ReUSB: Replay-Guided USB Driver Fuzzing

Jisoo Jang, Minsuk Kang, and Dokyung Song, *Yonsei University*

https://www.usenix.org/conference/usenixsecurity23/presentation/jang

This paper is included in the Proceedings of the
32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

ReUSB: Replay-Guided USB Driver Fuzzing

Jisoo Jang*, Minsuk Kang*, Dokyung Song†
Department of Computer Science
Yonsei University

Abstract

Vulnerabilities in device drivers are constantly threatening the security of OS kernels. USB drivers are particularly concerning due to their widespread use and the wide variety of their attack vectors. Recently, fuzzing has been shown to be effective at finding vulnerabilities in USB drivers. Numerous vulnerabilities in USB drivers have been discovered by existing fuzzers; however, the number of code paths and vulnerabilities found, unfortunately, has stagnated. A key obstacle is the statefulness of USB drivers; that is, most of their code can be covered only when given a specific sequence of inputs.

We observe that record-and-replay defined at the trust boundary of USB drivers directly helps overcoming the obstacle; deep states can be reached by reproducing recorded executions, and, combined with fuzzing, deeper code paths and vulnerabilities can be found. We present ReUSB, a USB driver fuzzer that guides fuzzing along *two-dimensional record-and-replay* of USB drivers to enhance their fuzzing. We address two fundamental challenges: faithfully replaying USB driver executions, and amplifying the effect of replay in fuzzing. To this end, we first introduce a set of language-level constructs that are essential in faithfully describing concurrent, two-dimensional traces but missing in state-of-the-art kernel fuzzers, and propose time-, concurrency-, and context-aware replay that can reproduce recorded driver executions with high fidelity. We then amplify the effect of our high-fidelity replay by guiding fuzzing along the replay of recorded executions, while mitigating the slowdown and side effects induced by replay via replay checkpointing. We implemented ReUSB, and evaluated it using two-dimensional traces of 10 widely used USB drivers of 3 different classes. The results show that ReUSB can significantly enhance USB driver fuzzing; it improved the code coverage of these drivers by 76% over a strong baseline, and found 15 previously unknown bugs.

*Equal contribution.
†Corresponding author.

1 Introduction

Device drivers continue to pose a significant threat to the security of OS kernels. They expose a wide, two-dimensional attack surface—system call and peripheral interface—but they are, unfortunately, too large to be bug-free and considered trustworthy. A peripheral interface widely used by drivers to interact with their corresponding peripheral devices is Universal Serial Bus (USB). The USB interface has constantly attracted adversaries due to its ubiquity and the variety of threats it can be exposed to [75]. Past attacks targeting the USB interface range from the ones launched by adversaries in proximity either with (e.g., evil maid attacks and social engineering attacks [76]) or without contact (e.g., attacks via wireless communication [8, 9, 11–14]), to even completely remote ones (e.g., attacks through USB-over-Ethernet/IP [43]).

Recently, fuzzing has proved to be extremely effective at finding vulnerabilities in kernel-mode device drivers [29, 46, 56, 62, 63], particularly in USB drivers [28, 56]. Despite their early success in finding numerous vulnerabilities, however, the effectiveness of existing USB driver fuzzers is less than ideal. Syzkaller [29], a state-of-the-art USB driver fuzzer, for example, suffers dismally low code coverage for most of USB drivers in Linux. A key challenge is the statefulness of USB drivers, which can be formulated as a *dependency challenge* [33] for kernel fuzzers. Addressing this challenge requires fuzzers to generate a specific sequence of input actions (e.g., system calls) such that the value and ordering dependencies [32, 53] of each action have all been resolved by the time it is invoked, but existing fuzzers, unfortunately, fail to do so.

We observe that a natural yet promising direction towards addressing the dependency challenge in kernel driver fuzzing is *record-and-replay*. The idea is to record normal executions of USB drivers at their two-dimensional trust boundary—while they are operating with their corresponding devices at the peripheral interface and user-mode programs at the system call interface—and *faithfully* replay the resulting two-dimensional traces to reconstruct deep USB driver states during fuzzing. A faithful replay, when combined with fuzzing, could potentially unlock, at the expense of one-time recording efforts, many driver code paths whose execution is conditioned on the driver being in deep normal operation states.

Although state-of-the-art kernel fuzzers can leverage recorded inputs as initial seeds for evolutionary fuzzing [29, 53, 62], they are not designed to harness the full power of input traces. As existing fuzzers focus heavily on evolving the input corpus into a set of *minimal* inputs rather than *realistic* ones, their input execution engines are not tailored to accurately replaying lengthy, concurrent two-dimensional input traces of USB drivers, and therefore fail to reconstruct deep states of USB drivers observed during recording. Worse, aggressive evolutionary fuzzing without accurate replay can inadvertently discard valuable parts of input traces, or even prune entire input traces, when executing them yields unstable or previously seen coverage only.

This paper proposes and advocates a new principle dubbed *replay-guided fuzzing* in USB driver fuzzing, which guides fuzzing along an accurate replay of recorded USB driver executions. This principle differs from evolutionary, trace-based approaches [29, 53], in that it requires input traces be *faithfully* replayed to reconstruct deep driver states observed while recording. Realizing this principle, however, poses the following challenges. First, the asynchronous and concurrent nature of drivers often imposes temporal constraints on input action invocations; that is, actions may not achieve the desired effects depending on when they are invoked. Second, driver executions could diverge across record and replay; there are many code paths that can execute concurrently, which can interleave differently during fuzzing. Third, normal operations of a driver as well as their replay tend to require long execution time, which could significantly slowdown fuzzing. Forth, input actions often make persisting side effects on the system, which could influence subsequent fuzzing iterations.

We present ReUSB, a replay-guided USB driver fuzzer design that can address these challenges, thereby making USB driver fuzzing significantly more effective than prior work. We first introduce a set of language-level constructs—execution context, dispatch mode, and post-dispatch delay—that are lacking in the input program description language used by existing fuzzers, but essential in faithfully describing concurrent, two-dimensional traces of USB driver executions. ReUSB uses these new constructs, during replay, (i) to exercise timing and concurrency control when dispatching recorded actions for execution, and (ii) to dynamically schedule an execution context based on the context of potentially unordered USB request messages. These techniques, in effect, allow ReUSB (i) to better satisfy temporal constraints of actions, and (ii) to react to divergences across record and replay runs, thereby increasing the fidelity of replay. We then amplify the effect of faithful replay by strictly guiding fuzzing along the replay, and by mitigating slowdown and side effects induced by prolonged and realistic replay via replay checkpointing. These techniques together constitute *replay-guided fuzzing* that can reach deeper USB driver code paths than the state-of-the-art.

The idea of using record-and-replay to aid software and hardware analysis has been around for decades. Our work distinguishes, however, from prior work in that we aim to replay (i) USB driver executions, and (ii) by only invoking input actions at their two-dimensional trust boundary without intrusively interposing on driver executions. Prior record-and-replay systems target different software/hardware—e.g., user-mode programs [51], virtual machines [17, 19, 24, 25], and hardware [30, 54, 79]—and typically require much more intrusive interposition during either record, replay, or both. For example, record-and-replay systems for debugging or tolerating failures of concurrent, nondeterministic programs [7, 17, 27, 44, 55, 65], unlike ReUSB, typically interpose on all synchronization events in addition to external program inputs.

We implemented ReUSB by modifying Syzkaller [29], a state-of-the-art kernel fuzzer. Our evaluation shows that, ReUSB, our approach that guides USB driver fuzzing along accurate replay, can significantly enhance the effectiveness of USB driver fuzzing: By fuzzing 10 inherently stateful USB drivers using ReUSB, we increased the coverage of the driver code by up to 76% over a strong baseline, and found 20 bugs, of which 15 were previously unknown. These results demonstrate the benefit of faithful replay in USB driver fuzzing. Besides these end results of fuzzing, we also present a detailed, systematic evaluation of the fidelity of our proposed record-and-replay designed and implemented at the two-dimensional trust boundary of USB drivers. In summary, we make the following contributions.

- **A New Approach.** We propose a *replay-guided* USB driver fuzzer design that uses two-dimensional record-and-replay to enhance USB driver fuzzing. It works by (i) recording thousands of actions invoked through the trust boundary of a driver during its normal execution with user-mode programs and hardware, and (ii) guiding fuzzing into deeper driver code paths along the faithful replay of the recorded actions.
- **Replay and Fuzzing Techniques.** We identify and propose language-level constructs lacking in the input program description language used by existing fuzzers, and propose time-, concurrency-, and context-aware replay that uses these constructs to faithfully reproduce USB driver executions. We also propose replay-guided fuzzing to amplify the effect of faithful replay, and replay checkpointing to mitigate the slowdown and side effects induced by replay.
- **Practical Impact.** We implemented ReUSB and evaluated it using 10 inherently stateful USB drivers that implement wireless communication protocols: Wi-Fi, Bluetooth, and NFC. We increased the driver coverage by up to 76% over a strong baseline, and found 15 previously unknown bugs. We responsibly disclosed them along with patches to the kernel community, and they were all confirmed and patched.

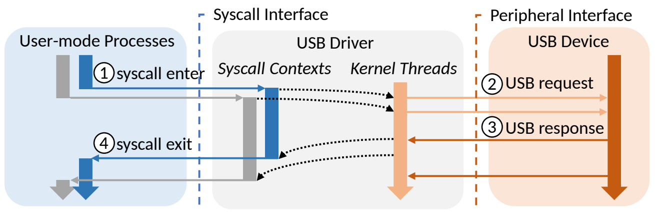


Figure 1: USB driver execution flow.

2 Background

2.1 USB Device Drivers

As depicted in Figure 1, kernel-mode USB drivers have a two-dimensional input space, comprising the system call and peripheral interface, and work as follows. ① User-mode processes wishing to interact with the USB device make system calls through the system call interface. ② USB drivers make USB requests to the USB device as a result of either system calls or other events triggered by the kernel. ③ USB devices reply by sending USB responses, and drivers receive these via the peripheral interface. ④ Drivers can now hand over the responses to the process context that has been waiting for them in the kernel mode. Observe that there are multiple concurrently running execution contexts (e.g., kernel threads) in drivers, and that the aforementioned events that occur can therefore be interleaved differently across driver executions.

The input space of USB drivers can be attacked by a wide variety of adversaries: (i) the system call interface by local, user-mode adversaries, and (ii) the peripheral interface by nearby (via attacks with or without physical contact) or remote (via attacks over network) adversaries. Wi-Fi drivers, for example, expose a nearby, contactless attack surface that can wirelessly be bypassed and compromised [9, 13, 14]. USB drivers could also be exposed to remote adversaries as well, via, e.g., USB-over-Ethernet/IP solutions that are nowadays increasingly being adopted in work-from-home scenarios.

2.2 Kernel Fuzzing

Fuzzing is a dynamic testing technique widely used to find bugs in USB drivers [28, 56, 64] and other kernel subsystems [22, 29, 36, 37, 39, 53, 60, 63, 66, 81]. Notationally, kernel fuzzing can be described as iteratively (i) *generating* program P that comprises an ordered set of input actions $P = \{a^{(i)}\}$ for $i = \{1, 2, \ldots\}$, and (ii) *executing* P to exercise kernel code paths. For USB drivers, an input action could be either invoking a system call or responding to a USB request made by the driver, which we denote by $a_{syscall}$ and a_{usb}, respectively. A key challenge for kernel fuzzers is how they *generate* P such that executing P triggers previously uncovered code paths and unknown bugs in the kernel. Existing fuzzers use a variety of techniques to generate programs: evolutionary fuzzing [16, 29, 82], symbolic execution [37, 38, 58, 62], tracing [46, 53], and hand-written grammars [29]. Once an input program has been generated, existing fuzzers consecutively execute the input actions [29] or consume the input bytes [56, 63] in the order they appear in the program.

2.3 Record-and-Replay

Record-and-replay systems for concurrent, event-driven programs differ mainly in (i) *where* (i.e., the boundaries at which) record-and-replay is defined (e.g., input boundaries or inter-process synchronization boundaries), and (ii) *what* events are recorded (e.g., input events or inter-process synchronization events). Prior work makes different design choices depending on the desired degree of (i) *fidelity*, i.e., how faithfully, in terms of semantics and performance, a recorded execution shall be reproduced, and (ii) *intrusiveness*, i.e., how intrusively the target's execution can be interposed during record-and-replay.

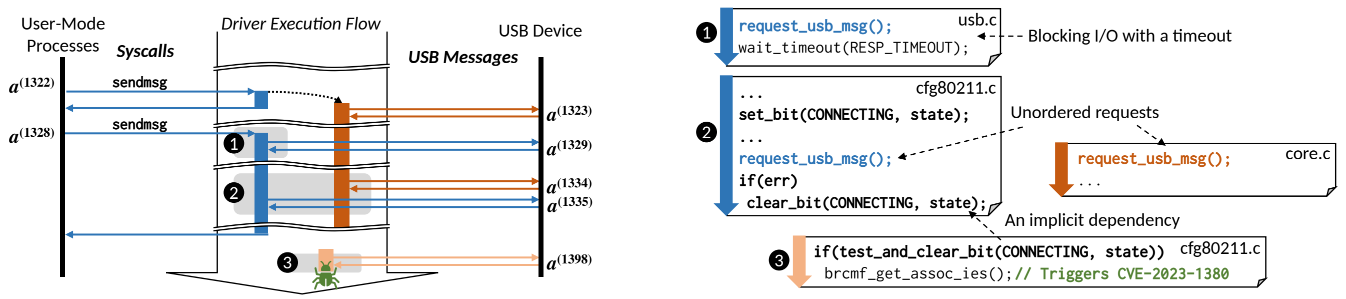
Fidelity vs. Intrusiveness. Record-and-replay designed for retrospective analysis of concurrent programs such as debugging [7, 24, 27, 50, 65], intrusion analysis [25], or answering queries about past system state [23] typically aims to achieve high *semantic* fidelity. Achieving this generally requires a *more intrusive* interposition of the target; for example, a fully deterministic replay would require interposing and recording all nondeterministic events. On the other hand, record-and-replay designed for performance benchmarking [77, 79] aims for high *performance* fidelity, i.e., preserving, during replay, the timing characteristics of a program's execution observed while recording [79]. This is often accomplished, unlike semantic fidelity, with *less intrusive* interpositions during record-and-replay, to minimize their run-time overheads.

Record-and-Replay for Kernel Fuzzing. Record-and-replay, when used to enhance kernel fuzzing, is typically defined, *non-intrusively*, at the trust boundary of the kernel—e.g., the system call interface [53] or the PCI interface [46]—due to following benefits. First, bugs triggered by only controlling external inputs provided from outside of the trust boundary directly constitute security threats that can be posed by adversaries at the boundary. Second, the less intrusive interposition, the more applicability; this is because interposing on the kernel's execution typically requires instrumentation or modification at the source code level [34, 54], which is not possible for commercial-off-the-shelf OS kernels. Third, intrusive interpositions could introduce significant run-time overheads during replay [46, 63], which could slowdown fuzzing.

3 Motivation & Design Principle

3.1 Challenges

We now explain the challenges of finding deep bugs in USB drivers by using CVE-2023-1380 as a running example. It is a hard-to-reach, previously unknown kernel memory disclosure



```
request_usb_msg();
wait_timeout(RESP_TIMEOUT);
```

```
...
set_bit(CONNECTING, state);
...
request_usb_msg();
if(err)
 clear_bit(CONNECTING, state);
```

```
request_usb_msg();
...
```

```
if(test_and_clear_bit(CONNECTING, state))
 brcmf_get_assoc_ies();// Triggers CVE-2023-1380
```

Figure 2: Execution flow of BCM43236's Linux Wi-Fi driver that triggered CVE-2023-1380.

bug we found in BCM43236's Linux Wi-Fi driver. Figure 2 shows the observed execution flow of the driver that triggered this bug, annotated with (i) a subset of input actions invoked (see §2.2 for notation), and (ii) a simplified version of the relevant source code. The 1398th action $a^{(1398)}_{usb}$ triggers the bug in `brcmf_get_assoc_ies`, which is called only when the driver is in the `CONNECTING` state (❸ in Figure 2). Putting the driver into this state requires the driver to initialize itself, and the network interface to be brought up; technically, it requires *transitively* satisfying hundreds of $a^{(1398)}_{usb}$'s dependencies—both implicit and explicit dependencies identified by prior work [18, 32, 33, 53]—and, additionally, addressing the following challenges we identified.

C1. Temporal Constraints. The effect of invoking an input action is often dependent on the passage of time as well; that is, some actions can make their desired effects on the target only when they are invoked in a certain time interval. Actions often need to be invoked after some delay, before some timeout, or concurrently while previously invoked ones are executing. Triggering this bug indeed requires several input actions (e.g., $a^{(1328)}_{syscall}$ and $a^{(1329)}_{usb}$ in Figure 2) be concurrently invoked, and before a timeout (see ❶). If not, the driver simply signals an error after the timeout, failing to initialize itself.

C2. Unordered Concurrent Requests. Triggering deep bugs in USB drivers often also requires a set of input actions be invoked in a nondeterministic order. Invoking $a^{(1322)}_{syscall}$ and $a^{(1328)}_{syscall}$, for example, makes the driver generate multiple *unordered* USB requests concurrently from different execution contexts (see ❷), which requires a set of USB response actions be invoked in a specific order that is different in each run (e.g., the actions starting from $a^{(1329)}_{usb}$ to $a^{(1335)}_{usb}$ in Figure 2). If the unordered USB requests were not served with their expected responses, the driver would simply clear the `CONNECTING` bit, putting itself into a state where the bug can no longer be triggered.

C3. Lengthy Inputs. This bug also suggests that triggering deep bugs in USB drivers requires a long sequence of input actions be invoked, which takes a long time to execute; in fact, the execution flow that triggered the bug involved invoking

Table 1: Comparison with state-of-the-art kernel fuzzers, illustrating realistic driver executions enabled by ReUSB.

	Syzkaller [29]	MoonShine [53]	USBFuzz [56]	**ReUSB**
Input Action Types	**Syscall+USB**	Syscall	USB	**Syscall+USB**
Max. # of Actions	20	Dozens	Dozens	**12,000**
Program Timeout	Seconds	Seconds	Seconds	**Up to 5min**

1,094 system calls and responding to 304 USB requests, which in total lasted about 9.7 seconds. This poses a significant challenge for USB driver fuzzers, because it implies that they need to generate and execute large input programs, which could make fuzzing less effective due to both the enlarged mutation space and the reduced fuzzing throughput.

C4. Side Effects. Interacting with low-level device drivers including this buggy USB driver requires making various changes on low-level, global system states such as the network interfaces exposed through device files. Input programs generated by fuzzers need to access these files from user space to interact with the driver, but doing so, unfortunately, could make system state changes that outlive their execution, which, in turn, could influence subsequent fuzzing iterations.

3.2 State-of-the-art

State-of-the-art kernel fuzzers [29,53,56] fall short in addressing the above challenges, and thus fail to reconstruct deep driver states and trigger deep bugs, e.g., our running example.

Synchronous Execution. Existing fuzzers synchronously execute each action in a program; that is, they invoke an action as soon as the previously invoked one returns. Synchronous execution, however, cannot satisfy temporal constraints arising from delays, timeouts, or concurrency used by USB drivers. (C1). For instance, synchronous execution may result in starvation: When a program synchronously invokes a system call action whose return requires a subsequent USB response action be invoked, the program would *starve*—the USB action can never be invoked. Some fuzzers such as Syzkaller provide an asynchronous mode of execution, but using this mode without considering delayed realization of an action's effects may result in invoking subsequent actions too early.

Sequential Execution. Existing fuzzers execute actions sequentially, in the order they appear in the program. Due to nondeterministic interleavings of concurrently executing driver code paths (C2), however, USB drivers can make USB requests in a different order in each execution. This means that sequentially providing USB responses in the order they appear in the program may not successfully serve the requests that interleave differently during fuzzing. Failure to fulfill requests may result in shallow or unstable code coverage, which degrades the performance of existing fuzzers due to their aggressive coverage-based input minimization and pruning.

Constrained Program Size and Execution Time. Existing fuzzers highly constrain both the size (i.e., the number of actions contained in a program) and the execution time of input programs, as shown in Table 1, to prevent its execution from causing prolonged delays while fuzzing (C3). As shown earlier with the input program that triggers a real bug, however, the constraints used by existing fuzzers—e.g., less than or equal to 20 actions, and a timeout of several seconds in the case of Syzkaller [29]—are not nearly enough to exercise deep driver code paths, and bugs in them.

Constrained Program Behavior. To reduce interference between program executions caused by their system-wide side effects while fuzzing (C4), existing fuzzers, e.g., Syzkaller, confine the execution of each program with multiple process-level sandboxes, which serve two purposes: First, they prevent each program from making disruptive system state changes that could affect subsequent program executions. Second, they confine the lifetime of the resources created by each program; they are automatically destroyed as the program finishes executing. Unfortunately, however, these sandboxes cannot be employed when fuzzing low-level drivers, because programs must be able to access system state to operate the drivers.

3.3 Design Principle & Goals

Motivated by the deficiencies of state-of-the-art fuzzers in solving the challenges we identified, we propose a *record-and-replay* approach to USB driver fuzzing. Record-and-replay is a design principle that goes beyond using recorded inputs merely as *initial seeds* [62] or as *fragments* [35] for input generation during fuzzing, in that, under this principle, the execution of the target observed during recording shall faithfully be reproduced by *accurately replaying recorded inputs*. Fuzzing stateful targets such as USB drivers can greatly benefit from deep states reconstructed via accurate replay, because deep bugs in USB drivers—as we showed with our running example—can only be triggered from deep states. Inspired by this insight as well as the challenges we identified, we set the following as the goals of our USB driver fuzzer design.

$\boxed{\text{G1}}$ **Non-Intrusive, High-Fidelity Replay.** We aim first to achieve high-fidelity record-and-replay without intrusive interpositions. High-fidelity replay, which can bring fuzzers to deeper code paths, would require addressing (i) temporal constraints imposed on input action invocations (C1), and (ii) nondeterministic interleavings of unordered events (C2). We aim to address them by only controlling external inputs, because intrusive interpositions could complicate threat assessment and make our fuzzer less applicable (see §2.3).

$\boxed{\text{G2}}$ **High-Speed, Clean-State Fuzzing.** We simultaneously aim to mitigate the run-time overheads and side effects necessarily caused by realistic driver executions reproduced with replay; that is, we aim to execute a sequence of programs at a high speed, even though each consists of more than thousands of actions derived from traces of recorded executions (C3), and, additionally, with no interference between program executions, even though they are not contained within process-level sandboxes and thus cause system-wide side effects (C4).

Non-Goals. We do not aim to achieve full determinism nor high performance fidelity (see §2.3). Also, inferring a model of operation of the target software/hardware from multiple traces for the purpose of (i) rehosting the target software/hardware [20, 26, 30, 31, 48, 54] or (ii) model-guided (or grammar-guided) input mutation [18, 32, 62, 66] is not our goal.

4 ReUSB Design

We now present ReUSB, a replay-guided USB driver fuzzer designed to leverage faithful yet non-intrusive two-dimensional record-and-replay for an effective USB driver fuzzing. ReUSB is designed, (i) given a trace program P derived from the traces of recorded driver executions, to reproduce them with high fidelity yet non-intrusively at the trust boundary of USB drivers (achieving $\boxed{\text{G1}}$), and, (ii) given a sequence of programs $\{P^{(1)}, P^{(2)}, P^{(3)}, \ldots\}$ mutated from P during fuzzing, to execute them at a high speed without any interference between their executions ($\boxed{\text{G2}}$).

Overview. As depicted in Figure 3, ReUSB works in two phases: recording and replay-guided fuzzing. To directly model realistic threats posed by adversaries, we define record-and-replay *non-intrusively* at the two-dimensional trust boundary—the system call and peripheral interface—of USB drivers. ① During the recording phase, we interpose on *all* actions invoked through both dimensions of the boundary, recording their time, duration, execution context, payload, etc. (§4.1). ② The raw traces of recorded driver executions are then compiled into a *trace program*, in which actions are annotated with our language-level constructs that facilitate their accurate replay (§4.2). ③ Then, mutational fuzzing can be started with ReUSB by using a set of such trace programs as its initial seed corpus. ④ During fuzzing, each mutated program is converted into our own per-context bytecode representation (§4.3), which is then executed by our bytecode executor that employs time-, concurrency-, and context-aware replay for a faithful reproduction of recorded driver executions (§4.4). ⑤ To mitigate the impact of prolonged and side-

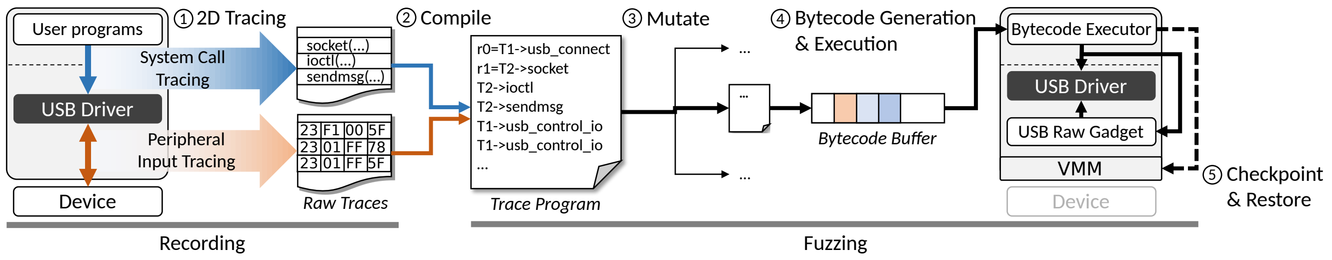


Figure 3: ReUSB overview.

effecting replay of lengthy trace programs on fuzzing, ReUSB uses replay-guided mutational fuzzing, and accelerates it by checkpointing replay runs at fine-grained intervals, thereby amplifying the effect of accurate replay on fuzzing (§4.5).

Key Techniques. ReUSB accomplishes the two design goals with the following replay and fuzzing techniques.

- **Time-, Concurrency-, and Context-Aware Replay.** For a high-fidelity replay, we introduce three language-level constructs: *execution context*, *dispatch mode*, and *post-dispatch delay*. ReUSB uses (i) post-dispatch delays to control the timing of each input action's invocation, and (ii) both execution contexts and dispatch modes to control concurrency between input actions invoked from different execution contexts; such time- and concurrency-aware replay increases the likelihood of satisfying the temporal constraints of input actions. In addition, ReUSB dynamically controls scheduling of execution contexts at run time by using events generated by the driver as a contextual cue; such context-aware replay allows ReUSB to react to diverging interleavings of unordered events as they manifest at run time.
- **Replay-Guided Fuzzing with Replay Checkpointing.** To amplify the effect of faithful replay on fuzzing, ReUSB performs *replay-guided fuzzing*, which focuses CPU time on fuzzing trace programs whose execution exercises deep code paths of drivers that were exercised during their normal executions. ReUSB then accelerates replay-guided fuzzing and, simultaneously, eliminates interference between program executions, by leveraging an incremental VM checkpoint-and-restore [61, 64], with our *replay checkpointing* policy that checkpoints reference replay runs at small, fixed intervals. This policy, when applied to replay-guided fuzzing, can frequently restore the VM to a checkpoint that corresponds to a deep driver state during fuzzing, thereby reducing the impact of faithful replay on the fuzzing speed, and eliminating side effects of realistic replay.

4.1 Execution Recording

We record the full payload of *all* input actions invoked at the two-dimensional trust boundary of a USB driver: the ones at the peripheral interface (e.g., USB responses) and others at the system call interface (e.g., system calls), thereby generating complete observations of external inputs for the recorded driver execution. In addition, we record the *timing* and *context* of these external inputs, e.g., USB requests corresponding to USB responses. The resulting trace, when compiled and executed with ReUSB, can recreate the recorded driver execution with high fidelity, without physical devices.

Precisely, the following must be recorded: for each system call, (i) the timestamp of its invocation and return, (ii) the execution context that invoked the call (i.e., the thread ID), (iii) a deep copy of all the arguments, and (iv) the return value; for each USB response, we record (i) the creation time of its corresponding request, (ii) the execution context where the request was created (i.e., the ID of the kernel thread or the workqueue that created the request), (iii) the arrival time of the response message, and (iv) the full payload of the request and response messages. Our execution recording comprises a set of raw system call and USB message traces, which include all the recorded values described above, plus the root filesystem image. This image is required during replay when any recorded system call accesses a file available in the filesystem.

4.2 Trace Compilation

The raw traces of a recorded execution are *compiled* into a trace program, which is of a form that can easily be (i) modified, i.e., fuzzed, and (ii) executed to accurately reproduce the recorded driver execution. We use, as the language used to describe an input program, the syscall description language of Syzkaller [29], which we augmented with a set of new constructs—*execution context*, *dispatch mode*, and *post-dispatch delay*—as described below.

Compiling Raw Traces. Each raw trace, either a system call trace or a USB message trace, is first translated into a ReUSB program. The translation process is largely straightforward: Each system call is translated into a system call action $a_{syscall}$; each USB response and its context (e.g., their corresponding USB requests) is translated into a single USB response action a_{usb}. All the translated ReUSB programs are then merged into a single *trace program*, where actions are chronologically ordered using the time at which they start influencing the driver execution; a system call action is considered influencing the driver execution at its invocation time, while a USB response

```
T17->sendmsg$unix(r29, &(0x7f0000036500)={0x0, 0x0,
↪  &(0x7f00000364c0)=[{&(0x7f00000363c0)="6c04010134000000...",
↪  0x98}, {&(0x7f0000036480)="110000006f72672e...", 0x34}], 0x2},
↪  0x4000) +258us
T17->poll(&(0x7f0000036540)=[{r23, 0x1}, {r24, 0x1}, {r29, 0x1},
↪ {r30, 0x1}, {r31, 0x1}, {r37, 0x1}, {r38, 0x1}, {r39, 0x1},
↪ {r43, 0x1}, {r44, 0x1}, {r45, 0x1}, {r46, 0x1}], 0xc,
↪  0xffffffffffffffff) & +47482us
T2->syz_usb_control_io(r0, 0x0, &(0x7f0000029a00)={0x84,
↪  &(0x7f00000299c0)={0x20, 0x0, 0x0, 0x0, 0x18, '\x00'/24}, 0x0,
↪  0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
↪  0x0}) +2066us
T1->syz_usb_ep_write(r0, 0x81, 0x6,
↪  &(0x7f0000029ac0)="0e04010c2000") +450us
T2->syz_usb_control_io(r0, 0x0, &(0x7f0000029b40)={0x84,
↪  &(0x7f0000029b00)={0x20, 0x0, 0x0, 0x0, 0x18, '\x00'/24}, 0x0,
↪  0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
↪  0x0}) +503us
T7->syz_usb_ep_write(r0, 0x81, 0x6,
↪  &(0x7f0000029c00)="0f0400010104") +509455us
```

Figure 4: An example two-dimensional trace program.

action at the time of receipt from the host side.

Annotating Actions. During compilation, ReUSB annotates each action using three constructs we introduce in the input program description language, as follows.

- **Execution Context:** Each action is annotated with its *execution context* logged while recording. This construct is the key to solving problems caused by the concurrent, non-deterministic nature of drivers executions: By switching between execution contexts during replay, ReUSB can react to diverging interleavings of unordered USB requests.
- **Dispatch Mode:** Each action is assigned a *dispatch mode* of either asynchronous or synchronous; the dispatch mode of an action is set asynchronous, if the duration of the action completely overlaps with any action subsequently invoked from a different execution context but returned earlier while recording. ReUSB uses the dispatch mode to control the concurrency between execution contexts during replay.
- **Post-Dispatch Delay:** Each action is annotated with a *post-dispatch delay*, and ReUSB uses, as a post-dispatch delay of an action, the duration from its return to the invocation of its subsequent action; ReUSB enforces this delay between action dispatches during replay, to account for temporal constraints that require delays.

These annotations can be seen in Figure 4, which shows an excerpt of the program compiled from two-dimensional traces of a Wi-Fi driver. Observe that all actions begin with their execution context (e.g., T1 and T2), and end with a potential delay in realizing their effect; asynchronously dispatched actions are marked with a symbol & after the call (e.g., poll); actions without this symbol are synchronously dispatched.

4.3 Bytecode Generation

To execute a given input program P, it must be converted into its bytecode representation. To facilitate context switches at run time, ReUSB performs *per-context* bytecode generation,

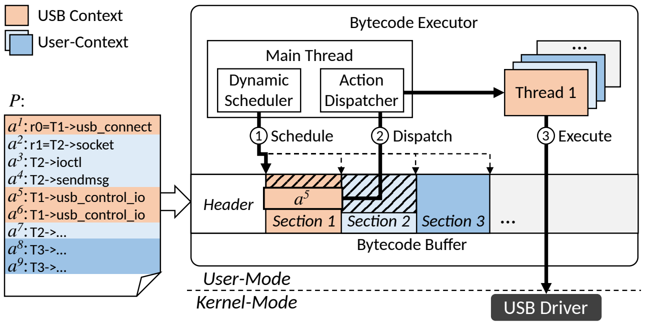


Figure 5: Per-context bytecode generation and execution.

as illustrated in Figure 5: ReUSB splits the bytecode buffer into *sections*, and assigns each execution context found in P a unique section in the bytecode buffer. All actions recorded in each context are serialized, in their bytecode form, into the section assigned to that context. During bytecode generation, each action's dispatch mode (of either asynchronous and synchronous) and delay are also serialized into each action's bytecode representation, which are read by ReUSB's bytecode executor for a controlled action dispatch. This per-context organization of the bytecode buffer facilitates scheduling of the execution contexts at run time, and switching between them.

4.4 Bytecode Execution

ReUSB's bytecode executor is designed as a user-mode interpreter, which interprets and executes the bytecode representation of P. As illustrated in Figure 5, the executor repeatedly ① schedules an execution context, ② dispatches the action found at the top of the scheduled context to the thread assigned to that context, and ③ executes the action from the assigned thread. It is worth noting that, besides system call actions $a_{syscall}$, USB response actions a_{usb} are also executed by this user-mode executor via user-mode USB device emulation. Unlike prior work that uses synchronous and sequential replay of recorded actions, ReUSB's bytecode executor performs time-, concurrency-, and context-aware replay to enhance the fidelity of replay (i.e., G1), as follows.

Time- and Concurrency-Aware Dispatch. Recall that executing a given input program either synchronously or asynchronously without careful timing and concurrency control over action executions may cause the program (i) simply fail to reproduce the recorded execution, or (ii) be blocked indefinitely, due to unsatisfied temporal constraints (see C1). ReUSB addresses these two problems by using the post-dispatch delay and dispatch mode annotations of each action, whose values are set such that ReUSB reproduces, during replay, *timing and concurrency characteristics* similar to the ones observed while recording. Using these annotations, ReUSB's interpreter dispatches an action for execution (i) only after the post-dispatch delay of its preceding action to satisfy delay constraints, and (ii) asynchronously, i.e,. invoking its subsequent actions in other execution contexts immediately without waiting for its return, to satisfy concurrency constraints between actions that could block the execution when unsatisfied. Note that actions in the same execution contexts are always invoked synchronously.

Context-Aware Dynamic Scheduling. Recall that drivers have multiple concurrently running execution contexts, which permits, during replay, them to make USB requests in an order different from the order observed while recording (see C2). ReUSB reacts to the diverging interleavings of such *unordered* USB requests as they occur during replay, by reordering input actions. To this end, we use the following *context-aware* dynamic scheduling policy. At each scheduling point at run time, when (i) there is a USB request made by the driver, and (ii) the request can be served by USB response actions a_{usb} pending in more than one execution contexts, ReUSB inspects the *context* of the current request (i.e., the content and the execution context of the current as well as prior requests), and then dynamically schedules the execution context whose pending response, *contextually*, best matches the current request. If this policy fails to find a matching response for the request, ReUSB falls back to the scheduling policy that follows the chronological order in which actions were observed while recording. A context switch occurs when the newly scheduled context differs from the current one.

4.5 Replay-Guided Fuzzing

The key premise of ReUSB is that, from the *normal* code paths exercised through record-and-replay, many new neighboring yet deep code paths can further be exercised when combined with fuzzing. Using record-and-replay in fuzzing, however, comes at the expense of an enlarged mutation space, increased overheads caused by prolonged replay, and side effects caused by realistic replay. We address these problems by employing a principle dubbed *replay-guided fuzzing*. We realize this principle in ReUSB by employing our own (i) mutational fuzzing algorithm, (ii) checkpointing algorithm, and (iii) the program generation and execution constraints, proposed below.

Replay-Guided Fuzzing. We first propose a new mutational fuzzing algorithm, dubbed replay-guided fuzzing, which *strictly* guides fuzzing along the replay. This algorithm deviates from *evolutionary* fuzzing algorithms used by most of state-of-the-art fuzzers, in the following respects. First, while existing fuzzers select, in each fuzzing iteration, a program from the corpus for mutation, typically prioritizing the ones with high coverage yet small execution time and program size, ReUSB always selects one among trace programs to explicitly prioritize the depth of exploration over breadth. Second, ReUSB fuzzes the selected ones with small-scope mutations only—insert, mutate, and delete—whereas existing fuzzers frequently generate programs which substantially differ from the original, e.g., Syzkaller's generating random programs or concatenating completely different programs [29]. Third, existing fuzzers heavily focus on evolving the corpus into a set of deterministic and small programs through aggressive pruning and minimization; that is, when they find an input program that leads to new coverage, they add it to the corpus only if the coverage is stable, and only after exhaustively minimizing it. ReUSB, by contrast, does not minimize programs, because such minimization attempts cause a prohibitive overhead for lengthy trace programs, and likely fail due to prevalent ordering dependencies in them. These principled deviations effectively make ReUSB guide USB driver fuzzing along the replay of recorded executions of the drivers, and, therefore, towards their deeper code paths and vulnerabilities.

Replay Checkpointing. To reduce the impact of faithful replay on the overall fuzzing speed and persisting side effects (i.e., $\boxed{\text{G2}}$), we employ VM checkpoint-and-restore in ReUSB, following prior work [61, 64]. Our contribution here is a new checkpointing policy. The checkpointing policies proposed and used by prior work [61, 64] are not optimal for fuzzing drivers with lengthy traces of recorded executions. We propose a new checkpointing policy dubbed *replay checkpointing*: This policy performs checkpointing (i) only during replay runs, i.e., while executing trace programs as-is without mutating them, and (ii) at small, fixed intervals. In contrast to prior work that may checkpoint any execution producing new coverage [61, 64], our policy strictly bounds the number of checkpoints created during fuzzing, which has the following benefits. First, it reduces the run-time and memory overheads associated with checkpoint creation. Second, with enough checkpoint storage, it obviates the need for checkpoint replacement, eliminating potential thrashing problems that the checkpointing policy used by prior work could face [64].

Relaxing Constraints. An additional benefit of ReUSB's using a VM checkpoint-and-restore mechanism is that various constraints imposed by existing fuzzers on both program generation and execution can be relaxed in support of more realistic replay. As described in §3.2, the contraints employed by state-of-the-art fuzzers are too strict to exercise deep code paths, and to find vulnerabilities in their neighboring paths.

- **Program Size and Execution Time:** ReUSB uses significantly looser program size and execution time constraints than existing fuzzers. This could inadvertently result in (i) a lower fuzzing speed (see C3), and (ii) a bloated search space for mutational fuzzing. ReUSB mitigates the former by accelerating fuzzing via replay checkpointing, and the latter by focusing fuzzing on trace programs.
- **Program Behavior:** ReUSB additionally lifts the sandboxing policies employed by existing fuzzers. Each input program execution, whose behavior is not constrained by sandboxes, can produce system-wide side effects that persist its lifetime (see C4). ReUSB eliminates such side effects, by always restoring the entire VM back to a clean, known state from a checkpoint after executing each program.

Table 2: A list of USB devices and their corresponding drivers.

Class	Vendor	Device	Driver Source Code (drivers/...)
Wi-Fi	Broadcom	BCM43236	net/wireless/broadcom/brcm80211
	Qualcomm	AR9271	net/wireless/ath/ath9k
	Ralink	RT5370	net/wireless/ralink/rt2x00
	Realtek	RTL8812BU	github.com/morrownr/88x2bu-20210702*
		RTL8821AU	github.com/aircrack-ng/rtl8812au*
	Mediatek	MT7601U	net/wireless/mediatek/mt7601u
		MT7610U	net/wireless/mediatek/mt76/mt76x0
Bluetooth	Broadcom	BCM20702	bluetooth/btbcm.c
	CSR	CSR8510	bluetooth/btusb.c
NFC	NXP	PN533	nfc/pn533

*Out-of-tree drivers whose source code is available at the shown URL.

5 Implementation

Recording. We implemented our execution recording environment using a VM hosted by QEMU 4.0.0 [10], and accelerated with Linux KVM [52]. We used an XHCI USB controller available in QEMU [71] as our virtual peripheral bus to which physical USB devices to record were forwarded. We created a hypercall for hot-plugging the USB device to the virtual bus, and another for unplugging it, and used these hypercalls to control recording entirely from user space within the guest VM. We recorded the entire driver executions from the moment the device starts to interact with the kernel. Specifically, we let the driver operate normally with user-mode programs and the hot-plugged device, during which we recorded (i) system calls using STRACE [4], and (ii) USB messages using Wireshark [5] and USBMON [70]. To compile raw STRACE traces and Wireshark PCAP [3] traces into a trace program we used an improved version of TRACE2SYZ [53], and a tool dubbed PCAP2SYZ that we authored, respectively.

Replay and Fuzzing. We based our implementation of USB driver replay and fuzzing on Syzkaller [29]. We extended Syzkaller's bytecode instructions with multiple new instructions, and modified Syzkaller's bytecode interpreter to support our time-, concurrency-, and context-aware replay. We implemented our replay-guided fuzzing by modifying the fuzzing algorithm of Syzkaller. We implemented our replay checkpointing based on Agamotto [64] such that ReUSB periodically checkpoints the VM during replay, and restores it to a checkpoint before executing each program. Like Syzkaller [28], our interpreter also uses USB devices emulated in the user space (via Linux's raw gadget [41]) as a mechanism to replay and fuzz the USB interface.

6 Evaluation

Target Drivers. Table 2 shows our evaluation targets: 10 *wireless* USB drivers of 3 classes—Wi-Fi, Bluetooth, and Near-Field Communication (NFC)—found either in the Linux kernel v5.14 or in the latest out-of-tree drivers at the time of writing. We chose these drivers, because they (i) implement inherently stateful wireless networking protocols, (ii) expose an additional wireless attack surface, and (iii) operate *real* USB devices available on the market at an affordable price. That is, they are challenging fuzzing targets, and vulnerabilities, when found, can have a significant real-world impact.

Table 3: Wireless networking configurations instantiated in our dual-VM driver execution recording environment.

	Configuration	VM Ⓐ	VM Ⓑ
Wi-Fi	Client & AP (§A.1)	Client station	Access point
Bluetooth	Peer-to-Peer (§A.2)	Observer (Master)	Advertiser (Slave)
NFC	Peer-to-Peer (§A.3)	Initiator (Master)	Target (Slave)

Table 4: A list of generated traces in each VM per configuration. All the listed devices were traced twice, once in each VM, except for MT7601U, as it does not support an AP mode.

	VM Ⓐ		VM Ⓑ		Duration (sec.)
	Device	# of Actions (Syscall/USB)	Device	# of Actions (Syscall/USB)	
Wi-Fi	BCM43236	**1,894** (1,495/ 399)	MT7610U	-	20
	AR9271	**8,143** (1,577/ 6,566)	MT7610U	-	19
	RT5370	**6,311** (1,568/ 4,743)	MT7610U	-	19
	RTL8812BU	**24,529** (2,761/21,768)	MT7610U	-	23
	RTL8821AU	**9,328** (2,550/ 6,778)	MT7610U	-	23
	MT7601U	**4,099** (1,494/ 2,605)	MT7610U	**9,489** (2,047/ 7,442)	20
	MT7610U	**15,011** (2,639/12,372)	BCM43236	**1,484** (1,051/ 433)	20
	MT7610U	-	AR9271	**11,094** (2,600/ 8,494)	24
	MT7610U	-	RT5370	**11,272** (2,244/ 9,028)	22
	MT7610U	-	RTL8812BU	**12,577** (1,104/11,473)	21
	MT7610U	-	RTL8821AU	**6,728** (1,104/ 5,624)	19
Bluetooth	BCM20702	**6,108** (3,866/ 2,242)	CSR8510	**1,219** (1,037/ 182)	21
	CSR8510	**9,219** (6,423/ 2,796)	BCM20702	**2,004** (1,035/ 969)	24
NFC	PN533	**475** (437/ 38)	PN533	**528** (484/ 44)	4

6.1 Recorded Executions

Dual-VM Recording Environment. To record executions of the wireless USB drivers targeted in our evaluation, we created a *dual-VM* recording environment, which comprises *two* VMs configured to *wirelessly* communicate with each other on a *single* host: Each VM runs a full wireless networking software stack including the target driver that corresponds to the physical USB device forwarded to it, and the two VMs wirelessly communicate with each other through these devices. We recorded the target wireless driver's execution in each VM, using the mechanisms described in §5.

Wireless Networking Configurations. We wrote pairs of configuration scripts, where each pair instantiates a typical wireless communication scenario, shown in Table 3, between two VMs in our recording environment. That is, in each configuration, *two* devices of the same kind, either Wi-Fi, Bluetooth, or NFC, are attached to a single host, and forwarded to different VMs (i.e., VM Ⓐ and VM Ⓑ). The script running in each VM then (i) configures the forwarded device and the corresponding driver, and (ii) drives wireless communication with the other VM. We provide further details about the recorded wireless communication scenarios in Appendix A.

Generated Traces. Using our configuration scripts, we gener-

Table 5: The fidelity of two-dimensional record-and-replay measured in basic block coverage. The percentages enclosed in the parentheses denote the ratio of the coverage of replay runs w.r.t. the original recording run. TCAD refers to "Time- and Concurrency-Aware Dispatch", and CADS refers to "Context-Aware Dynamic Scheduling".

		Record	Replay		
			Baseline	After TCAD	After CADS
BCM43236	Client	3,533	1,346 (38.1%)	**2,507 (71.0%)**	**3,441 (97.4%)**
	AP	3,205	1,295 (40.4%)	**1,440 (44.9%)**	**2,827 (88.2%)**
AR9271	Client	4,956	1,709 (34.5%)	**2,808 (56.7%)**	2,808 (56.7%)
	AP	4,311	1,447 (33.6%)	**2,721 (63.1%)**	2,721 (63.1%)
RT5370	Client	4,232	1,269 (30.0%)	**1,832 (43.3%)**	**1,933 (45.7%)**
	AP	3,724	1,005 (27.0%)	**2,073 (55.7%)**	**2,831 (76.0%)**
RTL8812BU	Client	11,440	3,852 (33.7%)	**6,716 (58.7%)**	6,716 (58.7%)
	AP	10,508	3,835 (36.5%)	**7,676 (73.0%)**	7,676 (73.0%)
RTL8821AU	Client	7,381	2,434 (33.0%)	**3,809 (51.6%)**	**3,821 (51.8%)**
	AP	6,945	2,115 (30.5%)	**3,544 (51.0%)**	**3,561 (51.3%)**
MT7601U	Client	3,492	1,384 (39.6%)	**1,824 (52.2%)**	**1,853 (53.1%)**
MT7610U	Client	4,458	1,629 (36.5%)	**1,762 (39.5%)**	**2,265 (50.8%)**
	AP	3,976	1,326 (33.4%)	**1,437 (36.1%)**	**2,240 (56.3%)**
BCM20702	Master	2,921	2,302 (78.8%)	2,302 (78.8%)	2,302 (78.8%)
	Slave	2,683	2,428 (90.5%)	**2,634 (98.2%)**	2,634 (98.2%)
CSR8510	Master	2,872	1,400 (48.7%)	**2,297 (80.0%)**	2,297 (80.0%)
	Slave	2,598	2,566 (98.8%)	2,566 (98.8%)	2,566 (98.8%)
PN533	Master	700	326 (46.6%)	**677 (96.7%)**	677 (96.7%)
	Slave	631	321 (50.9%)	**607 (96.2%)**	607 (96.2%)
Geometric mean			42.0%	**62.5%**	**69.7%**

ated 19 unique traces using a combination of 10 USB devices, which are summarized in Table 4. Even though we recorded driver executions under typical wireless communication scenarios whose duration is mostly less than a minute (measured from the invocation of the first recorded action to the last), the number of recorded actions mostly exceeds thousands. These actions have many ordering dependencies reflecting their statefulness (see 3.1), which, when unresolved, make replay fail to reproduce the recorded driver execution.

6.2 Record-and-Replay Fidelity

Using 19 trace programs created using our dual-VM recording setup (see §6.1), we first thoroughly examine the fidelity of ReUSB's record-and-replay. We measure the fidelity using the *stable* basic block coverage obtained as follows. We performed, for each trace program, 3 independent replay runs, and conservatively counted the resulting coverage that were (i) invariant across these runs, and (ii) also found in the original record runs. We only counted the coverage of the target drivers and the common networking code shared by them by limiting coverage instrumentation to their source code.

Experimental Setup. We used Syzkaller's executor as a baseline (referred to as Baseline), which is used by most of prior work. Baseline synchronously and sequentially executes actions found in a trace program (see §2.3), without ReUSB's time-, concurrency-, context-aware replay. Starting from this baseline, we *incrementally* applied the techniques we proposed for faithful driver execution replay: our time-and concurrency-aware dispatch (TCAD), and context-aware dynamic scheduling (CADS). For a conservative evaluation, the constraints imposed by Syzkaller were relaxed in all configurations including Baseline. The results are shown in Table 5.

Table 6: Basic block coverage achieved through one- vs. two-dimensional replay. The percentages enclosed in the parentheses denote the ratio w.r.t. the two-dimensional one.

		System Call Only [53]	USB Message Only [56]	**Two-Dimensional**
BCM43236	Client	136 (4.0%)	1,280 (37.2%)	3,441
	AP	152 (5.4%)	1,278 (45.2%)	2,827
AR9271	Client	136 (4.8%)	1,401 (49.9%)	2,808
	AP	152 (5.6%)	1,417 (52.1%)	2,721
RT5370	Client	136 (7.0%)	543 (28.1%)	1,933
	AP	152 (5.4%)	968 (34.2%)	2,831
RTL8812BU	Client	136 (2.0%)	3,819 (56.9%)	6,716
	AP	152 (2.0%)	3,835 (50.0%)	7,676
RTL8821AU	Client	136 (3.6%)	2,058 (53.9%)	3,821
	AP	152 (4.3%)	2,074 (58.2%)	3,561
MT7601U	Client	136 (7.3%)	1,098 (59.3%)	1,853
MT7610U	Client	136 (6.0%)	1,304 (57.6%)	2,265
	AP	152 (6.8%)	1,320 (58.9%)	2,240
BCM20702	Master	278 (12.1%)	670 (29.1%)	2,302
	Slave	278 (10.6%)	670 (25.4%)	2,634
CSR8510	Master	278 (12.1%)	605 (26.3%)	2,297
	Slave	278 (10.8%)	602 (23.5%)	2,566
PN533	Master	30 (4.4%)	163 (24.1%)	677
	Slave	26 (4.3%)	163 (26.9%)	607
Geometric mean		5.5%	39.6%	

Time- and Concurrency-Aware Dispatch. Most of the target drivers of our evaluation benefited from our proposed time- and concurrency-aware dispatch (TCAD) during two-dimensional replay. Quantitatively, ReUSB's TCAD improves the coverage of the driver source code from 42.0% to 62.5% on average (geometric mean). This means (i) that there are indeed many temporal constraints that require actions be invoked in a delayed manner or concurrently while previously invoked actions are executing (see §3.1), and (ii) that ReUSB's TCAD can help better satisfy such temporal constraints.

Context-Aware Dynamic Scheduling. Although the effectivenss of our context-aware dynamic scheduling (CADS) is not as dramatic as TCAD, it does increase the coverage of the driver source code, from 62.5% to 69.7%. In particular, ReUSB's CADS significantly improves the coverage of BCM43236 and MT7610U over TCAD that uses sequential execution. These two drivers benefited most from ReUSB's CADS because of their high degree of concurrency support; that is, there are many concurrently running code paths in these drivers making USB requests without ordering them, which effectively allows the requests to interleave differently with each other between record and replay runs.

Two-Dimensional Record-and-Replay. We now ablate all of either system calls or USB messages from trace programs to quantify the effectiveness of *two-dimensional* record-and-replay relative to one-dimensional ones. All of the techniques we proposed were applied in these experiments. The one-dimensional baselines could be thought of as *stronger* ver-

Table 7: Errors hit while fuzzing the target drivers using our proposed techniques.

Device	Role(s)	Error Type	Prev. Unknown	Upstream Patch	Buggy Kernel Code Loc.
BCM43236	Client&AP	Slab out-of-bounds	✓	4920ab1	Driver
	Client&AP	Slab out-of-bounds	✓	4920ab1	Driver
	Client&AP	Stack out-of-bounds	✓	0a06cad	Driver
	Client&AP	Stack out-of-bounds	✓	660145d	Driver
	Client&AP	Null pointer deref.	✓	683b972	Driver
	Client&AP	Shift out-of-bounds	✓	81d17f6	Driver
	Client&AP	Slab out-of-bounds	✓[1]	6788ba8	Driver
	Client	Slab out-of-bounds	✓[2]	0da40e0	Driver
AR9271	Client&AP	Stack out-of-bounds	✓	8a2f35b	Driver
	Client&AP	Null pointer deref.	✗[3]	-	Driver
	Client&AP	Null pointer deref.	✗[3]	-	Driver
	Client&AP	Use-after-free	✓	f099c5c	Driver
	AP	Divide-by-zero	✗[3]	-	Driver
MT7610U	Client&AP	Null pointer deref.	✓	bd5dac7	Driver
MT7601U	Client	Null pointer deref.	✓	803f317	Driver
CSR8510	Master	Slab out-of-bounds	✗[4]	-	BT subsystem
	Master	Corrupted `list`	✗[4]	-	BT subsystem
PN533	Master	Slab out-of-bounds	✓	9f28157	Driver
	Master&Slave	Use-after-free	✓	9dab880	Driver
	Slave	Use-after-free	✓	4bb4db7	NFC subsystem

[1]Assigned CVE-2022-3628. [2]Assigned CVE-2023-1380.
[3]Previously discovered by Syzbot [78], using manually-written system call descriptions tailored to AR9271's driver.
[4]Previously discovered by Syzbot [78], using a custom harness and manually-written system call descriptions.

sions of MoonShine [53] (for syscall only) and USBFuzz [56] (for USB message only). The results are shown in Table 6, which highlights the importance of two-dimensionality; without replaying system calls, only 39.6% of basic blocks, on average (geometric mean), were covered; without USB messages, only 5.5% of basic blocks were covered on average.

6.3 Fuzzing Effectiveness

We now quantify the effect of our proposed techniques on USB driver fuzzing, conservatively, over multiple strong baselines. To this end, we conducted a series of mutational fuzzing experiments, using, as an initial seed corpus, the trace programs obtained via our execution recording described in §6.1. Following the guidance provided by prior work [40, 49], we use the number of bugs found as the primary metric of our evaluation, and employ code coverage metrics as necessary for demonstrating the effect of ReUSB's individual techniques.

Experimental Setup. We configured ReUSB considering *realistic* threats posed by adversaries on the peripheral side of USB drivers' trust boundary, meaning that we let the fuzzer fuzz only USB response actions invoked through the USB interface. That said, if one wishes to find vulnerabilities under a different threat model, the mutation scope can be confined differently, e.g., confined to system calls when considering threats by local adversaries. We performed parallel fuzzing using 32 instances of the fuzzer. To detect manifestations of bugs as they are triggered by ReUSB, we used Kernel AddressSanitizer [42] and UndefinedBehaviorSanitizer [59].

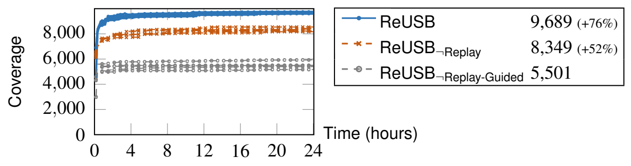
Table 8: Constraint relaxations accomodated by ReUSB.

	Syzkaller [29]'s Default	**Relaxed to ...**
Max. # of actions in P (Prog. Size)	20	**12,000**
Timeout for executing P (Prog. Exec. Time)	5 seconds	**300 seconds**
Sandboxes for P's exec. (Prog. Behavior)	Multiple namespaces	**No namespace**

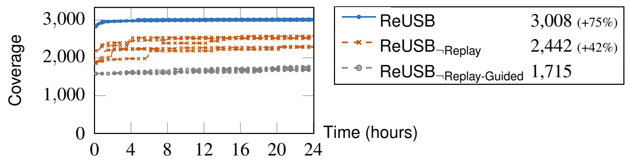
Results. While fuzzing the target drivers using ReUSB, we found 20 bugs, summarized in Table 7, of which 15 were previously unknown. Many of them were found in the mainline kernel, even though the mainline USB drivers have been heavily fuzzed by Google's Syzbot [78]. Bugs were found in a variety of components: the target drivers themselves and the Linux's Bluetooth [15] and NFC [68] subsystems, demonstrating that ReUSB's replay-guided fuzzing with our enhanced record-and-replay is effective at finding bugs in wireless networking drivers and the common networking code shared by them, which are among the most stateful components in the kernel. Of the *known* bugs, two in the Linux Bluetooth subsystem (not in the individual drivers) were previously discovered by Syzbot [78]. Our inspection of Syzbot's reports for these bugs revealed that Syzbot triggered these bugs by injecting malformed packets *directly into* Linux's Bluetooth subsystem, using (i) a Bluetooth-specific packet injection mechanism, and (ii) system call descriptions manually written to use that custom injection mechanism. By contrast, ReUSB triggered these bugs by merely interacting with the driver at their trust boundary, without requiring human experts to write any harness, nor custom system call descriptions.

Bug Finding Capability. The key component of ReUSB that enabled the discovery of these previously unknown bugs is replay-guided fuzzing (§4.5). We were not able to discover any of these bugs with an unmodified Syzkaller, given (i) the same number of fuzzer instances, (ii) the same amount of fuzzing time, and (iii) the same initial seed corpus, for the following reasons. First, Syzkaller constrains the execution time, size, and behavior of a program to an extent that it cannot reach driver code paths beyond initialization, when most of the bugs found were triggered during or after a successful initialization. Second, Syzkaller exhaustively minimizes any program that finds new coverage, which led to (i) a prohibitive overhead without much gain (i.e., most of minimization attempts failed due to prevalent ordering dependencies), and (ii) inadvertently discarding important actions when Syzkaller fails to faithfully replay them. ReUSB, in contrast, did not suffer from these problems, as it used (i) looser constraints as described in Table 8, (ii) trace programs as-is without minimizing them, and (iii) high-fidelity replay.

Code Coverage. We now study ReUSB's capability in reaching deeper code paths in the target drivers. To this end, we conducted two controlled fuzzing experiments by using, as an initial seed corpus, (i) all of our 19 trace programs, and (ii) a single trace program generated with BCM43236 in its client mode, whose driver had the highest number of bugs



(a) Using all of our 19 trace programs as an initial seed corpus.



(b) Using BCM43236's client mode trace program as an initial seed corpus.

Figure 6: Basic block coverage measured while fuzzing the drivers with different initial seed corpora. The plots depict the results of 5 independent runs, and the numbers in the legends are the final coverage results averaged over the runs.

Table 9: The amount of VM execution time saved via replay checkpointing. The percentages enclosed in the parentheses denote the ratio w.r.t. the total amount of VM time that would have taken if there were no replay checkpointing support.

	# of Execs.	VM Exec. Time	Throughput	VM Time Saved
ReUSB	262091.00	992.61 hrs	4.40/min	**279.42 hrs (28.15%)**
ReUSB$_{\neg\text{Replay-Guided}}$	514047.40	757.94 hrs	**11.30/min**	19.52 hrs (2.58%)

(see Table 7). Figure 6 depicts the coverage results of these two experiments. Like prior experiments, we only counted the coverage of the target drivers and the common networking code shared by them. To individually evaluate ReUSB's techniques, we used the following configurations: (i) ReUSB, (ii) ReUSB$_{\neg\text{Replay}}$ that ablates time-, concurrency-, and context-aware replay from ReUSB, and (iii) ReUSB$_{\neg\text{Replay-Guided}}$ that *additionally* ablates ReUSB's replay-guided mutational fuzzing algorithm, and, instead, uses Syzkaller's evolutionary one. In both experiments, ReUSB$_{\neg\text{Replay-Guided}}$ achieves the lowest coverage. A primary reason is that actions not faithfully replayed are simply considered having no effect and, therefore, discarded during Syzkaller's minimization process. The coverage improves after applying ReUSB's fuzzing algorithm (see ReUSB$_{\neg\text{Replay}}$), which further improves after applying ReUSB's high-fidelity replay techniques (ReUSB).

Fuzzing Throughput. A major drawback of combining faithful replay with fuzzing is that it necessarily increases the execution time of each program, which, in turn, could reduce the overall fuzzing throughput. We proposed replay checkpointing to mitigate the effects of prolonged replay of lengthy trace programs, and, here, we quantify its effectiveness by measuring the total amount of VM time saved via checkpoint restoration during the ReUSB and ReUSB$_{\neg\text{Replay-Guided}}$ fuzzing runs using all of our 19 trace programs as an initial seed corpus. Table 9 summarizes the results; we can

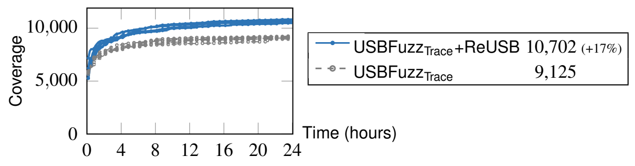


Figure 7: AFL-style edge coverage achieved by USBFuzz with and without ReUSB's accurate replay. The plots depict the results of 5 independent runs, and the numbers in the legends are the final coverage results averaged over the runs.

observe that ReUSB's replay-guided fuzzing benefits significantly more from replay checkpointing than Syzkaller's evolutionary fuzzing, saving 28.15% of the VM execution time replay-guided fuzzing could have spent in total without replay checkpointing (see "VM Time Saved"). Although the fuzzing throughput of replay-guided fuzzing is lower than Syzkaller's evolutionary fuzzing, we argue that the benefit (e.g., 76% coverage increase Figure 6a) outweights the cost.

USBFuzz with Accurate Replay. We further show the benefit of accurate replay on USB driver fuzzing with USBFuzz [56]. To ensure a conservative comparison, we substantially modified USBFuzz's implementation so that it can leverage our BCM43236's client mode trace. In particular, we added support for invoking system calls in USBFuzz, by extracting the system calls in the trace and invoking them from USBFuzz's user agent. We also translated the USB messages in the trace into a format that can be consumed by AFL [82], USBFuzz's fuzzing engine. Using this trace-enhanced configuration of USBFuzz as the baseline (denoted by USBFuzz$_{\text{Trace}}$), we created a *replay-enhanced* configuration of USBFuzz that incorporates ReUSB's accurate replay atop (USBFuzz$_{\text{Trace}}$+ReUSB). Instead of implementing a native replay support in USBFuzz, which requires all of our engineering efforts be duplicated for USBFuzz, we chose to *emulate* this configuration by leveraging ReUSB's replay engine. Specifically, when USBFuzz generates an input whose prefix matches a prefix of the trace, we let (i) ReUSB's replay engine replay the longest matching prefix first, and then (ii) USBFuzz consume the rest. Figure 7 depicts the results. Replay-enhanced USBFuzz outperforms the trace-enhanced one (by 17% on average), demonstrating that USBFuzz can benefit from accurate replay as well, even without some optimizations we proposed such as replay checkpointing.

6.4 Analysis of Bugs Found

We draw further insights from the bugs we found with ReUSB by examining their attack vectors, causes, and manifestations.

Attack Vector Analysis. The key benefit of defining record-and-replay as well as fuzzing at the trust boundary is that the bugs, when found, constitute direct security threats posed by adversaries facing the boundary (see §2.3). In an effort to find bugs that could be triggered in real USB attack scenarios, we targeted the peripheral attack surface of USB drivers by operating under a threat model where adversaries are on the device side who only control USB responses, and not system calls (see §6.3). Observe that, even though the adversaries cannot directly control system calls, they can still induce normal system call activities by impersonating a normal device, until they are able to trigger the target bug by sending a crafted USB message. Most of the bugs found by ReUSB can be triggered by inserting or removing only a single USB response from recorded traces, which means that they can indeed be triggered by adversaries who only control the peripheral side.

Root Cause Analysis. The majority of bugs were caused by failing to sanitize device-provided values, though they must not be trusted in many real-world USB attack scenarios. ReUSB triggered this class of bugs by sending a USB message that contains values unexpected by the driver. Another cause of the bugs was failing to consider all possible orders in which actions can be invoked. ReUSB triggered this class of bugs by sending a USB message at a point unexpected by the driver. For example, a bug found in the NFC subsystem was caused by failing to coordinate actions that reference the same object, when invoked in an order different from recorded traces: an object deallocated by a USB disconnect action was later referenced by system calls through a dangling pointer, causing a use-after-free error. Yet another cause of the bugs was failing to handle missing USB responses: ReUSB triggered this class of bugs by not sending a USB message expected by the driver, e.g., the one found in MT7610U. We identified other causes of bugs (i.e., other programming errors) as well. For instance, a previously unknown use-after-free bug found in AR9271 was caused by incorrectly ordered deallocations of cross-referencing objects. This bug had been lurking undetected so far before ReUSB found it, simply because existing fuzzers had never reached the erroneous code.

Memory Safety Violations. A bug in the driver may manifest, perhaps in the most dangerous way, as a memory safety violation. ReUSB found many such dangerous bugs that manifested as either spatial or temporal memory safety violations. Some bugs manifested as slab or stack out-of-bounds errors, either when device-provided values were used in pointer arithmetic operations without sanitization, or when device-provided strings were used by the driver to call string manipulation functions without checking whether a null character exists at their end. Several other bugs manifested as use-after-free errors, whose root cause is either missing or incorrect ordering between use and free operations, as they were triggered by sending USB messages in an unexpected order.

Other Safety Violations. The bugs triggered by ReUSB manifested as other safety violations as well. For example, a bug was triggered, as a driver used malformed values as an argument of Linux's socket buffer API call whose semantics is to return a null pointer when unexpected values are provided. The driver then dereferenced this null pointer without any check, causing a null-pointer dereference error. Another interesting bug was found in AR9271, which uses a value received from the device as the denominator of a division operation without checking, leading to a division-by-zero error.

7 Discussion & Limitations

Improving Replay Fidelity. Further improving semantic fidelity could potentially uncover more bugs during replay-guided fuzzing. Future work could explore different designs than ours, such as a design with a more intrusive record-and-replay, which can increase the semantic fidelity. Future work could also explore designs that can achieve higher performance fidelity, which could potentially help detect concurrency bugs, especially when combined with dynamic concurrency bug detection tools such as KCSAN [21].

Diversifying Recording Scenarios. We evaluated ReUSB with multiple classes of devices, but, for a conservative evaluation, only with a limited number of trace programs recorded under standard wireless communication scenarios (see §A). ReUSB could potentially uncover more bugs with driver executions recorded under a more diverse set of usage scenarios. USB drivers other than wireless networking ones could also be recorded during their normal operation, and fuzzed with ReUSB to uncover bugs in them. We intend to expand the corpus of traces both in variety and in size in our future work.

Supporting Other OSs. ReUSB's record-and-replay, as it is defined at the trust boundary of USB drivers, requires interposing on both system calls and USB messages. Interposing on system calls can entirely be done in userspace, whereas doing so for USB messsages requires support from the kernel. Although we only implemented ReUSB for Linux using USBMon [70] and USB raw gadget [41], we believe that the ReUSB design can be applied to closed-source OSs such as Windows, because interposing on the USB layer (and not individual drivers) can be done by using USBPcap [2] or by writing USB Device Emulation (UDE) drivers [6].

Supporting Other Peripheral Buses. We designed ReUSB for fuzzing USB drivers, but our replay techniques could be applied when fuzzing drivers that use other peripheral buses such as PCI or I2C. We acknowledge, however, that the I/O interception mechanism of record-and-replay needs changing. A recent study demonstrates that intercepting I/O of a DMA-capable PCI device is feasible [46]. We intend to extend ReUSB for other peripheral buses in our future work.

8 Related Work

Finding Bugs in Device Drivers. Researchers have proposed a plethora of techniques to find bugs in OS kernels, which include static [45, 58, 80], dynamic [18, 29, 32, 34, 47, 57, 63, 64, 67, 81], and hybrid [22, 37, 38, 53, 62, 83] analysis.

Many [57, 62, 67], like ReUSB, enhance dynamic analysis of device drivers. For instance, EASIER replaces I/O with stub functions [57], Charm relays I/O to remote devices [67], and Drifuzz uses symbolic I/O [62], in an effort to enhance driver fuzzing with minimal or no hardware requirement. ReUSB, using a different record-and-replay approach that requires hardware only while recording, also facilitates driver fuzzing.

Defending against USB Attacks. The USB interface has been known notoriously for its unique attack surface and insecurity [75]. Unfortunately, their attack surface is still expanding; for example, with an increasing demand for work-from-home, the host-side USB stack is nowadays getting exposed to devices attached over network via USB-over-Ethernet/IP [69]. Researchers have proposed (i) fine-grained access control mechanisms to defend against attacks [72–74], and (ii) analysis techniques (e.g., using custom hardware [1] and hybrid fuzzing [38]) to find vulnerabilities in the USB software stack. ReUSB's replay and fuzzing techniques also enhance dynamic analysis of the host-side USB software stack.

9 Conclusion

We presented ReUSB, a replay-guided USB driver fuzzer design that can significantly enhance the effectiveness of USB driver fuzzing. The guiding principles of our ReUSB design are non-intrusive yet high-fidelity record-and-replay, as well as high-speed, clean-state fuzzing. ReUSB realizes (i) non-intrusive yet faithful replay with new language-level constructs that enable time-, concurrency-, and context-aware replay of recorded executions, and (ii) high-speed, clean-state fuzzing with replay-guided mutational fuzzing accelerated by replay checkpointing. These techniques allow ReUSB to fuzz deep driver code paths at a high speed, leveraging trace programs consisting of thousands of system calls and USB messages. We evaluated ReUSB by fuzzing 10 USB drivers in Linux, where we found 15 bugs confirmed to be previously unknown, and increased their code coverage by 76%.

Acknowledgments

The authors would like to thank the anonymous shepherd and reviewers for their valuable feedback. This material is based upon work partly supported by the National Research Foundation (NRF) grant funded by the Korea government's Ministry of Science and ICT (MSIT) under the award 2022R1C1C1003551. We also gratefully acknowledge the awards 2022-22-0103 and 2023-22-0108 from the Yonsei University Research Fund of 2022 and 2023, respectively.

References

[1] Facedancer11. http://goodfet.sourceforge.net/hardware/facedancer11.

[2] Open source USB packet capture for Windows. https://desowin.org/usbpcap.

[3] PCAP capture file format. https://tools.ietf.org/id/draft-gharris-opsawg-pcap-00.html.

[4] strace. https://strace.io.

[5] Wireshark. https://www.wireshark.org.

[6] Write a UDE client driver. https://docs.microsoft.com/en-us/windows-hardware/drivers/usbcon/writing-a-ude-client-driver.

[7] Gautam Altekar and Ion Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[8] Armis Labs. BlueBorne vulnerabilities, 2017. https://armis.com/blueborne.

[9] Ian Beer. An iOS zero-click radio proximity exploit odyssey, 2020. https://googleprojectzero.blogspot.com/2020/12/an-ios-zero-click-radio-proximity.html.

[10] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, 2005.

[11] Gal Beniamini. Over the air - vol. 2, pt. 2: Exploiting the Wi-Fi stack on Apple devices, 2017. https://googleprojectzero.blogspot.com/2017/10/over-air-vol-2-pt-2-exploiting-wi-fi.html.

[12] Gal Beniamini. Over the air - vol. 2, pt. 3: Exploiting the Wi-Fi stack on Apple devices, 2017. https://googleprojectzero.blogspot.com/2017/10/over-air-vol-2-pt-3-exploiting-wi-fi.html.

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

[14] Gal Beniamini. Over the air: Exploiting Broadcom's Wi-Fi stack (part 2), 2017. https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_11.html.

[15] BlueZ Project. BlueZ: Official Linux Bluetooth protocol stack. http://www.bluez.org.

[16] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as Markov chain. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[17] Thomas C Bressoud and Fred B Schneider. Hypervisor-based fault tolerance. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[18] Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. SyzGen: Automated generation of syscall specification of closed-source macOS drivers. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2021.

[19] Jim Chow, Tal Garfinkel, and Peter M Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2008.

[20] Abraham A. Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. HALucinator: Firmware re-hosting through abstraction layer emulation. In *Proceedings of the USENIX Security Symposium*, 2020.

[21] Jonathan Corbet. Finding race conditions with KCSAN, 2019.

[22] 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 ACM Conference on Computer and Communications Security (CCS)*, 2017.

[23] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M Chen. Eidetic systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

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

[25] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[26] Bo Feng, Alejandro Mera, and Long Lu. P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *Proceedings of the USENIX Security Symposium*, 2020.

[27] Dennis Michael Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2006.

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

[29] Google. syzkaller - kernel fuzzer. https://github.com/google/syzkaller.

[30] Liwei Guo and Felix Xiaozhu Lin. Minimum viable device drivers for ARM TrustZone. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2022.

[31] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. Toward the analysis of embedded firmware through automated re-hosting. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2019.

[32] HyungSeok Han and Sang Kil Cha. IMF: Inferred model-based fuzzer. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[33] Yu Hao, Hang Zhang, Guoren Li, Xingyun Du, Zhiyun Qian, and Ardalan Amiri Sani. Demystifying the dependency challenge in kernel fuzzing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2022.

[34] Felicitas Hetzelt, Martin Radev, Robert Buhren, Mathias Morbitzer, and Jean-Pierre Seifert. VIA: Analyzing device interfaces of protected virtual machines. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2021.

[35] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Proceedings of the USENIX Security Symposium*, 2012.

[36] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2019.

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

[38] Kyungtae Kim, Taegyu Kim, Ertza Warraich, Byoungyoung Lee, Kevin Butler, Antonio Bianchi, and Dave (Jing) Tian. FuzzUSB: Hybrid stateful fuzzing of USB gadget stacks. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2022.

[39] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[40] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[41] Andrey Konovalov. USB raw gadget, 2021. https://www.kernel.org/doc/html/v5.14/usb/raw-gadget.html.

[42] Andrey Konovalov and Dmitry Vyukov. KernelAddressSanitizer (KASan): A fast memory error detector for the Linux kernel. *LinuxCon North America*, 2015.

[43] Ignat Korchagin. Exploiting USB/IP in Linux. *Black Hat ASIA*, 2017.

[44] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M Chen, and Jason Flinn. Respec: Efficient online multiprocessor replay via speculation and external determinism. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[45] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. Dr. Checker: A soundy analysis for Linux kernel drivers. In *Proceedings of the USENIX Security Symposium*, 2017.

[46] Dominik Maier and Fabian Toepfer. BSOD: Binary-only scalable fuzzing of device drivers. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2021.

[47] A. Theodore Markettos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[48] Alejandro Mera, Bo Feng, Long Lu, and Engin Kirda. DICE: Automatic emulation of DMA input channels for dynamic firmware analysis. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2021.

[49] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. FuzzBench: An open fuzzer benchmarking platform and service. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

[50] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[51] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate record-and-replay for HTTP. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2015.

[52] Open Virtualization Alliance. Linux kernel virtual machine. https://www.linux-kvm.org.

[53] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing OS fuzzer seed selection with trace distillation. In *Proceedings of the USENIX Security Symposium*, 2018.

[54] Heejin Park and Felix Xiaozhu Lin. GPUReplay: A 50-kb GPU stack for client ML. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[55] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2010.

[56] Hui Peng and Mathias Payer. USBFuzz: A framework for fuzzing USB drivers by device emulation. In *Proceedings of the USENIX Security Symposium*, 2020.

[57] Ivan Pustogarov, Qian Wu, and David Lie. Ex-vivo dynamic analysis framework for Android device drivers. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2020.

[58] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. SymDrive: Testing drivers without devices. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[59] Andrey Ryabinin. UBSan: Run-time undefined behavior sanity checker, 2014.

[60] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In *Proceedings of the USENIX Security Symposium*, 2017.

[61] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-Net: Network fuzzing with incremental snapshots. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2022.

[62] Zekun Shen, Ritik Roongta, and Brendan Dolan-Gavitt. Drifuzz: Harvesting bugs in device drivers from golden seeds. In *Proceedings of the USENIX Security Symposium*, 2022.

[63] 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 *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[64] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent ByungHoon Kang, Jean-Pierre Seifert, and Michael Franz. Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In *Proceedings of the USENIX Security Symposium*, 2020.

[65] Sudarshan M Srinivasan, Srikanth Kandula, Christopher R Andrews, Yuanyuan Zhou, et al. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2004.

[66] 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 Symposium on Operating Systems Principles (SOSP)*, 2021.

[67] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *Proceedings of the USENIX Security Symposium*, 2018.

[68] The kernel development community. Linux NFC subsystem, 2021. https://www.kernel.org/doc/html/v5.14/networking/nfc.html.

[69] The kernel development community. USB/IP protocol, 2021. https://www.kernel.org/doc/html/v5.14/usb/usbip_protocol.html.

[70] The kernel development community. usbmon, 2021. https://www.kernel.org/doc/html/v5.14/usb/usbmon.html.

[71] The QEMU Project Developers. USB emulation. https://qemu-project.gitlab.io/qemu/system/devices/usb.html.

[72] Dave Jing Tian, Adam Bates, and Kevin Butler. Defending against malicious USB firmware with GoodUSB. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2015.

[73] Dave Jing Tian, Grant Hernandez, Joseph I. Choi, Vanessa Frost, Peter C. Johnson, and Kevin R. B. Butler. LBM: A security framework for peripherals within the Linux kernel. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2019.

[74] Dave Jing Tian, Nolen Scaife, Adam Bates, Kevin Butler, and Patrick Traynor. Making USB great again with USBFILTER. In *Proceedings of the USENIX Security Symposium*, 2016.

[75] 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 *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2018.

[76] Matthew Tischer, Zakir Durumeric, Sam Foster, Sunny Duan, Alec Mori, Elie Bursztein, and Michael Bailey. Users really do plug in USB drives they find. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2016.

[77] Richard A. Uhlig and Trevor N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Survey (CSUR)*, 29(2):128–170, jun 1997.

[78] Dmitry Vyukov. Syzbot and the tale of thousand kernel bugs. *Linux Security Summit*, 2018.

[79] Zev Weiss, Tyler Harter, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. ROOT: Replaying multithreaded traces with resource-oriented ordering. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[80] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in OS kernels. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2018.

[81] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2019.

[82] Michał Zalewski. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl.

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

A Wireless Networking Configurations

A.1 Wi-Fi Client & AP

A common Wi-Fi communication scenario is connecting a device as a client station to a nearby access point (AP). In this setup, we configure VM Ⓐ as a client and VM Ⓑ as an AP. VM Ⓑ starts *hostapd*, a user-mode service in Linux that can manage APs, configuring the device as an AP and scannable. After the configuration of VM Ⓑ, VM Ⓐ starts *wpa_supplicant*, a user-mode service in Linux that implements a Wi-Fi client station, which scans nearby APs and associates itself with VM Ⓑ's device using a predefined configuration of SSID and password. After association, each side can perform actions using command-line tools: *wpa_cli* in VM Ⓐ and *hostapd_cli* in VM Ⓑ. Using these tools, we check the status of the AP and client, change the network configuration, etc. We then kill user-mode services, and unplug the devices from the VMs.

A.2 Bluetooth Peer-to-Peer

In this setup, each VM is configured to run BlueZ [15], the official Linux Bluetooth protocol stack and vendor-specific Bluetooth device drivers, by using *bluetoothctl*, a user-mode command-line interface to BlueZ. Both VM Ⓐ and VM Ⓑ starts *bluetoothd*, a user-mode Bluetooth service for Linux. VM Ⓐ is configured as an observer, which scans nearby advertisers. VM Ⓑ is configured as an advertiser, which makes itself as discoverable. After scanning, VM Ⓐ discovers the advertiser run by VM Ⓑ, and it initiates the pairing process. After pairing, each side can perform actions using *bluetoothctl*. We then kill user-mode services, and unplug the devices from the VMs.

A.3 NFC Peer-to-Peer

Near Field Communication (NFC) is widely used for many short-range communication such as payment and authentication. This setup configures each VM to run the Linux NFC subsystem [68] for a P2P communication via NFC. Because of the range restriction of NFC, this setup places two devices less than 4cm from each other. Both VM Ⓐ and VM Ⓑ starts *neard*, a user-mode NFC service for Linux. Using *nfctool*, VM Ⓐ is configured as an initiator, which sends commands to the target run by VM Ⓑ. VM Ⓑ is configured as a target, which responds to the commands. Afterwards, user-mode services are killed and the devices are unplugged.

B CVE Details

B.1 CVE-2022-3628

This CVE was assigned to an intra-object buffer overflow bug we found in BCM43236's Wi-Fi driver in Linux. This bug can be triggered by an attacker who controls the USB device; in particular, ReUSB triggered this bug after sending 219 USB messages, followed by another message that contains a manipulated index value (i.e., `event->emsg.bsscfgidx`). This bug

```
ifp = drvr->iflist[event->emsg.bsscfgidx]; // intra-object buffer
↪  overflow
err = brcmf_fweh_call_event_handler(drvr, ifp, event->code,
                                     &emsg, event->data);
```

allows the attacker to read a value at an attacker-controlled location past the end of the vulnerable buffer `drvr->iflist`. This value read from an attacker-controlled location is interpreted as a pointer (i.e., `ifp`). This pointer is passed as an argument to `brcmf_fweh_call_event_handler`, which, unfortunately, uses this pointer to locate an event handler function (i.e., a code pointer) and eventually invokes that function through an indirect call. This means that the attacker controls the target of this indirect call, and, with knowledge of the code layout (i.e., KASLR needs bypassing) the attacker can call arbitrary functions with attacker-controlled arguments.

B.2 CVE-2023-1380

This CVE was assigned to a slab out-of-bounds read bug we found in BCM43236's Wi-Fi driver in Linux, which could lead to the leakage of sensitive kernel data. This bug can be triggered by an attacker who controls the USB device; in particular, ReUSB triggered this bug by sending 304 USB messages where the 302th message contains a manipulated length value, i.e., `assoc_info->req_len`. The bug occurs

```
req_len = le32_to_cpu(assoc_info->req_len);
conn_info->req_ie_len = req_len;
conn_info->req_ie = kmemdup(cfg->extra_buf, conn_info->req_ie_len,
                            GFP_KERNEL); // slab out-of-bounds read
```

when the driver duplicates a source buffer `cfg->extra_buf` to a new buffer as part of the multicasting process through a call to `kmemdup`, where the attacker-controlled `req_len` value, unfortunately, determines how much of the source buffer to duplicate. This means that the attacker can fill the duplicated buffer with the content of the source buffer, *plus* whatever that follows. This duplicated buffer passes through multiple functions, whose content, which may contain sensitive kernel data located past the end of the source buffer, is eventually sent to a user-space socket.