functions can be triggered regardless of the inputs' direction, such as `usblp_probe()` and `usblp_resume()` in the `usblp` driver. Some functions can only be triggered by inputs in a specific direction, e.g. `usblp_ioctl()` can only be activated from the host side.

To further validate the coverage improvement from the synergistic fuzzing approach, we extracted and compared the branch coverage of 11 common drivers among Syzkaller, USBFuzz, and SATURN with the assistance of *kcov*, as depicted in Table 2. We observe that synergistic fuzzing delivers a 58% and 77% average improvement, respectively. We also experimented with running Syzkaller on these drivers for 48 hours, yielding a coverage improvement of 4%. Thus synergistic fuzzing approach proposed by SATURN can easily cover logic that Syzkaller has very little luck in and we hope the coverage improvement can be further extended to other drivers that SATURN can attach. We investigated additionally the reasons behind this and noted that the input generated by Syzkaller is partially initializing the states of USB interaction through one-side injection, whereas SATURN synthesizes inputs of both sides conforming to relevant constraints during synergistic fuzzing.

TABLE 2: Coverage of the common drivers among Syzkaller, USBFuzz, and SATURN over a 24-hour period.

Driver Name	Total Blocks	Syzkaller	USBFuzz	SATURN	Impr vs Syzkaller	Impr vs USBFuzz
usblp	262	58	47	155	+168%	+228%
usbhid	447	130	107	206	+59%	+92%
bcm5974	128	41	35	49	+19%	+41%
appletouch	184	37	39	42	+15%	+10%
fdti_sio	465	79	60	153	+94%	+154%
option	44	14	14	16	+16%	+19%
cp210x	369	89	92	114	+29%	+24%
ath9k_htc	308	99	74	117	+19%	+58%
sierra	195	39	51	94	+140%	+85%
cdc_ether	151	76	66	94	+24%	+41%
ipw	37	6	7	9	+60%	+33%
Total	2590	665	592	1048	+58%	+77%

5.3. Vulnerability Detection

To address **RQ3**, we conducted a two-week testing of the Linux kernel using SATURN, which resulted in the identification of 26 previously unknown vulnerabilities, among which are 4 CVEs. The vulnerabilities included 19 in the host-side drivers and 7 in the gadget-side drivers, with the majority found in the device drivers, as well as some in the core of the USB host and gadget subsystem. Table 3 provides detailed information, along with their associated vulnerability type. Unlike memory issues, logic bugs and deadlocks have no corresponding sanitization for detection and reporting. We employ kernel assertions (e.g., `BUG_ON()`), which violate the intended program specifications to detect logic bugs, and other kernel features to detect and prevent deadlocks during fuzzing campaigns, such as `CONFIG_LOCKDEP`. Most of these vulnerabilities are critical, including many that remained undiscovered and unreported in the kernel code base for decades. Even with the persistent testing of Linux USB driver modules by tools like Syzkaller and various other kernel driver fuzzers, employing substantial computing resources, these vulnerabilities remained unidentified.

TABLE 3: SATURN has discovered 26 previously unknown vulnerabilities, among which are 4 CVEs, including 19 on the host side and 7 on the gadget side.

Devices	Side	Kernel Operation	Bug Type (CVE)
midi	gadget	f_midi_transmit	deadlock
net	host	ethtool_get_drvinfo	logic bug
core	host	usb_stor_pre_reset	deadlock **CVE-2022-3903**
core	gadget	gadgets_make	logic bug
scsi	host	scsi_device_unbusy	logic bug
gspca	host	gspca_init_transfer	out-of-bound
hid	host	holtek_kbd_input_event	null-ptr-defer
u_serial	gadget	gs_start_io	logic bug
udc	gadget	dummy_timer	use-after-free
nfc	host	port100_send_cmd_sync	logic bug
sound	host	snd_rawmidi_free	deadlock
dvb	host	digitv_ctrl_msg	out-of-bound
scsi	host	sg_release	use-after-free
keyspan	host	keyspan_close	use-after-free
dvb	host	su3000_i2c_transfer	logic bug
sound	host	snd_rawmidi_receive	use-after-free
dvb	host	digitv_i2c_xfer	logic bug
dvb	host	dvb_frontend_get_event	logic bug
core	gadget	usb_composite_setup_continue	logic bug **CVE-2022-4662**
videobuf2	host	vb2_start_streaming	logic bug
f_hid	gadget	f_hidg_read	use-after-free
udc	gadget	usb_ep_queue	logic bug
sound	host	release_urbs	logic bug **CVE-2022-44042**
filesystem	host	filp_close	use-after-free
sound	host	snd_card_new	invalid free **CVE-2022-44041**
hid	host	roccat_disconnect	use-after-free

The ability of SATURN to identify previously unknown vulnerabilities can be attributed primarily to its proposed gadget attachment and synergistic fuzzing strategy. The gadget attachment mechanism allows the Linux kernel to interact with more types of USB drivers, enabling the fuzzer to expose bugs within these drivers. Additionally, the synergistic fuzzing strategy allows for higher-quality test case generation during the interaction process, allowing the fuzzer to bypass certain restrictive security checks within the kernel, leading to an expansion in the number of states explored and exposure of bugs in such logic. The above findings demonstrate that SATURN can significantly enhance the vulnerability detection capabilities of USB drivers.

5.4. Case Study

In this section, we further analyze the potential causes and effects of these uncovered vulnerabilities.

Case Study 1. Listing 3 depicts a double-free vulnerability discovered in the Advanced Linux Sound Architecture (ALSA) subsystem of version 6.0. This vulnerability is triggered by SATURN's gadget attachment mechanism and has been fixed by maintainers. During the initialization of a sound card instance, the kernel first invokes the `snd_card_new()` function and subsequently calls `snd_card_init()` function (Line 9). If the `snd_card_init()` fails, the system jumps to error processing logic (Line 20). In this scenario, the `put_device()` function (Line 21) triggers the device release logic and calls `kfree()` function to free the card instance first, then the system calls `kfree()` function again when returns to the caller `snd_card_new()` (Line 5). As a result, a double-free vulnerability arises, which, if exploited, could result in a denial-of-service (DoS) attack or the execution of arbitrary code on the system.

The vulnerability occurs in the gadget attachment stage. Other fuzzers that only perform randomized device attachments are unlikely to satisfy the match/probe processes and trigger any `kfree()` invocations during the initialization of the `snd_card` device, while SATURN was able to match on the `snd_card` driver and attempt to access internal initialization logic based on the generated configuration.

```
1  int snd_card_new(struct device *parent, int idx,
   ↪   const char *xid, ...)
2      ...
3      err = snd_card_init(card, parent, idx, xid,
   ↪   module, extra_size);
4      if (err < 0)
5          kfree(card);
6          return err;
7      }
8
9  static int snd_card_init(struct snd_card *card,
   ↪   struct device *parent, ...)
10     ...
11     if (err < 0)
12         goto __error;
13     ...
14     err = snd_ctl_create(card);
15     if (err < 0) {
16         dev_err(parent, "unable...");
17         goto __error;
18     }
19     ...
20   __error:
21     put_device(&card->card_dev);
22     return err;
23     }
```

Listing 3: Code snippet of the double-free vulnerability in ALSA. The first `kfree()` occurs in `snd_card_init()` and the second occurs in `snd_card_new()`, triggering the double-free vulnerability during the gadget attachment.

Case Study 2. Listing 4 demonstrates a vulnerability in the Linux gadget subsystem of version 5.19. This vulnerability is identified during SATURN's synergistic fuzzing stage. After establishing a connection between the host and the mass storage gadget device, the `fsg_main_thread()` function handles most of the processing logic on the gadget side, including responding to host requests. During its execution, `fsg_main_thread()` calls `get_next_command()`, which eventually calls `usb_ep_queue()` to instruct the kernel to perform the specified request through an endpoint. However, if the gadget device disconnects suddenly, another thread will call `fsg_disable()` to disable the device. This function will throw an interrupt, but after the investigation by kernel maintainers, they found that it runs in an atomic context with no fine-grained synchronization mechanisms, allowing `fsg_main_thread()` to use the endpoints after disabling. Therefore, `usb_ep_queue()` function may attempt to access the disabled endpoint, triggering the `WARN_ON_ONCE` assertion (Line 16).

The vulnerability is located in the deeper layers of the kernel code and can only be exposed when the host and device are actively interacting. Upon successful initialization of the mass storage gadget, SATURN leverages its synergistic fuzzing mechanism to generate system call sequences that emulate requests from the host's userspace program, which then triggers the gadget drivers' main thread logic. At this point, the gadget-side system call sequences generate a disconnection request to the gadget device, thus triggering the vulnerability. This is a bug that can only be identified in the interaction logic between the host and gadget, and the synergistic fuzzing phase is instrumental in exploring the deeper states of both sides and uncovering such issues.

```
1  static int fsg_main_thread(void *common_)
2  {
3      /* The main loop */
4      while (common->state != FSG_STATE_TERMINATED) {
5          ...
6          if (get_next_command(common) ||
7              exception_in_progress(common))
8              continue;
9          ...
10     }
11 }
12
13 int usb_ep_queue(struct usb_ep *ep, ...)
14 {
15     ...
16     if (WARN_ON_ONCE(!ep->enabled && ep->address)) {
17         ret = -ESHUTDOWN;
18         goto out;
19     }
20     ...
21 }
```

Listing 4: Code snippet of a warn vulnerability in `usb_ep_queue()` during the communication stage between the host and gadget. The vulnerability is triggered by a lack of fine-grained synchronization mechanisms after the endpoint is disabled, leading to potential illegal access.

6. Related work

Device Emulation-based Testing. Emulation-based techniques are commonly used for vulnerability detection, employing programs that imitate physical devices' functionality to reduce costs and increase flexibility. QEMU [3] is a

well-known device emulator that supports a wide range of peripherals and is commonly utilized by kernel fuzzers [4], [39]. However, the increasing complexity and sheer amount of peripherals can overwhelm QEMU's emulation capabilities. To complement QEMU's inadequate ability to emulate a vast array of peripheral devices in specific testing scenarios, alternative emulation methods have been proposed, such as PrIntFuzz [16], which utilizes an automated device simulation that supports device probing, hardware interrupts, and I/O interception to emulate hundreds of devices for subsequent fuzzing. Similarly, DR. FUZZ [42] is a semantic-informed mechanism that can efficiently generate inputs to construct relevant structures and pass the "validation chain" in initialization for subsequent device-free fuzzing.

Although current approaches in which model peripherals can complete the initialization phase, the emulation of adequate device functionalities remains a challenge, hindering the injection of legitimate data into the host side. Differently, SATURN utilizes the kernel's built-in gadget module as a peripheral, whose extensive feature implementation allows responding to a diverse range of host-side driver requests, providing advantages over QEMU-based fuzzers. Additionally, SATURN supports attaching and triggering various host and gadget drivers through synthesizing different attribute configurations, thereby expanding the scope of fuzzing.

USB Fuzzing. The kernel driver modules comprise an extensive code base, the majority of which is supplied by untrusted third-party vendors with suboptimal code quality, making security issues in these drivers critical. As a result, numerous testing approaches have been proposed, including static analysis [1], [33], [38], symbolic execution [7], [11], [20], [24], and fuzzing [9], [19], [25], [26], [30], [31], etc.

For USB driver testing, researchers can access two input surfaces: the userspace and the device-side operations. To inject from userspace, numerous kernel fuzzers fill system call sequences randomly to pass test payloads and triggers kernel crashes with assistance from diverse sanitizers [27], [28], [41]. On the other hand, some works aim to inject data from the device side directly, like FaceDancer [6], which utilizes programmable hardware as a peripheral to attack USB device drivers. However, they only inject from one side, making it difficult to test the deep processing logic between two sides' interactions. To address this issue, SATURN proposes a host-gadget synergistic fuzzing approach, allowing the host and gadget to play their essential roles in the testing process. Works in other areas are also intended to discover communication vulnerabilities among multiple components, such as KARONTE [23] utilizes static analysis to identify and track shared data in firmware.

In addition, SATURN focuses on threats occurring during the communication process after the USB device has connected to the host, while other research works [22], [34], [35] emphasize the connection establishment process, due to the implicit trust characteristics of USB protocols. For example, research [35] extracts offensive and defensive primitives that operate across layers of communication within the USB ecosystem and finds that USB attacks often abuse the trust-by-default nature. USBFuzz [22] emulates a USB device that is virtually attached and detached from the target system to detect these threats. USBFILTER [34] can trace individual USB packets back to their respective processes and block unauthorized access to any device.

7. Discussion

We have demonstrated the effectiveness of SATURN. In this section, we describe the limitations of our current implementation and potential solutions.

Crash Reproducibility. Although SATURN has identified 26 previously unknown vulnerabilities, it is incapable of reproducing kernel crashes in some cases. Crash reproduction is still an open research problem for which no definitive solution has been found for two main reasons. First, the Linux kernel is a concurrent, stateful system that runs multiple threads in both the userspace and kernel space, which causes indeterminism and state accumulation. Second, the limited memory capacity restricts the number of system call sequences that kernel fuzzers can record for crash reproduction. As a result, crashes that necessitate complex combinations of system calls may fail to reproduce because of lost call sequences. SATURN's implementation is based on the state-of-the-art Syzkaller, which implements some reproduction techniques that partially assisted us in reporting several bugs to kernel maintainers. In addition to the reproduction mechanism, we save the call stack information during crashes to aid in vulnerability analysis. In the future, we can use a monitor to compress and record more system call sequences executed by each attached device, allowing the fuzzer to analyze the kernel log more accurately during the reproduction process, resulting in a better-reproducing program to verify the triggered vulnerabilities.

False-positive Analysis and Manual Effort. In terms of the whole fuzzing campaign, kernel fuzzers, including SATURN, detect bugs by triggering real kernel crashes, which are monitored and recorded automatically, thus presenting a low false positive rate. From the perspective of invalid configurations in the gadget attachment stage, SATURN has an 81.58% success rate, which is higher than Syzkaller's 38.91% while maintaining a low false positive rate. Similar to traditional kernel fuzzing tools such as Syzkaller and USBFuzz, SATURN requires some manual preparation before the fuzzing campaign, which includes tasks like kernel compilation. Additionally, post-fuzzing processes, such as crash analysis and subsequent reporting to kernel developers, are essential. Apart from these standard procedures, SATURN conducts USB device driver fuzzing without human intervention. For instance, during the gadget attachment phase, after generating the gadget configuration and associating it with a particular USB device controller, SATURN employs `sysfs` to verify the condition of attachment. Upon successful validation, the system transitions to the synergistic fuzzing process. Conversely, SATURN utilizes extracted information from the kernel and updates the configuration, thus improving the success rate.

8. Conclusion

In this paper, we propose SATURN, a novel host-gadget synergistic USB driver fuzzing approach, aiming to cover the entire USB communication handling chain. First, SATURN systematically configures and attaches gadgets based on extracted driver information, thus triggering more driver types and facilitating the transition to interactive logic. Then, SATURN utilizes a persistent synergistic fuzzing process through canonical operation injection on the dynamically created device files of the host and gadget, enabling the exploration of the complex interaction logic on both sides.

We evaluate the effectiveness of SATURN on recent Linux kernels. Compared to state-of-the-art USB driver fuzzers such as Syzkaller and USBFuzz on the host side, as well as FUZZUSB on the gadget side, SATURN improves branch coverage by 1.53×, 3.69× and 2.3×, respectively. We then assessed the enhancements from each component. SATURN's driver attachment successfully activated 304 unique drivers, surpassing Syzkaller's 14 and USBFuzz's 44. Through a comparison of driver coverage across Syzkaller, USBFuzz, and SATURN, we found that synergistic fuzzing achieved improvements of 58% and 77%, respectively. These highlight SATURN's ability to initialize more drivers and effectively explore the interaction states. Additionally, SATURN successfully detected 26 previously unknown vulnerabilities, among which are 4 CVEs, including 19 on the host side and 7 on the gadget side. The above results demonstrate SATURN's capability to trigger complicated interactive logic between the host and the device sides and explore deep kernel states on the USB communication process.

9. Acknowledgement

We appreciate the reviewers' valuable and insightful comments. This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000, No2021QY0604, TC210H02S) and NSFC Program (No. 62022046, 92167101, U1911401, 62021002, U20A6003).

References

[1] Jia-Ju Bai, Tuo Li, Kangjie Lu, and Shi-Min Hu. Static detection of unsafe dma accesses in device drivers. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1629–1645, 2021.

[2] Joel Becker. configfs - Userspace-driven kernel object configuration. https://www.kernel.org/doc/Documentation/filesystems/configfs/configfs.txt. 2005.

[3] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, pages 10–5555. Califor-nia, USA, 2005.

[4] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2017.

[5] Andrey Konovalov Dmitry Vyukov. Syzkaller: an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller. 2015.

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

[7] Grant Hernandez, Farhaan Fowze, Dave Tian, Tuba Yavuz, and Kevin RB Butler. Firmusb: Vetting usb device firmware using domain informed symbolic execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2245–2262, 2017.

[8] John Hyde. Usb multi-role device design by example. *Comissioned by Cypress Semiconductors*, 2003.

[9] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE, 2019.

[10] Jakob Lell Karsten Nohl, Sascha Krißler. BadUSB — On accessories that turn evil. Black Hat. 2014.

[11] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. Hfl: Hybrid fuzzing on the linux kernel. In *NDSS*, 2020.

[12] Kyungtae Kim, Taegyu Kim, Ertza Warraich, Byoungyoung Lee, Kevin RB Butler, Antonio Bianchi, and Dave Jing Tian. Fuzzusb: Hybrid stateful fuzzing of usb gadget stacks. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2212–2229. IEEE, 2022.

[13] Andrey Konovalov. External USB fuzzing for Linux kernel. https://github.com/google/syzkaller/blob/master/docs/linux/external_fuzzing_usb.md. 2019.

[14] Greg Kroah-Hartman. Re: KASAN: use-after-free Write in keyspan_close. https://lore.kernel.org/all/YynnT7%2FmnzJVn7iz@kroah.com/.

[15] Greg Kroah-Hartman. udev–a userspace implementation of devfs. In *Proc. Linux Symposium*, pages 263–271. Citeseer, 2003.

[16] Zheyu Ma, Bodong Zhao, Letu Ren, Zheming Li, Siqi Ma, Xiapu Luo, and Chao Zhang. Printfuzz: fuzzing linux drivers via automated virtual device simulation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 404–416, 2022.

[17] Patrick Mochel. The sysfs filesystem. In *Linux Symposium*, page 313, 2005.

[18] Krzysztof Opasiak and Paweł Szewczyk. libusbg-neXt (libusbgx). https://github.com/linux-usb-gadgets/libusbgx.

[19] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing os fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 729–743, 2018.

[20] James Patrick-Evans, Lorenzo Cavallaro, and Johannes Kinder. Potus: Probing off-the-shelfusb drivers with symbolic fault injection. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[21] Hui Peng and Mathias Payer. USBFuzz: A Framework for fuzzing USB Drivers by Device Emulation. https://github.com/HexHive/USBFuzz. 2021.

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

[23] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1544–1561. IEEE, 2020.

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

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

[26] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl:hardware-assisted feedback fuzzing for os kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, 2017.

[27] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.

[28] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*, pages 62–71, 2009.

[29] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *2019 Network and Distributed Systems Security Symposium (NDSS)*, pages 1–15. Internet Society, 2019.

[30] Hao Sun, Yuheng Shen, Jianzhong Liu, Yiru Xu, and Yu Jiang. {KSG}: Augmenting kernel fuzzing with system call specification generation. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 351–366, 2022.

[31] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. Healer: Relation learning guided kernel fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 344–358, 2021.

[32] Syzkaller. CVE-2022-3239. https://nvd.nist.gov/vuln/detail/CVE-2022-3239. 2022.

[33] Xin Tan, Yuan Zhang, Xiyu Yang, Kangjie Lu, and Min Yang. Detecting kernel refcount bugs with two-dimensional consistency checking. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2471–2488, 2021.

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

[35] Jing Tian, Nolen Scaife, Deepak Kumar, Michael Bailey, Adam Bates, and Kevin Butler. Sok:" plug & pray" today–understanding usb insecurity in versions 1 through c. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 1032–1047. IEEE, 2018.

[36] USB-IF. USB on the Go and Embedded Host. https://www.usb.org/usb-on-the-go. 2012.

[37] usbkill.org. Official usb killer site. https://usbkill.com/. 2022.

[38] Qiushi Wu, Aditya Pakki, Navid Emamdoost, Stephen McCamant, and Kangjie Lu. Understanding and detecting disordered error handling with precise function pairing. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2041–2058, 2021.

[39] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1643–1660. IEEE, 2020.

[40] Pete Zaitcev. [PATCH] USB: usblp: fix a hang in poll() if disconnected. https://lore.kernel.org/all/20210303221053.1cf3313e@suzdal.zaitcev.lan/. 2021.

[41] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. Statefuzz: System call-basedstate-aware linux driver fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3273–3289, 2022.

[42] Wenjia Zhao, Kangjie Lu, Qiushi Wu, and Yong Qi. Semantic-informed driver fuzzing without both the hardware devices and the emulators. In *Network and Distributed Systems Security (NDSS) Symposium 2022*, 2022.

Appendix A. Meta-Review

A.1. Summary

This paper proposes Saturn, a USB fuzzer fuzzing both the host and the gadget sides at the same time by providing a sequence of syscalls from both sides, attaching different USB gadget drivers dynamically during fuzzing, and overcoming the blockers of initialization within USB host drivers to maximize the possibility of triggering more drivers loaded. Saturn outperforms state-of-the-art USB fuzzers, including Syzkaller, USBFuzz, and FuzzUSB.

A.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field
- Identifies an Impactful Vulnerability

A.3. Reasons for Acceptance

1) Saturn provides a new tool for USB fuzzing.
2) Saturn shows how fuzzing host/gadget at the same time benefits fuzzing each side.
3) Saturn detects new vulnerabilities within USB host/gadget stacks with better fuzzing throughputs.