Sönke Huster, Matthias Hollick, Jiska Classen: "To Boldly Go Where No Fuzzer Has Gone Before: Finding Bugs in Linux' Wireless Stacks through VirtIO Devices". Accepted for the 45th IEEE Symposium on Security and Privacy (S&P), 2024.

To Boldly Go Where No Fuzzer Has Gone Before: Finding Bugs in Linux' Wireless Stacks through VirtIO Devices

Sönke Huster*†, Matthias Hollick* and Jiska Classen*‡
*Secure Mobile Networking Lab (SEEMOO), TU Darmstadt, Germany
†Computer Security and Privacy, University of Göttingen, Germany
‡Hasso Plattner Institute, University of Potsdam, Germany
Email: huster@cs.uni-goettingen.de, mhollick@seemoo.de, jiska.classen@hpi.de

***Abstract*—The security of Linux kernel interfaces is paramount in preventing over-the-air, proximity, or other network attacks. The Linux kernel is fuzzed continuously to detect newly introduced bugs. Despite their long runtime, existing fuzzers fail to detect critical bugs, as they are unaware of physical device semantics and difficult to adapt to new devices. This paper proposes a novel fuzzer called VirtFuzz, which is based on Virtual I/O (VirtIO) device drivers. A proxy mechanism enables data collection from physical device interaction. These collected inputs are then used to fuzz through a virtual device. Using our universal VirtIO device, VirtFuzz is generic and can be easily adapted to various Linux VirtIO kernel drivers and their related subsystems. We use this approach to fuzz the Linux Bluetooth and Wireless LAN (WLAN) stacks. To demonstrate the adaptability of our approach, we additionally provide implementations to fuzz the networking and input stack. We find 31 new, manually confirmed bugs, with 6 Common Vulnerabilities and Exposuress (CVEs) assigned.**

***Index Terms*—Fuzzing, System Security, Linux, VirtIO, Bluetooth, Wireless LAN**

1. Introduction

WLAN and Bluetooth are prevalent wireless protocols used in many different device types, such as smartphones, laptops, Internet of Things (IoT) devices, and even desktop systems. Both protocols were first standardized in 1999; since then, their feature sets have evolved significantly. With the development of new features, their complexity increased. For example, in 2005, the Linux kernel Bluetooth subsystem had around 18 100 lines of code. In 2023, it comprises 80 200 lines of code, excluding device-specific drivers. This complexity increases the probability of bugs and, thus, vulnerabilities. In addition to their complexity and prevalence, potential vulnerabilities require no direct physical interaction for exploitation because both protocols are wireless.

The Linux wireless stack comprises device drivers for hardware interaction and subsystems to process these interactions and trigger new ones. These subsystems further manage state, e.g., in WLAN, participants have different roles. Since the stacks are implemented in the kernel, exploitable vulnerabilities could affect the whole system. However, previous research on Bluetooth and WLAN focused on attacks affecting only wireless standards and the confidentiality of data transmitted [1, 2, 3, 4, 5, 6]. Little research has been done on exploitable vulnerabilities leading to full system compromise, and mostly on other parts of the stack such as the wireless chip [7, 8, 9, 10, 11] or different operating systems [12].

Fuzzing is a well-established security testing method to uncover vulnerabilities and harden the kernel [13, 14, 15, 16, 17, 18]. Nevertheless, at the time of writing, popular Linux kernel fuzzers do not cover the full Bluetooth or WLAN stack or need significant adaptations to work with them. Therefore, we design and implement a generic Linux fuzzer called VirtFuzz, based on VirtIO [19, 20], a popular standard for para-virtualized devices. Each device is implemented separately, as it behaves differently, but the message interface between the hypervisor and the Virtual Machine (VM) is standardized. We use this standardized message-passing mechanism for fuzzing the Linux kernel. Many device types already have a VirtIO driver, enabling targeting various stacks, including Bluetooth and WLAN. We create a novel universal VirtIO device specifically for fuzzing, facilitating the integration of new device drivers. While this approach is highly adaptable, it reaches different states that are extraordinarily relevant to driver and subsystem security. Integrating new devices is more accessible, as it does not require complex grammar definitions or virtual device setup routines compared to other state-of-the-art fuzzers. VirtFuzz found multiple novel critical issues in processing WLAN beacon frames, exposing a Remote Code Execution (RCE) attack surface without user interaction. These issues were not discovered by other public Linux kernel fuzzers, despite being introduced to the kernel code base more than three years ago and although some kernel fuzzers are running continuously [21].

Furthermore, we implement and evaluate a new method to collect genuine initial input seeds. Initial seeds directly impact fuzzing success [22]. We connect the virtual device directly with the corresponding physical hardware and record the exchanged messages. By interacting with the physical hardware from within the VM, we assemble a set

of authentic inputs for fuzzing.

In summary, our main contributions are as follows:

- We design and implement VIRTFUZZ, a generic VirtIO-based Linux kernel fuzzer, and adapt it for the WLAN, Bluetooth, networking, and input stacks.
- We design, implement, and run a proxy collecting physical WLAN and Bluetooth device interaction.
- We find and responsibly disclose 31 bugs in the Linux kernel's WLAN and Bluetooth stack, with 6 severe vulnerabilities that got CVEs assigned. Since Android partially uses the Linux WLAN stack, it is affected by 3 of these CVEs.

We will publish the VIRTFUZZ source code upon acceptance of this paper.

2. Background

For readers unfamiliar with fuzzing and VirtIO, we introduce the essential concepts in the following.

Fuzzing. Fuzzing enables automated vulnerability discovery. A fuzzer generates inputs for a target, trying to reach bugs in the code causing an error condition. In coverage-guided fuzzing, the fuzzer adapts randomized inputs based on newly reached code paths. Grammar-based fuzzing generates inputs based on prior knowledge about the input format. Combining coverage guidance and grammar descriptions enables even more efficient fuzzing. Manes et al. provide an overview of different fuzzing approaches [23].

Kernel Coverage Collection. Coverage-guided fuzzers require knowledge of which code was executed. Coverage information guides the fuzzer, e.g., to decide whether an input covers new code paths. Since the fuzzer targets a compiled binary, the execution environment or the binary must support coverage collection. Modern compilers can instrument code for fuzzing. Therefore, specific code is added, e.g., into every basic block. Besides, comparisons can be instrumented so that for each kind of comparison, certain functions are executed containing both operands additionally to the comparison.

kcov [24] is a Linux kernel mechanism that collects coverage information of system calls. First, a system call initializes coverage collection. After executing the targeted system calls, the coverage collection is stopped, and the covered blocks or comparisons are made available to userspace.

Sanitizer. Sanitizers enable the detection of certain error conditions by adding additional checks. While decreased performance would be impractical on end-user systems, they are valuable for finding otherwise hard-to-detect bugs while fuzzing. Different sanitizers are optimized for different bug classes. For fuzzing the Linux kernel, VIRTFUZZ uses the Kernel Address Sanitizer (KASAN) [25], Undefined Behavior Sanitizer (UBSAN) [26], and Kernel Memory Leak Detector (KMLD) [27]. These sanitizers add checks at compile time, e.g., for each memory access, so invalid memory access can be detected. Memory safety issues are a common problem leading to severe vulnerabilities and are detected more easily using sanitizers [28].

VirtIO. The Virtual I/O (VirtIO) standard specifies virtual devices that *"look like physical devices to the guest within the virtual machine"* [19]. VirtIO devices are paravirtualized devices, meaning the VM recognizes that the device is virtual. This knowledge is used to implement a simpler driver, increasing performance [29].

Linux kernel version 6.3-rc3 has 19 VirtIO drivers for different subsystems, including, e.g., drivers for the Bluetooth, networking, and WLAN stacks. A VirtIO device specification contains a device ID, a certain number of virtqueues for communication, feature bits, device configuration, and its behavior [19]. Hypervisors implement each device individually since each device operates differently. Data is exchanged through virtqueues, and the data-type is device specific. The Bluetooth device uses, e.g., Host Controller Interface (HCI) frames, which is also the data usually exchanged between a physical Bluetooth controller and its driver. In contrast, regular WLAN frames are embedded into a generic netlink frame specifically used by the VirtIO WLAN driver, containing additional information such as the wireless channel.

As VirtIO devices are widely used in production, a common attack scenario is a VM escape by exploiting a vulnerable device implementation. Fuzzing through VirtIO is used to increase hypervisor security [30, 14]. To the best of our knowledge, it was not yet used the other way around, for kernel fuzzing. For VIRTFUZZ, we design a universal VirtIO device, which we use to provide fuzzing inputs to several Linux kernel interfaces.

3. Threat Model

In the following, we outline Linux attack surfaces and how they generalize to other systems.

Network, Proximity, and Physical Attacks. In this paper, we focus on Linux device drivers and their subsystems. Attackers could trigger vulnerable code paths over the Internet using TCP/IP (*Network Attacks*), over-the-air with WLAN and Bluetooth (*Proximity Attacks*), or by plugging in devices into a computer's ports (*Physical Attacks*) [31]. We assume an attacker without system-level access, i.e., they do not have an unprivileged user account on the target system. In the worst case, attackers could gain RCE through a vulnerable subsystem. However, even a Denial of Service (DoS) in the Linux kernel can have a severe impact—a device might end up in a boot loop because it crashes whenever it attempts to connect to a malicious WLAN access point. We discovered several DoS and RCE vulnerabilities requiring no user interaction to exploit the Linux WLAN stack.

While the input format for the WLAN VirtIO driver is data frames as sent over the air, the Bluetooth chip preprocesses frames. It communicates them to the host using the standardized HCI format [32], also supported by the

Bluetooth VirtIO driver. Depending on where a Bluetooth frame's contents are processed, a malicious frame could target *(1)* the Bluetooth chip's firmware, *(2)* the Linux host directly, even after pre-processing, or *(3)* the Linux host under the assumption of an already attacked Bluetooth chip. Most vulnerabilities identified by VIRTFUZZ fall into (3), and were not assigned CVEs. Nonetheless, these vulnerabilities are dangerous and should be patched—wireless chip firmware is optimized for performance and low energy consumption. The firmware has few mitigations, making it an easy target for attackers and a viable entry point to escalate into the operating system [7, 9, 11, 10, 33, 34, 35].

Local Privilege Escalation Attacks. Unprivileged user-space applications could try to escalate their privileges by attacking kernel drivers and their subsystems [36]. The impact of local privilege escalation is exceptionally high on systems that enforce strict user separation or even sandboxing, such as Android, which is based on the Linux kernel [31]. In contrast to wireless attacks, the initial attack vector in this scenario is typically malware. As protection, device drivers are only accessible indirectly through restricted APIs, e.g., user-space applications can pair a Bluetooth headset through an API call but cannot send arbitrary HCI commands.

Further Attacks. The vulnerabilities identified by VIRTFUZZ might also exist in other projects. Programmers tend to make similar mistakes when implementing specifications [2, 4], might consult the Linux source code for reference, or even unknowingly copy code when using GitHub's Copilot [37]. In fact, we found that a payload identified by VIRTFUZZ also crashes an embedded WLAN chip, confirming the existence of further threats in unknown targets.

4. VIRTFUZZ Design and Implementation

In this section, we formulate design goals for VIRTFUZZ and implement them accordingly. To this end, we introduce multiple required components.

4.1. Design Goals

Current Linux kernel fuzzers do not reach specific subsystem components processing untrusted inputs from wireless interfaces. They would require extensive adaptation for each subsystem to reach those components. Furthermore, they use arbitrary inputs for fuzzing or generate valid inputs from a pre-defined grammar.

We aim at fuzzing different components flexibly and without requiring significant fuzzer and target modifications. Besides, we want to implement a mechanism to easily collect seeds containing valid real-world inputs. Therefore, we propose a new fuzzer design to overcome these issues with the following design goals:

D1 Depth: The fuzzer should cover previously uncovered deep parts of the kernel.

D2 Extensibility: The fuzzer should be easily extendable to different interfaces.

D3 Authenticity: It should be easy to collect real-world inputs to be used as genuine seeds.

While fulfilling these design goals, the fuzzer should uncover real-world security risks; thus, its architecture should stay close to real-world scenarios.

4.2. Universal VirtIO Device

We create a novel *universal* VirtIO device designed explicitly for fuzzing, as shown in Figure 1. Using VirtIO as the message-passing mechanism fulfills two of our primary design goals: VirtIO drivers are implemented for a wide area of subsystems, thus fuzzing through a universal VirtIO device can cover a large amount of different kernel code deeply (**D1**) with only minor adaptations (**D2**).

This universal device does not need to implement device-specific behavior. Adaptation to serve inputs to newly specified devices—and thus kernel subsystems—is as trivial as changing some command line parameters. As the device behaves similarly to a physical device, vulnerabilities found are portable to real-world scenarios, as inputs processed by the VirtIO device take a path through the kernel similar to their physical equivalents.

For our universal VirtIO device, we extend the open-source hypervisor QEMU version 7.1 [38]. As described, a particular VirtIO device has a device ID, a number of virtqueues, a list of feature bits, and a device configuration [19]. VirtIO uses this device ID to detect device types. The virtqueues are the communication mechanism to exchange buffers with the virtual machine and vice-versa. Per the specification, a particular virtqueue can either

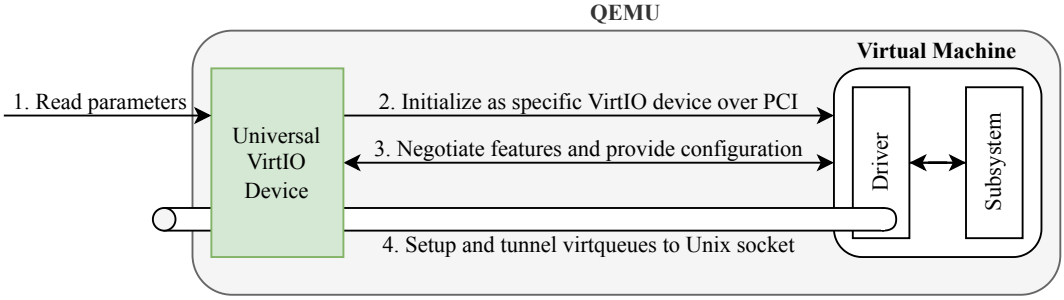


Figure 1: The setup procedure of our universal VirtIO device.

send or transmit data. Feature bits indicate the support for a particular driver behavior or the device. The device and driver exchange these on initialization and negotiate a common subset. The specification defines the correct device operation, depending on the negotiated features. For example, the Bluetooth device has feature bits indicating the support for vendor-specific HCI extensions. Finally, the device configuration is a C-structure containing information about the device, e.g., for Bluetooth, the vendor-specific command code for the Microsoft HCI extension.

For most devices, these parameters are constant per device and do not change during runtime. Thus, our universal VirtIO device accepts them as arguments and expects the configuration as a binary file. Furthermore, the index of the reception and, if applicable, the transmission queue must be indicated per device. All of this is usually implemented per device, as each device is defined to behave differently. However, we do not need emulated behavior for fuzzing but can directly forward the inputs and outputs.

With all these parameters set, our device registers itself as a VirtIO device over PCI, negotiates the features with the driver, and provides its static configuration. Afterward, it sets up all virtqueues and connects to a provided socket path. Then, it forwards the buffers received from the configured transmission queue to the socket and the received buffers from the socket to the reception queue.

Our final implementation includes the lightweight patches and device definitions for the following subsystems:

- Bluetooth with the `virtio_bt` driver,
- WLAN with the `mac80211_hwsim` driver,
- networking with the `virtio_net` driver, and
- input with the `virtio_input` driver.

4.3. Proxy

Related work has shown that fuzzing results vary significantly, depending on the initial seed set [22]. To obtain authentic seeds for fuzzing (**D3**), we propose proxy tunneling and recording buffers of the physically equivalent interface of the targeted one to the VM. Before a fuzzing run, the security researcher using VirtFuzz is thus able to use the interface in the virtual machine with physical devices. Hence, the researcher can, e.g., interact with different real-world devices from within the VM: By pairing different Bluetooth devices, exchanging data through one of the many protocols, or running a WLAN access point within the guest and joining it from a physical device, valid and genuine samples are collected. These recorded buffers are used as seeds for the fuzzing run. We later show that increased code coverage can be achieved faster using such a proxy's inputs. Such a proxy is an easy-to-use method to collect initial seeds for a successful fuzzing run more quickly, as the security researcher can use the interface inside the VM in the usual way. Figure 2 depicts this process.

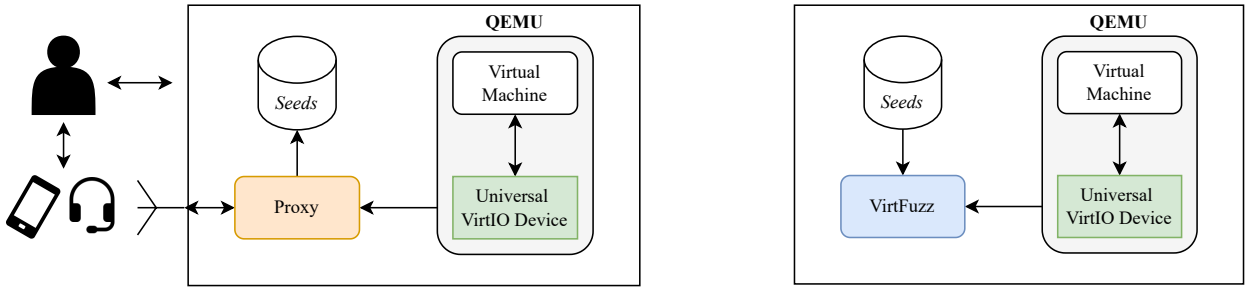
We implement the VirtFuzz proxy in Rust. It provides access to physical hardware for some VirtIO drivers by using our Universal VirtIO device. In VirtIO terminology, it is a backend running in a separate process. The proxy currently supports Bluetooth and a WLAN card in monitor mode, which must be configured to the appropriate channel. QEMU does not implement VirtIO WLAN and Bluetooth devices. Further devices can be added to the proxy, e.g., for fuzzing an Ethernet interface.

The buffers sent to the VM are saved to be used as seeds for later fuzzing campaigns. We verify the functionality from within the VM and can interact with physical devices outside the VM. We use it to collect genuine input seeds for WLAN and Bluetooth fuzzing.

4.4. Linux Kernel Adaptations

The Linux kernel includes an interface to collect code coverage information for fuzzing named *kcov* [24]. As kcov is only accessible from userspace, we modify the Linux kernel so that it writes the coverage information to QEMU's shared memory device. This device maps memory from the host to the guest. Thus, the coverage information written by the VM is directly accessible by VirtFuzz. The shared memory device was similarly used by Peng and Payer as a communication device for their USB fuzzer [16].

kcov was initially built to track the code covered by the execution of system calls. Thus, system calls must be made to start and finish the coverage collection. As we provide our inputs through the VirtIO device from outside the



(a) Prior to fuzzing, genuine packets from interaction with real-world devices are collected and stored.

(b) Afterward, the fuzzer is started with an initial corpus containing these genuine packets.

Figure 2: The complete VirtFuzz fuzzing process

VM, we introduce new functions controlling the coverage collection that must wrap the target's entry point. These enable and disable coverage collection and write delimiters to the shared memory. Listing 1 shows the required changes for the WLAN reception entry point. The entry point is very similar across different kernel versions, and the initial patch containing the annotations does not need changes. The first highlighted method call enables coverage recording and writes a constant value, indicating that the frame processing starts. On the reception of a WLAN frame, the covered addresses are appended to the shared memory. The last highlighted method call stops this coverage recording and writes a constant value, indicating that the frame processing is terminated.

kcov also supports collecting comparisons instead of covered addresses. By tracking comparisons, VirtFuzz can implement mutations to overcome failed comparisons without symbolic execution [39]. As a system call usually configures this, we add a kernel parameter indicating whether the covered addresses or comparisons should be recorded. VirtFuzz automatically chooses from collecting comparisons or coverage, depending on its mode.

To enable testing different kernel versions, we minimize the required changes in the kernel code. We provide a script that applies the correct set of patches depending on the version and thus support fuzzing Linux kernel version 5.13 up to the current version, 6.0. We will publish the patch sets and the script upon acceptance of this paper.

```
static void ieee80211_tasklet_handler(struct
    tasklet_struct *t) {
  struct ieee80211_local *local = from_tasklet(local, t,
      tasklet);
  struct sk_buff *skb;
  kcov_ivshmem_start(); // start coverage collection
  while ((skb = skb_dequeue(&local->skb_queue)) ||
   (skb = skb_dequeue(&local->skb_queue_unreliable))) {
    switch (skb->pkt_type) {
    case IEEE80211_RX_MSG:
      /* Clear skb->pkt_type in order to not confuse
          kernel netstack. */
      skb->pkt_type = 0;
      ieee80211_rx(&local->hw, skb);
      break;
    case IEEE80211_TX_STATUS_MSG:
      skb->pkt_type = 0;
      ieee80211_tx_status(&local->hw, skb);
      break;
    default:
      WARN(1, "mac80211: Packet is of unknown type %d\n",
        skb->pkt_type);
      dev_kfree_skb(skb);
      break;
    }
  }
  kcov_ivshmem_stop(); // end coverage collection
}
```

Listing 1: Annotations enabling VirtFuzz coverage collection in Linux WLAN frame reception.

4.5. Resulting VirtFuzz Architecture

We summarize the VirtFuzz architecture and provide further implementation details in the following. The fuzzer's core runs as a process on the host machine. It starts a VM in our modified QEMU and connects to the universal VirtIO device, the character device that emits the kernel log and opens the shared memory. After booting, it sends a single input buffer, watches the kernel coverage on the shared memory, and evaluates it. If the input covers previously uncovered areas, it is saved for later. Otherwise, it is discarded.

Our main fuzzing components are based on LibAFL. LibAFL is a framework written in Rust that provides building blocks to assemble fuzzers, such as mutators, input schedulers, and observers for different instrumentation methods [40]. LibAFL does not support QEMU's full system emulation at the time of writing. Accordingly, we extend LibAFL by several components to

1) execute inputs in a VM through our universal VirtIO device,
2) convert the kernel's kcov coverage to a coverage map,
3) evaluate the kernel's kcov comparisons records, and
4) detect crashes from reading the kernel messages and deduplicate these.

4.5.1. Executor. The LibAFL framework names the component running a target on an input the *Executor*.

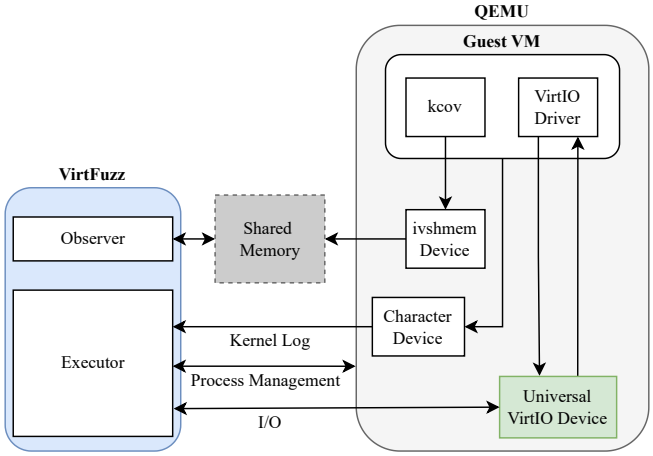


Figure 3: Overall fuzzer architecture.

```
{
  "virtio_id": 40,
  "virtqueue_num": 2,
  "virtqueue_tx": 0,
  "virtqueue_rx": 1,
  "features": [0, 1, 2],
  "config": "000000FF"
}
```

Listing 2: VirtIO example configuration for the Bluetooth device.

Our executor starts the modified QEMU with our universal VirtIO device attached. Therefore, it needs a device definition for the different targets. Most importantly, the device configuration requires the VirtIO ID and the indices of the virtqueues to send and receive data, as described above. As no such standardized VirtIO device configuration exists, we use a JSON object to carry that information. The device definition for, e.g., Listing 2 shows the Bluetooth VirtIO device; it includes optional features and a hexadecimal device configuration. Such a definition and entry point annotation in the Linux kernel are the only requirements to make a subsystem fuzzable by our approach (**D2**). This setup is extremely lightweight and straightforward compared to other fuzzers, as shown in Section 5.6.

When the QEMU process runs, the executor sends single buffers to the universal VirtIO device. It watches the shared memory where the kernel writes the recorded coverage for the delimiter indicating termination. After termination or timeout, the kernel messages are parsed to decide whether a crash occurred.

4.5.2. Coverage. The LIBAFL framework defines an observer as *"an entity that provides information from a single execution of the target"* [40]. We implement two different observers to be used with our modified kcov: One converts the recorded addresses to an edge coverage map, and the other parses the recorded comparisons.

Edge Coverage Map. Our coverage map observer parses the shared memory and reads all addresses covered by the execution after its termination. These are converted to an edge coverage map following the classical AFL algorithm [41, Sec. 1], which looks as follows.

```
loc = hash(prev_ip ^ ip) % self.len();
map[loc] = (map[loc] + 1) % 255;
prev_ip = ip >> 1;
```

The list of covered addresses can be written into a file for later analysis. Thus, a report can be created showing the covered code paths.

Comparisons Tracking. We furthermore implement an observer for kcov's comparison collections mode. Here, the kernel records comparisons made while processing a single input. After executing a single input buffer, it parses the shared memory containing the comparisons in kcov format and translates them into the standard format used by LIBAFL's *cmplog* technique.

4.5.3. Crash Deduplication. In LIBAFL, each execution is classified as interesting or not. An input triggering an interesting execution is saved into the corpus. Otherwise, it is discarded. Therefore, several *Feedbacks* exist, e.g., a *MapFeedback* maximizing the edge coverage.

Since different inputs might trigger the same crash, we implement a mechanism that processes the crash message log to a unique crash identifier. Here, the instruction pointer and error message are combined. This identifier is then used for crash deduplication: If a crash with the same identifier has already been found, it can be discarded. For certain bugs, it might be useful to obtain the same crash triggered by slightly different inputs. Thus, this deduplication of crashes can be disabled.

4.5.4. Input Scheduling. Bluetooth and WLAN are particularly stateful. Sequences of frames influence this state. To increase statefulness, we do not reset the VM after each input but continue fuzzing. As a byproduct, this leads to performance gains. However, it makes coverage metrics unstable and less reliable, e.g., a frame that uses a specific connection covers different code when sent after a frame that opens such a connection than before the processing of such a frame. Thus, we cannot rely on this data to use it for improved input scheduling mechanisms suggested by recent research [42]. Instead, we use simple randomized scheduling, which increases the probability of scheduling dependent frames after another.

4.5.5. Mutations. The inputs are binary frames. During fuzzing, they are mutated repeatedly by a set of mutators. We apply the LIBAFL *havoc* mutations, such as bit flips, byte insertions, and deletions. These mutations are expected to increase edge coverage and are widely used [43].

Instances using comparison tracking additionally use *Input-to-State* mutations inspired by REDQUEEN [39] and WEIZZ [44]. Comparison tracking records which operands are compared during an execution. Afterward, the mutator searches the frame for recorded operands. Finally, they are replaced by the counterpart operand. This enables VIRTFUZZ to pass roadblocks such as magic bytes.

To apply the mutations to a frame, we use the optimized mutation scheduling algorithm *MOPT* [45]. Here, the efficiency of the different mutations is measured, and their scheduling is adapted accordingly.

4.6. Guest Image

VIRTFUZZ targets a VM running a Linux kernel. Therefore, the VM requires an image with an Operating System (OS). We adapt SYZKALLER's image generation script to generate a suitable image, which creates a minimal Debian installation [15]. This later enables the usage of the same guest image to evaluate our fuzzer compared to SYZKALLER.

Especially for WLAN, states triggered from userspace enable different kernel code paths. For example, some code is only reachable when acting as an access point, scanning for networks, or being connected to a network. After the discovery of 27 bugs in the Linux Bluetooth and WLAN stack, we implement a method to trigger different WLAN states. Other kernel fuzzers, such as SYZKALLER, trigger these states via system calls. VIRTFUZZ cannot do so, as our architecture does not use a userspace agent. Thus, we add several simple `systemd` services to the guest image that enable these states. In addition to that, we also use the default service for acting as an access point shipped

with the Debian package `hostapd`. We show the service enabling permanent Wi-Fi scanning in Listing 3. As `systemd` services can be enabled via the kernel command line, VIRTFUZZ enables the appropriate service, depending on which state should be fuzzed.

```
[Unit]
Description=Permanently scan for Wi-Fi

[Service]
ExecStart=/bin/bash -c "ip l set wlan0 up && while true;
    do iw wlan0 scan; done;"
Restart=always

[Install]
WantedBy=multi-user.target
```

Listing 3: One of the two custom `systemd` services that enable permanent scanning for Wi-Fi networks.

5. Evaluation

In this evaluation, we answer the following research questions:

RQ1 How does VIRTFUZZ compare to existing state-of-the-art fuzzers?
RQ2 Does VIRTFUZZ perform well in new bug discovery?
RQ3 Do recorded proxy seeds improve bug finding?
RQ4 How does VIRTFUZZ compare concerning device flexibility and adaptation?

As a baseline fuzzer for comparison, we use SYZKALLER [21]. It is a popular Linux kernel fuzzer, fuzzing the kernel, including the Bluetooth and WLAN stack.

5.1. General Fuzzing Setup

We ran VIRTFUZZ on different setups with varying durations during the development and evaluation, which we illustrate in the following. We found the vulnerabilities during the whole process of VIRTFUZZ's development, with most of the Bluetooth vulnerabilities identified on a Thinkpad T495. The majority of the WLAN vulnerabilities were discovered on the machine used for the evaluation, introduced in Section 5.2.

For the WLAN and Bluetooth seed collection, we used an Intel Wireless-AC 9260 card of a Thinkpad T495. Bluetooth seeds were collected during interaction with the following devices: an Android smartphone, mouse and keyboard, and headphones. We scanned for nearby networks to collect the WLAN seeds and attempted to connect to them. Furthermore, we ran the VM as Access Point (AP), and attempted to connect from the outside. VIRTFUZZ supports fuzzing the WLAN subsystem in several modes, such as ad-hoc, station, and infrastructure (AP).

5.2. Advanced Evaluation Setup

We evaluate VIRTFUZZ on a VM with 12 cores of an AMD EPYC 7302 16-core processor with 64 GB RAM. For storage, an SSD is used. We target WLAN on Linux 5.19 and Bluetooth on Linux 5.16. For this, we run

- SYZKALLER,
- VIRTFUZZ with pre-recorded seeds, and
- VIRTFUZZ without pre-recorded seeds

on the two targets for 24 h each.We repeat these experiments three times.

We track executions, bugs discovered, and raw coverage as covered addresses. SYZKALLER records the coverage of the whole kernel; our approach only measures the coverage of the related subsystem. Therefore, we restrict the recorded coverage to the related subsystems, which means `net/wireless`, `net/mac80211`, and `drivers/net/wireless/mac80211_hwsim` for WLAN and `net/bluetooth` for Bluetooth. Furthermore, SYZKALLER also records the coverage of setting up the virtual devices and bringing them into a particular state: e.g., to fuzz WLAN, SYZKALLER sets up an IBSS network between two virtual devices. For Bluetooth fuzzing, SYZKALLER sets up fake Bluetooth connections and responds to HCI commands. Thus, coverage is hard to compare. To counter this issue, we automatically convert the SYZKALLER corpus into binary frames that are replayed with VIRTFUZZ. At the end of SYZKALLER's run, we convert the whole corpus and replay it with VIRTFUZZ. We implement services that mimic SYZKALLER's setups for WLAN and Bluetooth each. Hence, the state of the subsystems is the same compared to running SYZKALLER directly. Afterward, we filter SYZKALLER's recorded coverage to contain only functions covered by replayed frames. Furthermore, for the evaluation, we implement a similar mechanism of fake responses to HCI commands for some instances of VIRTFUZZ. By this, we obtain a fair coverage comparison over time.

We sanitize the targets with KASAN only [25], in contrast to previously using multiple sanitizers for VIRTFUZZ, and use the same VM image for SYZKALLER as well as for VIRTFUZZ. This image is based on Debian Stretch and an adapted version of SYZKALLER's standard image. We described the used image for VIRTFUZZ in Section 4.6. As it is similar to SYZKALLER's standard image, we can use it for our runs and the ones of SYZKALLER.

In addition to these controlled experiments, SYZKALLER has run for several years on different Linux kernel versions and automatically reports bugs [21]. While we ran VIRTFUZZ in different scenarios during design and implementation, its total runtime is significantly below SYZKALLER's. For our bug evaluation in Section 5.3, we continued running VIRTFUZZ beyond this 24 h runs. Thus, it includes bugs found outside the previous 24 h experiments.

5.3. Bug Discovery & Relevance

Upon submission of this paper, VIRTFUZZ discovered 31 individual bugs concerning the Linux Bluetooth and

WLAN stack. Multiple sanitizers revealed different bug types: KASAN [25], UBSAN [26], and KMLD [27]. The identified vulnerability types are displayed in Table 1. They consist of various memory safety issues, such as over- and underflows, illegal reads and writes, and null pointer dereferences, found mainly by KASAN.

Table 2 shows more details for each bug, including the affected function and the version introducing the bug. For 12 findings, we received six CVEs with a maximum CVSS score by the NVD of 8.8 [46], indicating high severity. The earliest affected version is Linux 2.6.27, published in 2008, and most bugs have been in the Linux kernel for several years.

SYZKALLER fuzzes the Linux kernel continuously [21] but only found one of the Bluetooth bugs in parallel [47]. It also supports the injection of WLAN frames since 2020 [48]. This shows that it targeted similar parts of the kernel code. However, we discovered severe vulnerabilities in Linux WLAN, which have been in the kernel since 2019, and Bluetooth vulnerabilities that were introduced in Linux 2.6.27, published in 2008.

Thus, we conclude that VIRTFUZZ successfully discovered new vulnerabilities and helped improve the Linux kernel, answering **RQ2** and partially addressing **RQ1**. To demonstrate our findings' relevance and properties, we look deeper into two significant vulnerabilities in the following.

5.3.1. WLAN: Heap Overflow on Malicious Beacon Frame. We discovered a heap overflow in the Linux WLAN stack (CVE-2022-41674). An invalid beacon frame triggers the heap overflow. Beacon frames are used to, e.g., advertise an access point. They are processed when scanning for nearby networks. On Android, this happens automatically in the background for location determination. While Android uses a different Bluetooth stack based on Rust to improve security [49], Android's WLAN stack is mostly the same as Linux and is also affected by this vulnerability.

802.11ax improves scanning efficiency by introducing a multiple BSSID element. This element announces multiple networks from the same access point in one frame [50]. The Linux WLAN stack contains an integer overflow on processing such an MBSSID beacon frame. The overflow allows an attacker to write up to 256 arbitrary bytes to the kernel heap. We display the vulnerable code in Listing 4. The length of the MBSSID element is calculated in the line highlighted in red. As both variables are just one byte long, this calculation can overflow. The following `memcpy` then copies additional 256 bytes due to the implementation of its size calculation. A mitigation is to use a different type for `cpy_len`. Exploiting this vulnerability leads to over-the-air DoS attacks without user interaction required. An advanced attacker might be able to exploit this for RCE.

We responsibly disclosed this issue in private to SUSE as a Linux vendor. After informing the kernel security team and the other vendors, patches for this and several other findings were released.

5.3.2. Bluetooth: Various Vulnerabilities on Multiple Connection Complete Events. We uncovered various vulnerabilities in the Linux Bluetooth stack on multiple connection complete events. These events are defined for different connection types of Bluetooth and, in general, occur on establishing a new connection [32]. To trigger the vulnerabilities, a combination of four frames is required: A first frame containing a connect request, two duplicate connect complete events, and a fourth frame using that connection, for example, a disconnect event.

In Section 4.5.4, we illustrated that VIRTFUZZ does not perform a reset after an input but continues so that the output on a crash can contain thousands of frames. We build a tool to replay these payloads and minimize them to identify the frame combination triggering a vulnerability, which found the combination of the four frames.

On receiving the first connection complete event, the related kernel object is created. Receiving the second connection complete event for the same connection triggers a cleanup routine, as a kernel object with the same identifier already exists, which frees the assigned memory. The last frame needs to use that connection. Depending on the frame type, this leads to null pointer dereferences, Use after Frees (UaFs), or slab out of bounds in several kernel list calls.

All the functions for different Bluetooth connection types share this vulnerability. Our patch fixing this vulnerability was merged into the kernel tree[1].

1. Commit `d5ebaa7c5f6f` ("Bluetooth: hci_event: Ignore multiple conn complete events")

TABLE 1: Vulnerability Types Found by VIRTFUZZ

Target	Type	#
Bluetooth	Memory leak	1
	Null pointer dereference	7
	Slab out of bounds	2
	Use-after-free write	3
	Use-after-free read	1
	Wild memory access	2
WLAN	Heap overflow	1
	Infinite loop	1
	Integer underflow	1
	Null pointer dereference	2
	Slab out of bounds	4
	User memory access	1
	Use-after-free read	5

```
static void cfg80211_update_notlisted_nontrans(struct
     wiphy *wiphy, struct cfg80211_bss *nontrans_bss,
     struct ieee80211_mgmt *mgmt, size_t len) {
      u8 *ie, *new_ie, *pos;
  const u8 *trans_ssid, *mbssid;
  u8 cpy_len;
  [...]

  /* copy the IEs after MBSSID */
  cpy_len = mbssid[1] + 2;
  memcpy(pos, mbssid + cpy_len,
         ((ie + ielen) - (mbssid + cpy_len)));
```

Listing 4: Excerpts of the vulnerable code leading to a heap overflow in the Linux kernel's WLAN stack.

TABLE 2: Vulnerabilities in the Bluetooth and WLAN Stack of the Linux Kernel Found by VIRTFUZZ

#	Type	Subsystem	Function	Patched	Introduced in Version	CVE
1	Memory leak	Bluetooth Driver	virtbt_rx_handle	✓	Linux 5.13	CVE-2022-26878
2	Wild memory access	Bluetooth	hci_inquiry_result_with_rssi_evt	✓	Never	
3	Slab out of bounds	Bluetooth	hci_le_meta_evt	✓	Linux 3.2	
4	Null pointer dereference	Bluetooth	hci_sync_conn_complete_evt	✓	Linux 3.12	
5	Null pointer dereference	Bluetooth	msft_vendor_evt	✓	Never	
6	Use-after-free write	Bluetooth	klist_add_tail (hci_conn_add_sysfs)	✓	Linux 2.6.27	
7	Use-after-free write	Bluetooth	klist_release (hci_conn_del_sysfs)	✓	Linux 2.6.27	
8	Null pointer dereference	Bluetooth	klist_next (hci_conn_del_sysfs)	✓	Linux 2.6.27	
9	Null pointer dereference	Bluetooth	kill_device	✓	Linux 2.6.27	
10	Slab out of bounds	Bluetooth	kfree_skb_reason	✓	Linux 2.6.27	
11	Null pointer dereference	Bluetooth	aosp_do_open	✓	Linux 5.17	
12	Null pointer dereference	Bluetooth	msft_do_open	✓	Linux 5.17	
13	Use-after-free read	Bluetooth	hci_send_acl	✓	Linux 3.8	
14	Null pointer dereference	Bluetooth	klist_next (hci_conn_del_sysfs)	✓	Never	
15	Use-after-free write	Bluetooth	klist_add_tail (hci_conn_add_sysfs)	✓	Never	
16*	Wild memory access	Bluetooth Driver	rfcomm_run (virtbt_rx_work)	✓	5.13	
17	Slab out of bounds	802.11 Driver	__nla_validate_parse (hwsim_virtio_handle_cmd)	✓	Linux 5.7	
18	Null pointer dereference	802.11	cfg80211_rx_unprot_mlme_mgmt	✓	Linux 5.8	CVE-2022-42722
19	Heap overflow	802.11	cfg80211_update_notlisted_nontrans	✓	Linux 5.1	CVE-2022-41674
20	Slab out of bounds	802.11	cfg80211_gen_new_ie	✓	Linux 5.1	
21	Use-after-free read	802.11	ieee80211_update_bss_from_elems	✓	Linux 5.2	CVE-2022-42719
22	Use-after-free read	802.11	cfg80211_inform_bss_frame_data	✓	Linux 5.1	
23	Use-after-free read	802.11	cmp_bss	✓	Linux 5.1	
24	Null pointer dereference	802.11	__cfg80211_unlink_bss	✓	Linux 5.1	CVE-2022-42720
25	Use-after-free read	802.11	cfg80211_inform_single_bss_data	✓	Linux 5.1	
26	Use-after-free read	802.11	cfg80211_put_bss	✓	Linux 5.1	
27	User memory access	802.11	__cfg80211_unlink_bss	✓	Linux 5.1	
28	Infinite loop	802.11	cfg80211_add_nontrans_list	✓	Linux 5.1	CVE-2022-42721
29	Slab out of bounds	802.11	cfg80211_parse_mbssid_data	✓	Linux 5.10	
30	Integer underflow	802.11	cfg80211_parse_mbssid_data	✓	Linux 5.10	
31	Slab out of bounds	802.11	cfg80211_find_elem_match	✓	Linux 5.1	

Entries that specify *Never* in the *Introduced in Version* column were found in the development trees before being included in a release. Vulnerabilities marked with * were found during seed collection. If the vulnerability appears in an underlying kernel function, the first function on the stack of the subsystem is given in brackets. We assume that exploiting the WLAN vulnerabilities is possible over-the-air and triggering the Bluetooth vulnerabilities requires a compromised controller (see Section 3).

To our knowledge, our Bluetooth findings are not directly exploitable over-the-air, as explained in Section 3. Hence, we did not request CVEs. Combined with a firmware vulnerability, which frequently occur in Bluetooth controllers [7, 34, 33], these vulnerabilities lead at least to DoS. The UaF writes likely lead to RCE.

5.4. Performance

To evaluate the performance, we compare the basic block coverage and execution speed for WLAN and Bluetooth fuzzing in the following, addressing **RQ1**.

5.4.1. Speed Comparison. As seen in Figure 4a, our approach outperforms SYZKALLER in the execution speed by an order of magnitude in executions per second for WLAN fuzzing. When comparing the average total executions, VIRTFUZZ executed 22 times more inputs than SYZKALLER. This is caused by SYZKALLER's more complex architecture: A manager component starts a VM and runs a fuzzer inside this VM. Thus, the input generation, mutations, and executions happen inside the VM, which is slower than the host system. Furthermore, the fuzzer sets up virtual devices. The input is sent to the kernel in the form of time-consuming system calls and the device setup. In addition, the coverage collection requires multiple system calls per execution. The state and the new inputs are synchronized to the manager via Remote Procedure Calls (RPCs) between the host and the VM.

SYZKALLER performs significantly faster in fuzzing Bluetooth than WLAN, as displayed in Figure 4c. When fuzzing Bluetooth, VIRTFUZZ initially performs faster than SYZKALLER but then is slightly slower. This is due to the virtual device setup by SYZKALLER: When fuzzing Bluetooth, SYZKALLER sets up the device once and uses it for several inputs[2]. For WLAN fuzzing, SYZKALLER has to set up a socket connection to the kernel for each frame[3], decreasing the throughput. VIRTFUZZ's VirtIO-based architecture never requires such a setup routine. Thus, the architectural advantage of our solution is less relevant concerning execution speed for Bluetooth fuzzing.

Furthermore, both fuzzers slow down over time: By

2. See the method for sending a single Bluetooth frame at `executor/common_linux.h:2847` in [15, Commit 5bc3be51cc65]

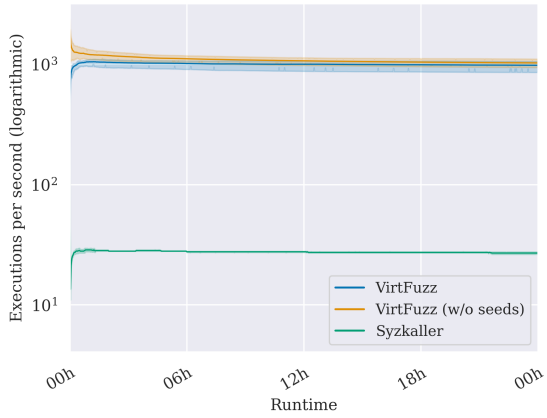
3. See the method for sending a single WLAN frame at `executor/common_linux.h:5380` in [15, Commit 5bc3be51cc65]

investigating our fuzzing runs we conclude that VIRTFUZZ focuses on inputs with variable-sized lists containing, e.g., discovered devices. Thus, more processing steps are undertaken per frame after several mutations introducing more list items, which take more time. We suspect that SYZKALLER has a similar issue. Similar inputs led to the discovery of severe WLAN vulnerabilities that we have found; thus, we do not consider this a limitation, despite the decrease in performance. We discuss the mutation strategy in further detail in the following Section 5.4.2.

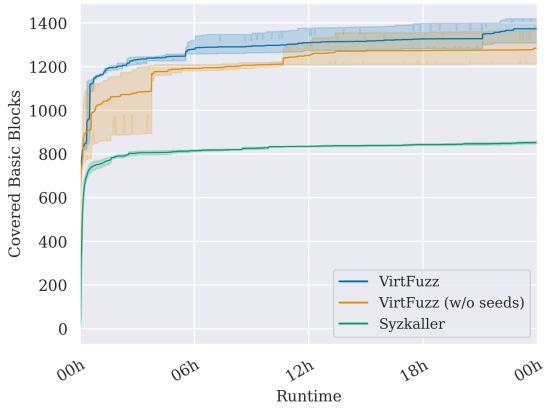
Secondly, SYZKALLER can send multiple frames to the stack in one execution, unlike our approach, which counts a single frame as one execution. A fair speed comparison is difficult. Thus, we analyze the WLAN corpora after SYZKALLER's runs to give an estimate. They contain 1615 inputs with a total number of 2197 frames. SYZKALLER uses corpus minimization. Thus we cannot calculate the mean number of frames of the whole run from the final corpus. Nonetheless, this factor of 1.36 frames per input generally approximate the frame count per input. This, or even a marginally higher factor, does not explain the massive difference in execution speed regarding WLAN fuzzing. Furthermore, both instances use the same sanitizer on all instances during the evaluation runs. We conclude that the speed difference is due to the differences in architecture, especially concerning the setup of the virtual device used for WLAN fuzzing.

Addressing the usage of recorded seeds (**RQ3**), it is notable that there is a difference with respect to execution speed when comparing runs with and without genuine inputs. While the instances without genuine inputs are initially an order of magnitude faster than the ones using initial inputs, they converge over time. Loading the input seeds from files takes more time than generating random bytes. Furthermore, random bytes are mostly invalid frames, dropped early during input parsing. Thus, the execution time is short initially, as many frames are already dropped before reaching the subsystems. This happens, e.g., for frames not containing a MAC address of a virtual WLAN device.

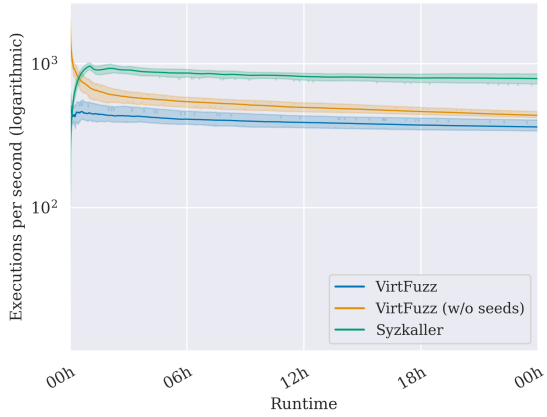
5.4.2. Coverage Comparison. A coverage comparison between SYZKALLER and VIRTFUZZ in Figure 4b shows that



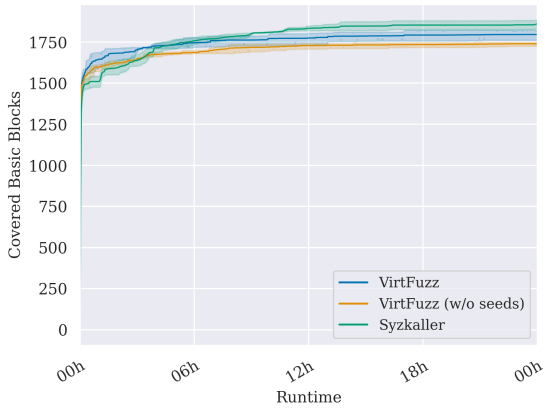
(a) Execution speed comparison for WLAN fuzzing.



(b) Basic block coverage comparison for WLAN fuzzing.



(c) Execution speed comparison for Bluetooth fuzzing.



(d) Basic block coverage comparison for Bluetooth fuzzing.

Figure 4: Performance comparisons when fuzzing the Linux WLAN and Bluetooth stacks three times for 24 h with VIRTFUZZ and SYZKALLER. The bold line shows the average value, the range depicts the minimal and maximum values.

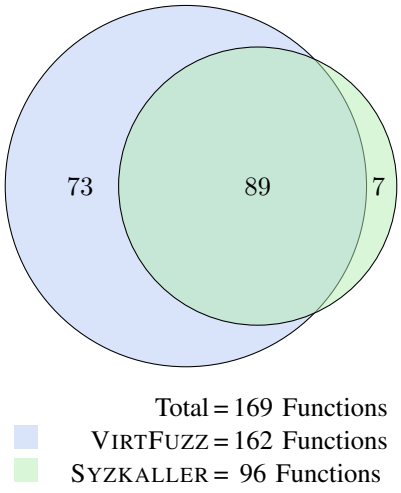


Figure 5: Venn Diagramm of functions executed by SYZKALLER and VIRTFUZZ in the Linux WLAN stack after fuzzing for 24 h.

VIRTFUZZ reaches more code regarding WLAN fuzzing. The higher initial coverage stems from inputs previously recorded by the proxy, but even without these seeds, mutations will eventually find more code paths when using our VirtIO-based approach. SYZKALLER has lower initial coverage than VIRTFUZZ, and its coverage increases more slowly over time.

When comparing the coverage of the Bluetooth evaluation, we reach a similar coverage despite having fewer executions, as displayed in Figure 4d.

We use basic block coverage as a comparison metric, the coverage format SYZKALLER currently uses. Evaluating fuzzers by basic block coverage is very common, and typically increased coverage leads to finding more bugs [51]. However, VIRTFUZZ internally uses edge coverage to decide whether a new input improves coverage. Here, the transitions between the basic blocks, as well as their count, are considered. We only use basic blocks for comparison, showing that VIRTFUZZ generally reaches more code.

VIRTFUZZ reaches a much higher function coverage than SYZKALLER when fuzzing the Linux WLAN stack, as shown in Figure 5. SYZKALLER covers 7 functions not reached by VIRTFUZZ. In contrast, VIRTFUZZ covers 73 functions not covered by SYZKALLER. 89 functions are covered by VIRTFUZZ and SYZKALLER. The bugs we discovered do not only reside in the newly covered code. By comparing the functions covered, we can assert that **SYZKALLER also reaches some of the vulnerable WLAN beacon frame parsing functions—without discovering the vulnerabilities.** The new WLAN vulnerabilities occur when parsing multiple BSSID beacon frames. These frames consist of a variable number of elements. The same basic blocks are reached independently of the total number of elements. Edge coverage differs since these basic blocks are executed repeatedly in different orders. We assume the covered basic blocks are similar, but not the transitions and counts. KASAN detects these bugs. Thus, this is not a side effect caused by different sanitizers. Our findings question if fuzzers should be mainly optimized toward basic block coverage.

We conclude that VIRTFUZZ is superior concerning execution speed for WLAN fuzzing. Furthermore, it performs better regarding basic block and function coverage. We cannot compare edge coverage, as SYZKALLER does not record it, but as VIRTFUZZ covers more basic blocks, it is plausible that it has a higher edge coverage. Regarding Bluetooth fuzzing, VIRTFUZZ performs on a similar level compared with SYZKALLER. Combined with the previous section, this performance evaluation answers **RQ1**.

5.5. Impact of Genuine Inputs

Our approach collects genuine inputs from real-world devices before fuzzing to have a good set of input seeds. These initial inputs are evaluated at the start of a fuzzing campaign and added to the corpus depending on their coverage. This section evaluates the impact of using genuine input seeds to answer **RQ3**. Figure 4 shows that using random initial inputs instead of pre-recorded ones initially increases the execution speed but converges later. We explained the reasons for this above in Section 5.4.1.

Regarding coverage, Figure 4 shows that more basic blocks are covered by using genuine seeds. This is reasonable, as the instance using genuine inputs reaches more code when VIRTFUZZ starts with valid WLAN or Bluetooth frames. Later, the instances using only random initial inputs reach a similar coverage but stay slightly below those using genuine initial inputs. Thus, using genuine seeds gives VIRTFUZZ a head start of several hours. Using the proxy is simple; thus, collecting genuine seeds requires significantly less time compared to the execution time it saves.

To dismiss the possibility of drawbacks of using recorded genuine seeds, we compare the functions covered by both configurations. We discover that the instance using random seeds does not cover additional functions. Hence, VIRTFUZZ gains a head start of several hours by using genuine seeds without any disadvantages regarding coverage and reaches deeper functionality faster.

5.6. Required Adaptations

To answer **RQ4**, we compare VIRTFUZZ with NYX [14] and SYZKALLER [15] concerning fuzzing new devices and versions. Therefore, we first introduce the required adaptations for these two solutions. Afterward, we demonstrate VIRTFUZZ's flexibility regarding new devices.

5.6.1. Comparison to Other Fuzzing Frameworks. NYX's inputs are generated from a manually defined specification for generic fuzzing. These specifications describe bytecode, which is generated by the fuzzer as input. NYX uses custom hypercalls implemented in a modified version of QEMU. These hypercalls indicate the start and the end of a single execution. Furthermore, they are required to

inject input into the target. Therefore, a hypercall is first called with the memory address where the fuzzing input should be written. Another hypercall triggers the fuzzer to write the input to the according memory location. Thus, to add a new device or subsystem to NYX, the entry and all exit points require manual annotation. The hypercalls and annotations depend on a complex set of kernel patches, with a patchset provided by Intel [52]. Furthermore, NYX requires a heavily patched version of QEMU. As NYX can inject inputs at arbitrary locations, it can universally fuzz any kernel component.

SYZKALLER also generates the inputs from a manually specified grammar. It does not rely on any custom kernel patches. As SYZKALLER fuzzes system calls, the targets must be reachable by those. Thus, to support WLAN and Bluetooth, SYZKALLER has modifications to open the related virtual device. Moreover, the grammar needs a precise definition for packet types, resulting in hundreds of lines of code only to describe WLAN and Bluetooth packets. For example, the Bluetooth frame grammar comprises 1880 lines as of writing this paper. Currently, there is work on using static analysis to help automate grammar creation [15, Issue #590].

VIRTFUZZ requires a QEMU patch introducing a VirtIO universal device, Linux kernel annotations, and a device definition.

We first look at the required adaptations for new versions of the Linux kernel. Compared to NYX, VIRTFUZZ requires only a tiny set of kernel patches, which can be easily ported to newer versions. Analyzing the required kernel changes for the initial release of NYX, they add 869 lines of code to 9 existing source code files [53]. In contrast, our patches only touch one existing source code file introducing 192 lines of code. Both numbers do not include the newly created files, as usually only the modified files make adapting to different versions hard. We publish the required kernel patches for Linux kernel versions 5.13 up to 6.0, bundled with a script automatically applying the set of patches for each version. Our modifications are mainly a new device driver for shared memory and a change to kcov to write to that memory. Most changes in existing code concern kcov and introduce seven new functions to start and stop remote coverage collection. As most of this does not modify existing functions, it is easy to port to newer versions. Furthermore, we require entry point annotation for the specific device. For example, the Bluetooth annotations consist of five function calls that must be inserted around the entry point. SYZKALLER does not need any modifications to work with a different Linux kernel version, which makes it easy to use with different kernel versions. However, in comparison to NYX, VIRTFUZZ requires clearly less deep-rooted adaptations to the kernel, which still makes it portable between different kernel versions.

Concerning device flexibility, VIRTFUZZ relies on a simple device definition and entry point annotations. The number of required annotations per device is slightly less than the annotations NYX requires per device: in addition to annotations surrounding the function entry point, Nyx also needs annotations to know where the inputs should be written. As previously mentioned, SYZKALLER and NYX require complex specifications. In contrast, VIRTFUZZ only requires a short device definition, which can easily be created by reading the VirtIO device specification for the particular device.

```
{
  "virtio_id": 1,
  "virtqueue_num": 2,
  "virtqueue_tx": 1,
  "virtqueue_rx": 0,
  "features": [0, 16],
  "config": "ABABABABABAB0100DC05"
}
```

Listing 5: VirtIO configuration for a network device.

We demonstrated the high effort required for NYX to be adapted to our targets. Due to this, we consider an evaluation compared to VIRTFUZZ for WLAN and Bluetooth fuzzing out of scope for this paper. Additionally, there would still not be a guarantee to find similar bugs, discouraging this effort even further.

5.6.2. Adding New VIRTFUZZ Devices. To demonstrate VIRTFUZZ's flexibility, we implement a network device and an input device. This allows fuzzing of the Linux network stack and the input stack. We outline this process in the following for the network device: First, we derive a device definition from the VirtIO network device specification, which we show in Listing 5. This format is generic, but exclusively used by VIRTFUZZ, as no other universal VirtIO device is used. Second, we identify and annotate the entry point of the `virtio_net` driver to the networking stack. We display these annotations in Listing 6. With these simple changes, VIRTFUZZ can fuzz the Linux network stack. Implementations and patches for both devices are part of the released source code. As of writing this paper, the additional devices did not lead to new bug discoveries, but we only ran them briefly.

We conclude that VIRTFUZZ is easily adaptable to new VirtIO devices, especially in contrast to SYZKALLER and NYX, which require complex grammar definitions. Furthermore, even though we require some minor patches applied to the kernel, it is portable between different kernel versions. This answers **RQ4**.

6. Related Work

Fuzzing as a software testing method has been used since the 1990s [54]. Nonetheless, noteworthy bugs and vulnerabilities were also found in recent years. Much research was published based on AFL [55], a coverage-guided gray-box fuzzer. This research was incorporated into AFL++ [56] and LIBAFL [40] to serve as new baseline fuzzers.

Several different fuzzing aspects have been subject to improvements and research. The selection of seeds [57, 58, 22], input scheduling [42, 59, 60, 61, 62], and mutations [45,

```
static void receive_buf(struct virtnet_info *vi, struct
     receive_queue *rq, void *buf, unsigned int len, void
     **ctx, unsigned int *xdp_xmit, struct
     virtnet_rq_stats *stats) {
  struct net_device *dev = vi->dev;
  struct sk_buff *skb;
  struct virtio_net_hdr_mrg_rxbuf *hdr;

  kcov_ivshmem_start(); // start coverage collection

  [...]

  napi_gro_receive(&rq->napi, skb);
  kcov_ivshmem_stop(); // end collection on success
  return;

  frame_err:
  dev->stats.rx_frame_errors++;
  dev_kfree_skb(skb);
  kcov_ivshmem_stop(); // end collection on error
}
```

Listing 6: Annotations enabling VIRTFUZZ's coverage collection in Linux network packet reception.

63] are some of them. Furthermore, the research includes the usage of snapshots to improve fuzzing performance [13, 14] approaches using symbolic execution in hybrid fuzzing for better coverage [64, 65] or using manual annotations to represent internal program states [66].

Traditional fuzzing is used on user-space programs [54]. Recently, it has been applied to other targets, such as fuzzing firmware [7, 9, 10, 11], hypervisors [14, 67], device drivers [16, 17, 68, 69, 70], or file systems [71]. Since many of these approaches cannot be used for embedded devices or need significant adaptions to run on multiple firmware images, over-the-air fuzzing is common for wireless stacks [34, 33, 72, 35].

The Linux kernel is a popular fuzzing target. The following highlights the most popular Linux fuzzers and how our VirtIO-based approach compares to them.

KAFL [18] and its successors REDQUEEN [39] and NYX [14] are general kernel fuzzers. They accomplish strong results by using special hardware features to track coverage and thus are more independent from the operating system. Furthermore, by using hypercalls for input generation, they can be applied to arbitrary code paths. In contrast to their many strengths, their downside is that adaptation for a specific kernel target and version requires time and expert knowledge. Previously, we showed that our approach is easier portable between versions, as it does not rely on heavy kernel modifications.

SYZKALLER fuzzes the Linux kernel permanently on several servers, continuously leading to the discovery of numerous vulnerabilities [15]. It fuzzes system calls, which need to be described by a complex grammar but does not require any kernel modifications, making it work out-of-the-box with every recent kernel version. We showed that despite covering similar functions, SYZKALLER could not uncover the severe WLAN vulnerabilities we found in the Linux stack. Contrary to our approach, it is more complicated to adapt to new targets, as the related calls have to be described in a complex grammar first. Moreover, unique setup methods for the corresponding virtual devices must be introduced, especially for targeting interfaces such as WLAN and Bluetooth. As we use a VirtIO interface for sending inputs to the kernel, targeting different devices becomes much more accessible.

PERISCOPE [73] targets device drivers of the Linux kernel. Therefore, it records the interactions of a peripheral device and its corresponding driver by hooking into the Linux kernel's page fault mechanism. This allows PERISCOPE to monitor the drivers' memory access and to modify the exchanged data on-the-fly, which is used for fuzzing. Similar to our approach, a modified version of kcov is used to record coverage. Despite their similarities, VIRTFUZZ and PERISCOPE focus on different parts of the kernel: Our approach targets the driver-agnostic subsystem, while PERISCOPE focuses on specific drivers. The findings reflect this: all vulnerabilities found by PERISCOPE were in two Android Wi-Fi drivers, most vulnerabilities VIRTFUZZ has found (28 of 31) were either in the Bluetooth- or Wi-Fi subsystems. Another difference is the portability between kernel versions: We expect that our approach is more portable, as the set of required kernel modifications is less intruding. Furthermore, the architecture of PERISCOPE and VIRTFUZZ are fundamentally different: VIRTFUZZ runs a VM with the target kernel and interacts with it through a standard protocol (VirtIO) from the host machine. The architecture of PERISCOPE is, in that regard, more similar to SYZKALLER, since most components run inside the target.

Using LIBAFL [40], we could incorporate several state-of-the-art techniques into our kernel fuzzer. REDQUEEN introduced so-called *Input-to-state* mutations to overcome fuzzing roadblocks such as magic bytes comparisons [39]. VIRTFUZZ integrates a similar mechanism based on REDQUEEN and WEIZZ [44] by evaluating comparisons recorded through kcov. Furthermore, VIRTFUZZ uses the optimized mutation scheduling *MOPT* [45].

7. Conclusion

Designing new Linux kernel fuzzers can reveal bugs with a severe security impact, even though existing fuzzers analyze the Linux kernel continuously. While focusing on wireless subsystems exposing a zero-click RCE attack surface, we designed a fuzzer that can be adapted to any stack accessible through a VirtIO driver. VIRTFUZZ is based on a new universal VirtIO device. We built an additional easy-to-use proxy, enabling the collection of real-world inputs as seeds. We showed that using them leads to a headstart on a fuzzing campaign, thus increasing the fuzzing success. VIRTFUZZ uncovered bugs introduced into the Bluetooth stack as early as 2008 and the WLAN stack in 2019. Responsible disclosure of these bugs improved Linux wireless security. We will open-source VIRTFUZZ to increase the security of existing and future Linux kernel releases.

Acknowledgments

We thank Fabian Freyer for his feedback on this paper, Dominik Maier for the discussions about our fuzzer, and Johannes Berg for the quick response to our Wi-Fi vulnerabilities, as well as the insightful exchange regarding the kernel's subsystem.

This work has been co-funded by the LOEWE initiative (Hesse, Germany) within the emergenCITY centre. It has also been co-funded by the German Federal Ministery of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

Availability

We publish VIRTFUZZ under an open-source license at https://github.com/seemoo-lab/VirtFuzz.

References

[1] M. Vanhoef, "Fragment and forge: Breaking Wi-Fi through frame aggregation and fragmentation," in *Proceedings of the 30th USENIX Security Symposium*. USENIX Association, August 2021.

[2] M. Vanhoef and F. Piessens, "Key reinstallation attacks: Forcing nonce reuse in WPA2," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2017.

[3] J. Classen and M. Hollick, "Happy MitM: Fun and Toys in Every Bluetooth Device," in *WiSec '21: Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, C. Pöpper, M. Vanhoef, L. Batina, and R. Mayrhofer, Eds. ACM, June 2021, pp. 72–77, event Title: 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks. [Online]. Available: http://tubiblio.ulb.tu-darmstadt.de/130931/

[4] D. Antonioli, N. O. Tippenhauer, and K. B. Rasmussen, "The KNOB is broken: Exploiting low entropy in the encryption key negotiation of bluetooth BR/EDR," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1047–1061. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/antonioli

[5] D. Antonioli, N. O. Tippenhauer, and K. Rasmussen, "BIAS: Bluetooth Impersonation AttackS," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 549–562.

[6] M. von Tschirschnitz, L. Peuckert, F. Franzen, and J. Grossklags, "Method Confusion Attack on Bluetooth Pairing," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1332–1347.

[7] J. Ruge, J. Classen, F. Gringoli, and M. Hollick, "Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets," in *Proceedings of the 29th USENIX Security Symposium*, S. Capkun and F. Roesner, Eds. USENIX Association, August 2020, pp. 19–36. [Online]. Available: http://tubiblio.ulb.tu-darmstadt.de/130926/

[8] J. Classen, F. Gringoli, M. Hermann, and M. Hollick, "Attacks on Wireless Coexistence: Exploiting Cross-Technology Performance Features for Inter-Chip Privilege Escalation," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 1229–1245.

[9] D. Maier, L. Seidel, and S. Park, "BaseSAFE: Baseband SAnitized Fuzzing through Emulation," *CoRR*, vol. abs/2005.07797, 2020. [Online]. Available: https://arxiv.org/abs/2005.07797

[10] G. Hernandez, M. Muench, D. Maier, A. Milburn, S. Park, T. Scharnowski, T. Tucker, P. Traynor, and K. R. B. Butler, "FirmWire: Transparent Dynamic Analysis for Cellular Baseband Firmware," in *Symposium on Network and Distributed System Security (NDSS)*, 2022.

[11] E. Kim, D. Kim, C. Park, I. Yun, and Y. Kim, "BaseSpec: Comparative Analysis of Baseband Software and Cellular Specifications for L3 Protocols," in *Symposium on Network and Distributed System Security (NDSS)*, 2021.

[12] D. Heinze, J. Classen, and M. Hollick, "ToothPicker: Apple Picking in the iOS Bluetooth Stack," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[13] 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)*. USENIX Association, Aug. 2020, pp. 2541–2557. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/song

[14] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo

[15] D. Vyukov, "Syzkaller," 2015. [Online]. Available: https://github.com/google/syzkaller

[16] H. Peng and M. Payer, "USBFuzz: A framework for fuzzing USB drivers by device emulation," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2559–2575. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/peng

[17] 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 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2123–2138. [Online]. Available: https://doi.org/10.1145/3133956.3134069

[18] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-Assisted feedback fuzzing for OS kernels," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 167–182. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo

[19] OASIS Committee Specification 01, "Virtual I/O Device (VIRTIO) Version 1.2," July 2022, edited by Michael S. Tsirkin and Cornelia Huck. https://docs.oasis-open.org/virtio/virtio/v1.2/cs01/virtio-v1.2-cs01.html.

[20] R. Russell, "Virtio: Towards a de-facto standard for virtual i/o devices," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, p. 95–103, jul 2008. [Online]. Available: https://doi.org/10.1145/1400097.1400108

[21] D. Vyukov, "syzkaller: Adventures in Continuous Coverage-guided Kernel Fuzzing," 2020. [Online]. Available: https://www.youtube.com/watch?v=YwX4UyXnhz0

[22] A. Herrera, H. Gunadi, S. Magrath, M. Norrish, M. Payer, and A. L. Hosking, "Seed Selection for Successful Fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 230–243. [Online]. Available: https://doi.org/10.1145/3460319.3464795

[23] V. M. Manes, H. Han, C. Han, S. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The Art, Science, and Engineering of Fuzzing: A Survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, nov 2021.

[24] "kcov: code coverage for fuzzing," https://www.kernel.org/doc/html/v5.19/dev-tools/kcov.html.

[25] "The Kernel Address Sanitizer (KASAN)," https://www.kernel.org/doc/html/v5.19/dev-tools/kasan.html.

[26] "The Undefined Behavior Sanitizer - UBSAN," https://www.kernel.org/doc/html/v5.19/dev-tools/ubsan.html.

[27] "Kernel memory leak detector," https://www.kernel.org/doc/html/v5.19/dev-tools/kmemleak.html.

[28] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "Sok: Sanitizing for security," *CoRR*, vol. abs/1806.04355, 2018. [Online]. Available: http://arxiv.org/abs/1806.04355

[29] G. Motika and S. Weiss, "Virtio network paravirtualization driver: Implementation and performance of a de-facto standard," *Computer Standards & Interfaces*, vol. 34, no. 1, pp. 36–47, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0920548911000559

[30] A. Bulekov, B. Das, S. Hajnoczi, and M. Egele, "Morphuzz: Bending (Input) Space to Fuzz Virtual Devices," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 1221–1238. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/bulekov

[31] R. Mayrhofer, J. V. Stoep, C. Brubaker, and N. Kralevich, "The Android Platform Security Model," *ACM Trans. Priv. Secur.*, vol. 24, no. 3, apr 2021. [Online]. Available: https://doi.org/10.1145/3448609

[32] *Bluetooth Core Specification v5.3*, Bluetooth SIG Std., Rev. v5.3, Jul. 2021. [Online]. Available: https://www.bluetooth.com/specifications/specs/core-specification-5-3/

[33] M. E. Garbelini, V. Bedi, S. Chattopadhyay, S. Sun, and E. Kurniawan, "BrakTooth: Causing Havoc on Bluetooth Link Manager via Directed Fuzzing," *USENIX Security Symposium*, 2022. [Online]. Available: https://asset-group.github.io/papers/BrakTooth.pdf

[34] M. E. Garbelini, C. Wang, S. Chattopadhyay, S. Sun, and E. Kurniawan, "Sweyntooth: Unleashing mayhem over bluetooth low energy," in *USENIX Annual Technical Conference (USENIX ATC)*, 2020. [Online]. Available: https://asset-group.github.io//papers/SweynTooth.pdf

[35] E7mer, "Owfuzz: a WiFi protocol fuzzing tool," Nov 2021, https://github.com/alipay/Owfuzz.

[36] V. Palmiotti, "Put an io_uring on it: Exploiting the Linux Kernel," 2022. [Online]. Available: https://www.graplsecurity.com/post/iou-ring-exploiting-the-linux-kernel

[37] GitHub, "Copilot – Your AI pair programmer," 2022. [Online]. Available: https://github.com/features/copilot

[38] QEMU, "A generic and open source machine emulator and virtualizer," 2022. [Online]. Available: https://www.qemu.org/

[39] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with Input-to-State Correspondence," in *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[40] A. Fioraldi, D. Maier, D. Zhang, and D. Balzarotti, "Libafl: A framework to build modular and reusable fuzzers," 2022, http://193.55.114.4/docs/ccs22_fioraldi.pdf.

[41] M. Zalewski, "Technical whitepaper for afl-fuzz." [Online]. Available: https://lcamtuf.coredump.cx/afl/technical_details.txt

[42] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-Based Greybox Fuzzing as Markov Chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1032–1043. [Online]. Available: https://doi.org/10.1145/2976749.2978428

[43] M. Wu, L. Jiang, J. Xiang, Y. Huang, H. Cui, L. Zhang, and Y. Zhang, "One Fuzzing Strategy to Rule Them All," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 1634–1645.

[44] A. Fioraldi, D. C. D'Elia, and E. Coppa, "WEIZZ: Automatic Grey-Box Fuzzing for Structured Binary Formats," in *Proceedings of the 29th ACM SIGSOFT*

International Symposium on Software Testing and Analysis, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–13. [Online]. Available: https://doi.org/10.1145/3395363.3397372

[45] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "MOPT: Optimized Mutation Scheduling for Fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1949–1966. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/lyu

[46] "CVE-2022-42719," Available from NVD, CVE-ID CVE-2022-42719., 2022. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2022-42719

[47] "KASAN: slab-out-of-bounds read in hci_le_meta_evt." [Online]. Available: https://syzkaller.appspot.com/bug?extid=e3fcb9c4f3c2a931dc40

[48] "Integrated syzkaller with mac80211_hwsim," 2020. [Online]. Available: https://github.com/google/syzkaller/pull/2079

[49] AOSP, "Gabeldorsche Architecture," 2022. [Online]. Available: https://chromium.googlesource.com/aosp/platform/system/bt/+/refs/heads/bringup/gd/docs/architecture/architecture.md

[50] IEEE, "IEEE Standard for Information Technology–Telecommunications and Information Exchange between Systems Local and Metropolitan Area Networks–Specific Requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 1: Enhancements for High-Efficiency WLAN," *IEEE Std 802.11ax-2021 (Amendment to IEEE Std 802.11-2020)*, pp. 1–767, 2021.

[51] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2123–2138. [Online]. Available: https://doi.org/10.1145/3243734.3243804

[52] S. Schulz and M. Tarral, "Linux branches for fuzzing with kAFL," 2022, https://github.com/IntelLabs/kafl.linux.

[53] S. Schumilo and C. Aschermann, "Initial Release of Nyx," 2021, https://github.com/IntelLabs/kafl.linux/commit/c612e238e455c34255bdb92efa7fd2fd963d287b.

[54] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, p. 32–44, dec 1990. [Online]. Available: https://doi.org/10.1145/96267.96279

[55] M. Zalewski, "american fuzzy lop." [Online]. Available: https://lcamtuf.coredump.cx/afl/

[56] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining Incremental Steps of Fuzzing Research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi

[57] S. Pailoor, A. Aday, and S. Jana, "Moonshine: Optimizing OS fuzzer seed selection with trace distillation," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 729–743. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/pailoor

[58] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 861–875. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/rebert

[59] S. Karamcheti, G. Mann, and D. S. Rosenberg, "Adaptive Grey-Box Fuzz-Testing with Thompson Sampling," *CoRR*, vol. abs/1808.08256, 2018. [Online]. Available: http://arxiv.org/abs/1808.08256

[60] Y. Chen, M. Ahmadi, R. M. farkhani, B. Wang, and L. Lu, "MEUZZ: Smart Seed Scheduling for Hybrid Fuzzing," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. San Sebastian: USENIX Association, Oct. 2020, pp. 77–92. [Online]. Available: https://www.usenix.org/conference/raid2020/presentation/chen

[61] C. Zhou, M. Wang, J. Liang, Z. Liu, and Y. Jiang, "Zeror: Speed Up Fuzzing with Coverage-sensitive Tracing and Scheduling," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 858–870.

[62] D. She, A. Shah, and S. Jana, "Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis," 2022. [Online]. Available: https://arxiv.org/abs/2203.12064

[63] P. Jauernig, D. Jakobovic, S. Picek, E. Stapf, and A.-R. Sadeghi, "DARWIN: Survival of the Fittest Fuzzing Mutators," *arXiv e-prints*, p. arXiv:2210.11783, Oct. 2022.

[64] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 745–761. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[65] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "HFL: hybrid fuzzing on the linux kernel," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/hfl-hybrid-fuzzing-on-the-linux-kernel/

[66] 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, pp. 1597–1612.
[67] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "HYPER-CUBE: High-Dimensional Hypervisor Fuzzing," 2020.
[68] Z. Ma, B. Zhao, L. Ren, Z. Li, S. Ma, X. Luo, and C. Zhang, "Printfuzz: Fuzzing linux drivers via automated virtual device simulation," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 404–416. [Online]. Available: https://doi.org/10.1145/3533767.3534226
[69] D. Maier and F. Toepfer, "BSOD: Binary-Only Scalable Fuzzing Of Device Drivers," in *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 48–61. [Online]. Available: https://doi.org/10.1145/3471621.3471863
[70] K. Kim, T. Kim, E. Warraich, B. Lee, K. R. B. Butler, A. Bianchi, and D. Jing Tian, "FuzzUSB: Hybrid Stateful Fuzzing of USB Gadget Stacks," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 2212–2229.
[71] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim, "Fuzzing file systems via two-dimensional input space exploration," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 818–834.
[72] M. E. Garbelini, Z. Shang, S. Chattopadhyay, S. Sun, and E. Kurniawan, "Towards Automated Fuzzing of 4G/5G Protocol Implementations Over the Air," *IEEE Global Communications Conference (GLOBECOM)*, 2022. [Online]. Available: https://asset-group.github.io//papers/AutoFuzz4G5G.pdf
[73] 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 *Network and Distributed System Security Symposium (NDSS)*, 2019.

Appendix A. Meta-Review

A.1. Summary

This paper assesses the security of Linux kernel interfaces by developing a fuzzer called VirtFuzz, which uses the VirtIO interface used by virtual machines to access virtual devices exposed by a hyper visor. Since VirtIO drivers know they are running in a virtualized environment, they can allow higher performance from guest VMs. VirtFuzz uses the VirtIO interface to pass fuzzed packets to the guest VM, and these interfaces cover large parts of the kernel subsystem, allowing the authors to fuzz Wi-Fi and Bluetooth subsystems. The fuzzer is built using libAFL and code coverage is provided through kcov. 31 new vulnerabilities are found in the Linux network subsystems, 6 of high severity.

A.2. Scientific Contributions

- Independent Confirmation of Important Results with Limited Prior Research
- Creates a New Tool to Enable Future Science
- Identifies an Impactful Vulnerability
- Provides a Valuable Step Forward in an Established Field

A.3. Reasons for Acceptance

1) New vulnerabilities in the Wi-Fi subsystem with major impact have been discovered with this methodology, and 31 bugs total have been discovered which is very significant.
2) Fuzzing is easier to set up compared to approaches such as syzcaller and fuzzing speed is increased.
3) Does not require heavy patching of the Linux kernel or qemu.
4) Open-source deployment promised by authors.

A.4. Noteworthy Concerns

Improvements to Bluetooth coverage through VirtFuzz approach is relatively minimal compared to Wi-Fi. It would be worthwhile for the authors to provide a limitations section or guidance to users of the tool so that it is clear when it can best be used to demonstrate advantages over other techniques.

Appendix B. Response to the Meta-Review

We will provide guidance on how to best use VirtFuzz in the README file of the published source code. This will enable those who build upon it to enhance driver fuzzing to make an informed decision and get started more easily.