Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints

Dokyung Song, *University of California, Irvine;* Felicitas Hetzelt, *Technische Universität Berlin;* Jonghwan Kim and Brent Byunghoon Kang, *KAIST;* Jean-Pierre Seifert, *Technische Universität Berlin;* Michael Franz, *University of California, Irvine*

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

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

August 12–14, 2020

978-1-939133-17-5

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

Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints

Dokyung Song
University of California, Irvine

Felicitas Hetzelt
Technische Universität Berlin

Jonghwan Kim
KAIST

Brent Byunghoon Kang
KAIST

Jean-Pierre Seifert
Technische Universität Berlin

Michael Franz
University of California, Irvine

Abstract

Kernel-mode drivers are challenging to analyze for vulnerabilities, yet play a critical role in maintaining the security of OS kernels. Their wide attack surface, exposed via both the system call interface and the peripheral interface, is often found to be the most direct attack vector to compromise an OS kernel. Researchers therefore have proposed many fuzzing techniques to find vulnerabilities in kernel drivers. However, the performance of kernel fuzzers is still lacking, for reasons such as prolonged execution of kernel code, interference between test inputs, and kernel crashes.

This paper proposes lightweight virtual machine checkpointing as a new primitive that enables high-throughput kernel driver fuzzing. Our key insight is that kernel driver fuzzers frequently execute similar test cases in a row, and that their performance can be improved by dynamically creating multiple checkpoints while executing test cases and skipping parts of test cases using the created checkpoints. We built a system, dubbed Agamotto, around the virtual machine checkpointing primitive and evaluated it by fuzzing the peripheral attack surface of USB and PCI drivers in Linux. The results are convincing. Agamotto improved the performance of the state-of-the-art kernel fuzzer, Syzkaller, by 66.6% on average in fuzzing 8 USB drivers, and an AFL-based PCI fuzzer by 21.6% in fuzzing 4 PCI drivers, without modifying their underlying input generation algorithm.

1 Introduction

Device drivers are the leading cause of kernel vulnerabilities [15, 47, 56]. A primary reason is the breadth and variety of driver implementations, which impedes scalable and coherent security analysis. Worse, they expose a richer attack surface than other kernel subsystems: kernel-mode drivers expose a peripheral attack surface in addition to the system call attack surface. Consequently, vulnerabilities in device drivers have been frequently discovered and exploited by adversaries through both attack surfaces in local and remote attack scenarios [7, 10, 12, 14, 20, 42].

In practice, fuzzing has proven to be effective at finding vulnerabilities in different kernel subsystems, including device drivers [2, 18, 24, 26, 29, 46, 53, 64]. Many techniques have been proposed to improve kernel fuzzing, e.g., via hand-written input grammars [24] or various forms of static and dynamic analysis [2, 18, 26, 29, 46, 53, 64]. For fuzzing device drivers, Syzkaller represents the state-of-the-art, incorporating many of the proposed techniques [24]. It recently added support for fuzzing the peripheral attack surface of USB device drivers in Linux, which leverages all of its existing smart fuzzing capabilities such as grammar- and coverage-guidance in input generation. In its early development stage, Syzkaller already discovered hundreds of vulnerabilities from a wide range of device drivers [23], demonstrating its effectiveness.

However, despite these recent developments, high-throughput kernel driver fuzzing still remains challenging. A driver's execution can easily be prolonged during its loading and initialization, or peripheral input processing in general. Low-priority, time-consuming tasks in kernel space are typically processed asynchronously and in a deferred manner, increasing total input processing time. Also, executing each test case may change the driver's internal state, which, in turn, can negatively influence subsequent test case executions. This influence can result in the driver locking itself up [18, 55], or unstable system state in general, when, for example, a memory corruption bug corrupts a wider system state. Unloading and reloading the driver after executing each test case, and rebooting the system after hitting a bug, can prevent the interference between test case executions, but doing so incurs a significant reduction in fuzzing throughput. As an alternative, prior work used a system snapshot created at system startup to always restore a clean state of the system for each test case, skipping time-consuming reboots. However, snapshot techniques at the virtual machine level without optimizations can be too costly (e.g., QEMU's VM snapshot [1]), and user-mode system snapshot techniques either suffer from similar performance problems [2] or require extensive driver porting efforts when a user-mode kernel is used [64].

This paper proposes a new primitive—*dynamic virtual machine checkpointing*—to address the aforementioned challenges and enable high-throughput, clean-state kernel driver fuzzing. The core idea is to continuously create checkpoints during a fuzzing run in order *to skip previously observed, and checkpointed operations* that a kernel driver fuzzer performs. We find that test cases generated by fuzzers often have a substantial amount of similarities between them, leading to a repeated traversal of identical target driver states. Virtual machine checkpoints, strategically created by our checkpoint management policies, can be used to directly restore the virtual machine state established by time-consuming operations without repeatedly executing them. This primitive reduces the average test case execution time and, by design, ensures that no residual states remain after executing a test case; even if the test case causes a kernel panic, a known virtual machine state can be quickly restored from an existing checkpoint.

We built a system, called *Agamotto*[1], around this new virtual machine checkpointing primitive, which can *transparently* accelerate kernel driver fuzzers regardless of the OS and the peripheral bus on which a target driver operates. Agamotto abstracts away from the heterogeneity of the device drivers and OSs, leveraging our virtual machine checkpointing primitive together with other commodity virtual machine introspection mechanisms that are readily available for different OSs. Agamotto is also fuzzer-agnostic, as it provides an abstract *fuzzer interface* that can be implemented to accelerate any existing kernel driver fuzzers.

Different forms of virtual machine checkpointing mechanisms have been used in many contexts for high availability (e.g., fault tolerance or live migration) [16, 19], or debugging and introspection purposes [21, 34]. In these contexts, checkpoints are typically created on a single timeline, which together describe a single execution path that *has been realized* in production, debugging, or introspection settings. In contrast, checkpoints created during a fuzzing run describe multiple possible execution paths that *can be realized* in adversarial settings depending on the input. Under multi-path exploration, virtual machine checkpoints are frequently created, and used for virtual machine restoration; therefore, achieving time and space efficiency of checkpointing and restoration mechanisms becomes a key challenge. To address this, we heavily optimized both virtual machine checkpointing and restoration mechanisms, making their run-time and space overheads suitable for high-throughput fuzzing.

We thoroughly and conservatively evaluated the run-time and memory overheads of our proposed checkpointing primitives as well as the effectiveness of our system, Agamotto. The results show that the checkpointing primitive creates a new promising dimension in the optimization space of kernel driver fuzzing. In fuzzing 8 USB and 4 PCI drivers, 35.6% of test case executions on average skipped one or more operations by directly restoring the virtual machine from a checkpoint automatically created and managed by Agamotto. The creation and management of checkpoints incur a run-time overhead, but their impact on the fuzzing throughput is significantly reduced with our controlled checkpoint creation and optimized checkpointing primitives. Overall, the utility of multiple checkpoints created by Agamotto outweighed the cost; Agamotto improved the throughput of USB and PCI driver fuzzing, on average, by 66.6% and 21.6%, respectively. Moreover, as fuzzing went deeper, Agamotto became more effective—the throughput increased by up to 70.5%—thanks to the checkpoints created in deeper code paths. This is without making any change to the fuzzing algorithm (i.e., input generation algorithm) of the fuzzers we used. This means that our approach leverages an overlooked aspect of the fuzzing algorithm; the fuzzing algorithms employed by state-of-the-art fuzzers produce many similar test cases during a fuzzing run, and thus can benefit from checkpoints created while executing earlier test cases. Further improvement could also be possible by optimizing Agamotto's dynamic checkpointing policies together with the fuzzing algorithm itself.

In summary, we make the following contributions:

- **A new primitive in kernel driver fuzzing.** We introduce dynamic virtual machine checkpointing to accelerate kernel driver fuzzing. This new primitive is OS-, bus- and driver-agnostic, since it operates at the virtual machine level,, and it opens a new dimension in the optimization space of kernel driver fuzzing.
- **Checkpoint management policies and optimization techniques.** We present checkpoint management policies that can increase the utility of checkpoints created during our dynamic checkpointing process. We also present virtual machine checkpointing and restoration algorithms optimized for fuzzers' multi-path exploration.
- **Improved kernel driver fuzzing throughput.** By applying the proposed techniques to a state-of-the-art USB fuzzer, Syzkaller, we improved its throughput by 66.6% on average, *without* modifying the underlying fuzzing algorithm. We also built a fuzzer for PCI drivers based on AFL, and improved its throughput by 21.6% on average.

2 Motivation

2.1 Peripheral Attack Surface

Kernel subsystems are typically exposed to adversaries through the system call interface. Device drivers expose an additional attack surface: the peripheral interface. This attack surface is subject to physical attacks such as an "evil maid attack" [52], remote attacks such as an "airborne attack" [6, 10, 12], or even social engineering attacks [61]. An attacker having physical access to the victim system, e.g., an

[1] Available at: https://github.com/securesystemslab/agamotto

Table 1: Comparison between kernel fuzzing approaches.

	Clean State	Compatibility*	High Speed
No Snapshot [18, 24, 55]		✓	✓
User-mode Snapshot (LibOS) [64]	✓		✓
User-mode Snapshot (Emulation) [2]	✓	✓	
VM Snapshot	✓	✓	
VM Snapshot with Agamotto	✔	✔	✔

* Compatible with kernel-mode drivers.

evil maid, can compromise the system by physically connecting malicious peripherals. A remote attacker, who can reach communication peripherals such as Wi-Fi or Ethernet controllers, can remotely compromise the system by attacking these controllers as well as their device drivers.

Scope. This paper investigates the USB and PCI peripheral attack surfaces of kernel-mode drivers. Our choice was motivated by (i) their accessibility to potential adversaries, as exemplified by real-world attacks originating in USB or PCI devices [10, 12, 20, 42], and (ii) their widespread use. We find that many security-sensitive devices, such as communication peripherals that have their own external access vector, operate on USB, PCI, or both. According to a survey on Linux device drivers [31], more than 70% of drivers target either USB or PCI devices. Hundreds of bugs already found by Syzkaller's USB fuzzer are from a wide range of driver classes [23], which also reflects the prevalence of USB devices. Although this paper investigates two peripheral buses, we emphasize that our approach is not bus-specific. Any fuzzer for a peripheral attack surface, regardless of its underlying I/O interception mechanism, can be accelerated with our approach. We provide more insight on I/O interception in Section 3.6.

2.2 Why Use Snapshots?

Prior work used different snapshot techniques for fuzzing OS kernel subsystems [2] and user-space programs [63, 65]. The basic idea is to snapshot the target program before it starts processing input and run the program from that snapshot for each test input. This means that every test input executes on the same, clean state of the target program. No residual state remains, by construction, after each iteration of the fuzzing loop. Test inputs do not interfere with each other, increasing the reproducibility of bugs [64]. Even when a test input corrupts the program state by hitting bugs, a fresh target program state can always be restored from an existing snapshot, which effectively provides crash resilience. Test inputs after a crash can execute without re-executing time-consuming initial bootstrap operations (e.g., system reboot in kernel fuzzing). Fuzzers for user-space programs typically achieve this using *fork()*. A new, fresh child process is forked from a single parent process for each test input, the performance of which is optimized via the copy-on-write mechanism. Several kernel fuzzers also use different forms of snapshots for a reboot-free and reproducible fuzzing [2, 64].

2.3 Why Not Use Snapshots?

Although snapshot techniques ensure clean-state fuzzing, the snapshot operations themselves may pose a non-negligible overhead. In particular, system-wide snapshot techniques, e.g., using an emulated, user-mode virtual machine with a *fork*-based snapshot technique [2], or using a hardware-accelerated virtual machine with a full memory snapshot technique, can be expensive. Several fuzzing tools do not use snapshot techniques at all [24, 38, 53], due in part to the overhead. For example, LibFuzzer [38], an in-process user-space fuzzer, and Syzkaller [24], a state-of-the-art kernel fuzzer, execute each test case on the same running instance of the program, and cleaning the program state is left to the user. The user must write cleanup routines to clean up global states that may persist across fuzzing loop iterations. To reduce the overhead associated with virtual machine snapshots, a library OS approach was proposed [64]. This approach, however, lacks compatibility with kernel-mode drivers; it requires manual efforts (or a sophisticated tool [13]) to port device drivers into user-mode ones.

3 Design

This paper proposes dynamic virtual machine checkpointing as a key primitive to improve the performance of kernel driver fuzzing. The key idea is to *dynamically* create checkpoints during a fuzzing run, and use these checkpoints to skip time-consuming parts in the execution of test cases. Recurring sequences of operations that test cases perform need not be executed many times; instead, the state of a virtual machine established by such operations, once checkpointed, can be directly restored from a checkpoint. This idea underpins the design of our system, Agamotto.

Agamotto addresses the shortcomings of prior work, as described in Table 1. It uses virtual machine snapshots (or "checkpoints") and thus inherits all of its advantages—clean-state, reboot-free fuzzing. In contrast to prior snapshot-based approaches, which used a single snapshot created at a fixed point in time (usually at program startup), however, Agamotto creates *multiple* checkpoints automatically at strategic points during a fuzzing run. These checkpoints allow Agamotto to skip initial parts of many test cases, improving the overall fuzzing performance. In addition, we heavily optimized individual virtual machine checkpointing primitives for an efficient multi-path exploration, which limits the performance impact of the primitives themselves.

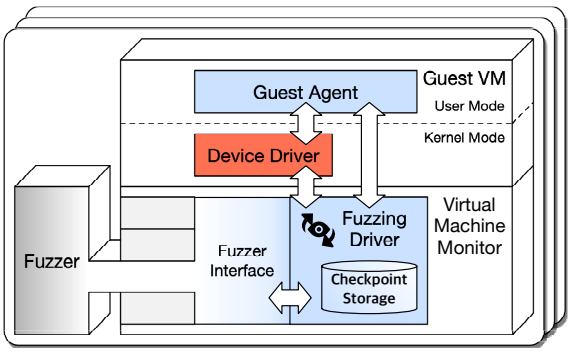


Figure 1: High-level overview of Agamotto.

3.1 System Overview

Figure 1 shows a high-level overview of Agamotto. The architecture of Agamotto takes the form of a typical virtual machine introspection infrastructure. A full operating system including the kernel-mode **device driver**—the fuzzing target—runs within a **guest virtual machine**. Unlike prior work [2], Agamotto does not impose any constraint on the mode of execution; the guest virtual machine can execute *natively*, using hardware support (e.g., Intel's Virtual Machine Extensions [28]) when available.

The **fuzzer**, whose primary task is to generate test cases and process their execution feedback, is placed *outside* this virtual machine, running alongside the **virtual machine monitor**. Some kernel fuzzers such as Syzkaller place the fuzzer inside the guest virtual machine. This architecture is not suitable when using virtual machine checkpointing, because, as we restore the virtual machine from a checkpoint, the fuzzer's internal states about the fuzzing progress would also get restored and thus lost. By placing the fuzzer outside the virtual machine, the fuzzer survives virtual machine restorations. Moreover, the fuzzer is shielded against guest kernel crashes and subsequent virtual machine reboots, limiting their impact on the fuzzing progress.

The **fuzzer interface** is a fuzzer abstraction layer that hides details about individual fuzzers from other components. A new fuzzer can be added by implementing various callbacks defined in this interface. These callbacks are invoked by the **fuzzing driver**, the core component of Agamotto placed inside the virtual machine monitor, which (i) drives the fuzzing loop interacting with both the fuzzer as well as the guest virtual machine, and (ii) creates and manages virtual machine checkpoints. The **guest agent**, running inside the guest virtual machine, provides the fuzzing driver with finer-grained virtual machine introspection capabilities. For example, as the guest agent starts at boot, it notifies the fuzzing driver of the boot event, so that it can start the fuzzing loop.

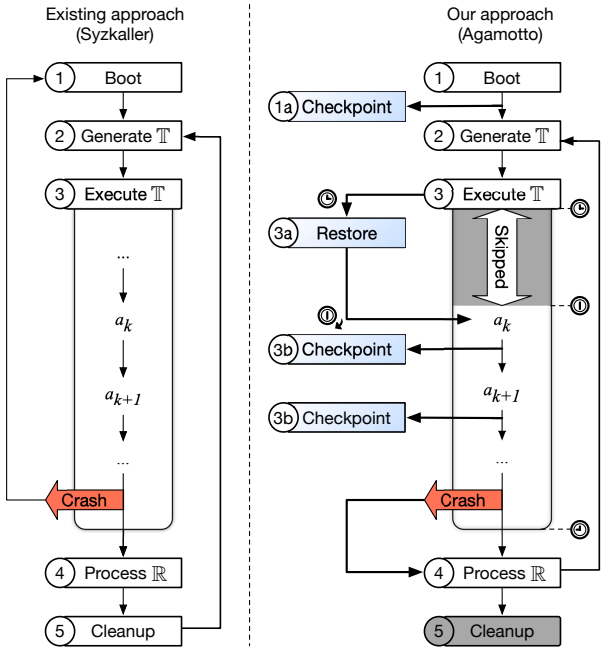


Figure 2: Fuzzing loop comparison.

3.2 Fuzzing Loop

The fuzzing driver component of Agamotto drives the main fuzzing loop. In each iteration of the fuzzing loop, a fuzzer generates a single test case, executes it, and processes the result of its execution as feedback. In fuzzing event-driven systems such as OS kernels, each test case generated by the fuzzer can be defined as the sequence of actions it performs on the target system. Formally, let $\mathbb{S} = \{S_0, S_1, ..., S_N\}$ be the set of states of the fuzzing target, and $\mathbb{T}$ be a fuzzer-generated test case, which comprises a sequence of N actions, denoted by an ordered set $\{a_1, a_2, ..., a_N\}$. An execution of $\mathbb{T}$, denoted by a function $exec(\mathbb{T})$, is a sequential execution of actions in $\mathbb{T}$ on the fuzzing target. Each action $a_i \in \mathbb{T}$ (for $i \in \{1, ..., N\}$) moves the state of the fuzzing target from S_{i-1} to S_i.[2] The target state observed by the fuzzer (e.g., coverage) is denoted by $\mathbb{R} = \{R_1, R_2, ..., R_N\}$, where each element $R_i \subset S_i$ is the fuzzer-observed state of the fuzzing target after executing a_i. We use this notation throughout the paper.

Figure 2 depicts Agamotto's fuzzing loop in comparison with Syzkaller's fuzzing loop using the above notation. The differences are (i) the added flows into checkpoint and restore and (ii) the removed flows into cleanup and reboot. Virtual machine restoration is initiated after generating, but before executing, a given test case. A checkpoint request is issued and evaluated after each action of a test case. Agamotto skips both cleanup and reboot, since a consistent virtual machine

[2] We use a transition-relation style of specifying concurrent, reactive programs (e.g, an OS kernel) to incorporate non-determinism [37, 50]. In other words, a_i is a relation between S_{i-1} and S_i, not a function.

state is always restored from a checkpoint without requiring manual cleanup, even after a crash.

After the guest virtual machine boots, but before it starts executing any test case (① in Figure 2), the first checkpoint, which we call the *root checkpoint*, is created (①a). Then, the fuzzer generates a test case (②) and starts executing it (③). Based on (i) the test case just generated and (ii) available checkpoints, the fuzzer decides what checkpoint the test case can start executing from and restores the virtual machine from the chosen checkpoint (③a). Initial parts of the test case, the result of which is already contained in the checkpoint, are skipped.

During the execution of a test case, secondary checkpoints are requested and created according to a configurable checkpoint policy. After executing each action, the test case executing inside the guest virtual machine sends a checkpoint request to the fuzzer (③b). Then Agamotto's checkpoint policy decides whether to checkpoint the virtual machine or not.

Once a test case has been executed, either successfully, with a failure (e.g., timeout), or with a system failure (e.g., kernel crash), the execution result (e.g., coverage) is sent to and subsequently processed by the fuzzer (④). If a test case did not execute in full, but only until kth action, a_k, due to timeouts or system failures, the result for only the executed parts of the test case, $\{R_1, R_2, ..., R_k\}$, will be sent to the fuzzer.

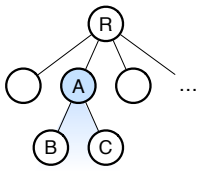
Since restoring the virtual machine entails a full system cleanup, Agamotto skips an explicit cleanup process, if any (⑤). To avoid influence between iterations, existing kernel driver fuzzers either perform an explicit cleanup [24] or simply ignore the issue [18, 55]. Agamotto uses virtual machine restoration, which does not allow any internal system state, even corrupted or inconsistent ones created by kernel bugs or panics, to transfer between iterations, without requiring manually-written cleanup routines.

A bug may occur during the cleanup process that we skip. However, potential bugs that arise in the cleanup process can be found by actively fuzzing the cleanup routines. This way, a cleanup routine can be tested more thoroughly, fully leveraging whatever smart fuzzing capabilities that the fuzzer provides. For example, a fuzzer may generate a corner test case that calls, the cleanup routine multiple times in between other actions, which may trigger more interesting and potentially more dangerous behavior of the driver under test.

3.3 Checkpoint Store and Search

While the fuzzing loop is running, multiple checkpoints get created, which we store in Agamotto's checkpoint storage. To reduce the overhead induced by processing QEMU's snapshot format we manually manage the (re)storing of guest and device memory pages and use memory-backed volatile storage to capture the remaining virtual machine state.

The volatile state of a virtual machine comprises its CPU and memory state, and any bookkeeping information about the virtual machine such as device states kept by the virtual machine monitor. A virtual machine checkpoint must contain all the volatile information to be able to fully restore the state of a virtual machine at a later point in time.



Node	Label
R	$\{\}$
A	$\{a_1, a_2\}$
B	$\{a_1, a_2, a_3, a_4\}$
C	$\{a_1, a_2, a_5, a_6, a_7\}$

Figure 3: Checkpoint tree example.

The state of a virtual machine upon each checkpoint request can be attributed to the executed part of the test case. Therefore, we label each newly created checkpoint as the *prefix* of a test case that represents only the executed part of a test case. That is, given a test case, $\mathbb{T} = \{a_1, a_2, ..., a_N\}$, the checkpoint created after executing kth action is labeled as $\mathbb{T}_{1..k} = \{a_1, a_2, ..., a_k\}$.

Since the root checkpoint is requested when no part of any test case has executed, it is labeled as an empty test case. Checkpoints subsequently created are marked as a non-empty test case. Checkpoints are stored in a prefix tree, which we call a *checkpoint tree*. Each node in this tree represents a checkpoint and is labeled as a prefix of the test case that was executing when this checkpoint was created. An example checkpoint tree is depicted in Figure 3.

The checkpoint tree forms an efficient search tree of checkpoints. After generating a new test case, Agamotto searches for a checkpoint from which to restore the virtual machine. To find the checkpoint that saves the largest amount of time in executing the test case, Agamotto traverses the checkpoint tree searching for a node that has a label that matches the *longest prefix* of the given test case. In Figure 3, given a test case, $\mathbb{T}' = \{a_1, a_2, a_7, a_8\}$, for example, Agamotto finds the node Ⓐ, which has the label that matches the longest prefix, $\{a_1, a_2\}$. Since the checkpoint tree is a prefix tree, this longest prefix match can be performed efficiently without scanning all the checkpoints stored in the tree.

The checkpoint tree also constitutes an incremental checkpoint dependency graph when checkpoint storage is further optimized with incremental checkpoints (see Section 3.5.1).

3.4 Checkpoint Management Policies

3.4.1 Checkpoint Creation Policy

Checkpointing is requested after executing each action in a test case. A checkpoint creation policy decides, upon each checkpoint request, whether to create a checkpoint or not. A checkpoint creation policy should create checkpoints frequently enough, to increase the chances of finding a checkpoint in restoring the virtual machine later, thus saving time.

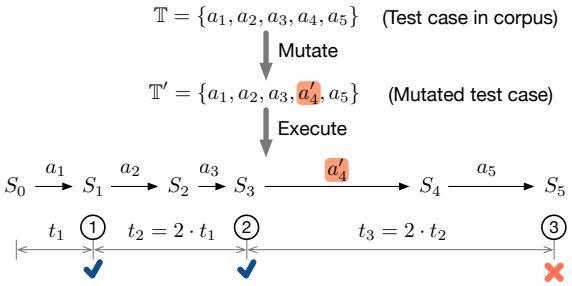


Figure 4: Checkpoint creation policy enforcement example.

Checkpointing should not be too frequent, however, because (i) the checkpointing operation itself adds a run-time overhead and (ii) each newly created checkpoint adds memory pressure to the checkpoint storage. Excessive creation of checkpoints, whose expected gain is less than its cost, must be avoided. We present two general checkpoint creation policies, which take these two requirements into account.

Checkpointing at Increasing Intervals. This policy creates checkpoints at configurable intervals in the timeline of the guest virtual machine. Upon each checkpoint request, we measure the time elapsed since the last checkpoint, and, if it exceeds the configured interval, a checkpoint is created. The intervals can be configured to be constant, or dynamically determined. We use an adaptive interval that increases as the level of the last checkpoint node in the checkpoint tree increases. In particular, we use an exponentially increasing interval using two as the base; this means that the policy requires a guest execution time twice as long as the one that was required for the last checkpoint (see ① and ② in Figure 4). The idea is to reduce the number of checkpoints created later in time during a test case execution, thus alleviating the overhead of checkpoint creation.

Disabling Checkpointing at First Mutation. This policy targets feedback-guided mutational fuzzers, which generate new test cases by mutating parts of older test cases in the corpus. It is well-known that the great majority of mutations do not produce a new feedback signal (e.g., coverage signal [41]), which means that a new test case is more likely to be discarded than to be used for further mutation. Therefore, the expected gain of checkpointing the execution of a test case after the point of a new mutation is low. To reduce the overhead of checkpointing, this policy restricts the creation of checkpoints when executing a mutated test case. Specifically, checkpointing is disabled starting from the location of the first mutation in each test case (see ③ in Figure 4). We do allow checkpointing, however, at any point before the new mutation, because the initial part of the test case still corresponds to a prefix of some older test case in the corpus and is likely to occur again as a base for new mutations.

3.4.2 Checkpoint Eviction Policy

Since the size of the checkpoint storage is limited, we cannot store as many checkpoints as created by the checkpoint creation policy. A checkpoint eviction policy evicts an existing checkpoint to free space for a newly created checkpoint when the memory limit allocated for checkpoint storage is reached. Given a configurable checkpoint pool size, checkpoints created by the checkpoint creation policy are unconditionally stored until there is no remaining space. If there is no available space upon creation of a checkpoint, we consult checkpoint eviction policies to find a node to evict.

The goal of a checkpoint eviction policy is to keep a high usage rate of the checkpoints in restoring a virtual machine. A checkpoint eviction policy needs to predict what checkpoints are likely to be used in the near future, to keep those candidates in the checkpoint tree, and evict others.

We use multiple checkpoint eviction policies, which we consult sequentially. Each policy takes a set of nodes in the checkpoint tree as input and produces one or more candidate nodes as output. If a policy produces more than one candidate node, we consult the next policy using the output nodes of the previous policy as its input. We continue consulting each policy in the pipeline until it finds a single checkpoint node to evict.

Policy-1: Non-Active. This policy is placed first in the pipeline, which prevents any active checkpoint nodes from being evicted. Active checkpoint nodes in the checkpoint tree include the node that the virtual machine is currently based on, and, recursively, the parent node of an active node. This policy selects all but the active nodes in the checkpoint tree as eviction candidates, preventing any active node from being evicted. We consider the checkpoints that are currently active to be *spatially close* because they were created in executing a single test case—the unit of fuzzing. This policy promotes preserving the spatial locality between the active checkpoint nodes by evicting others.

Policy-2: Last-Level. This policy selects the nodes in the last level of the checkpoint tree as eviction candidates. As the depth of the checkpoint tree increases, its nodes are labeled with longer, more specialized test cases. The intuition behind selecting last-level nodes as eviction candidates is that the shorter the test case that a checkpoint node is labeled with, the more likely the label matches test cases that the fuzzer would generate in the future. By evicting last-level nodes, this policy effectively balances the checkpoint tree, letting the tree grow horizontally, rather than vertically.

Policy-3: Least-Recently-Used. The last policy in the pipeline is the Least-Recently-Used (LRU) policy, a policy widely known to be effective at managing different types of caches such as CPU data and address translation caches.

We track the time each checkpoint was last used; we say a checkpoint was used, (i) when it was created, or (ii) when the virtual machine was restored from it. The policy evicts the checkpoint used earliest in time. As widely known, an LRU policy promotes the temporal locality present in the checkpoint usage pattern. The more recently a checkpoint was used, the more likely it will be used again. Unlike previous policies, this LRU policy always determines one and only one eviction candidate, because each checkpoint is used at a unique point in time.

3.5 Lightweight Checkpoint and Restore

3.5.1 Incremental Checkpointing

QEMU's default virtual machine snapshot mechanism stores all volatile states of a virtual machine in a snapshot image. Each snapshot can introduce prohibitive space overhead, however, the memory size of the virtual machine being the dominating factor. Thus, this full snapshot mechanism is not suitable for the fuzzing use case, where a large number of virtual machines are created, and their snapshots can quickly consume all the available memory. Creating a full snapshot can also introduce a prohibitively high run-time overhead for a virtual machine with high memory requirements.

To reduce both space and run-time overheads of checkpointing, Agamotto performs *incremental checkpointing*, where only the modified (or "dirty") memory pages are stored into each checkpoint image. The first checkpoint created by Agamotto after the first boot—the root checkpoint—would be identical to what a full snapshot mechanism would create, which contains all pages in memory. Whenever Agamotto creates a new checkpoint based on an existing one, however, only the memory pages that have been modified with respect to the base checkpoint are stored into the checkpoint image. This incremental approach greatly reduces the size of a non-root checkpoint, as well as the time it takes to create one.

The dependencies between incremental checkpoints are already expressed in our checkpoint tree data structure; that is, the virtual machine state of a given node in the checkpoint tree can be fully restored by following the path from the root to that node and incrementally applying checkpoints.

3.5.2 Delta Restore

A strawman approach to restoring a virtual machine using incremental checkpoints is to sequentially apply incremental checkpoint images starting from the root to the target node in an incremental checkpoint tree. The number of memory pages that this strawman approach should restore, however, is greater than the one that a non-incremental snapshot approach would restore; the root checkpoint in an incremental checkpoint tree already contains the full virtual machine state, and additional restorations of incremental checkpoints will add further overhead.

Algorithm 1 Delta restore

```
1:  function DeltaRestore(Src, Dst)
2:      ▷ Collect pages that need to be restored
3:      L ← LowestCommonAncestor(Src, Dst)
4:      Dirty_{Src..L} ← Dirty_{Src}
5:      Temp ← Parent(Src)
6:      while Temp is not L do
7:          Dirty_{Src..L} ← Dirty_{Src..L} ∨ Dirty_{Temp}
8:          Temp ← Parent(Temp)
9:      end while
10:     Dirty_{Dst..L} ← Dirty_{Dst}
11:     Temp ← Parent(Dst)
12:     while Temp is not L do
13:         Dirty_{Dst..L} ← Dirty_{Dst..L} ∨ Dirty_{Temp}
14:         Temp ← Parent(Temp)
15:     end while
16:
17:     ▷ Restore pages starting from the target node
18:     Dirty_{Delta} ← Dirty_{Src..L} ∨ Dirty_{Dst..L}
19:     Temp ← Dst
20:     while Dirty_{Delta} is not empty do
21:         RestorePages(Dirty_{Delta} ∧ Dirty_{Temp})
22:         Dirty_{Delta} ← Dirty_{Delta} ∧ ¬Dirty_{Temp}
23:         Temp ← Parent(Temp)
24:     end while
25: end function
```

In the fuzzing context, high-performance restore is a requirement, because the virtual machine is restored at the beginning of every iteration of the fuzzing loop. However, since Syzkaller's default Linux kernel configuration for USB fuzzing requires at least 512MB of working memory, and Windows requires a minimum of 4GB, it would take up to several seconds for the strawman approach to restore the full virtual machine memory. We, therefore, introduce the *delta restore* algorithm, which minimizes the number of memory pages that are copied during a virtual machine restoration. The full algorithm is described in Algorithm 1. The key idea is to restore (i) only the pages that have been modified in either the current or target virtual machine state after their execution has diverged, and (ii) each modified page only once via bottom-up tree traversal. This means that the number of memory pages that are copied during a virtual machine restoration is bounded by *the number of pages modified* within the current or the target virtual machine state. Observe that, in the strawman approach, the number of copied memory pages is greater than or equal to *the number of all pages* in memory.

Figure 5 contrasts (a) the top-down, strawman approach with (b) our bottom-up, delta restore approach in restoring a virtual machine state. In the given checkpoint tree, the node (Dst) refers to the checkpoint that the system is being restored to, and the node (Src) is a temporary node representing the current system state from which the restoration starts. The node (B) refers to the last checkpoint that the current system state is based on, and the node (R) refers to the root checkpoint.

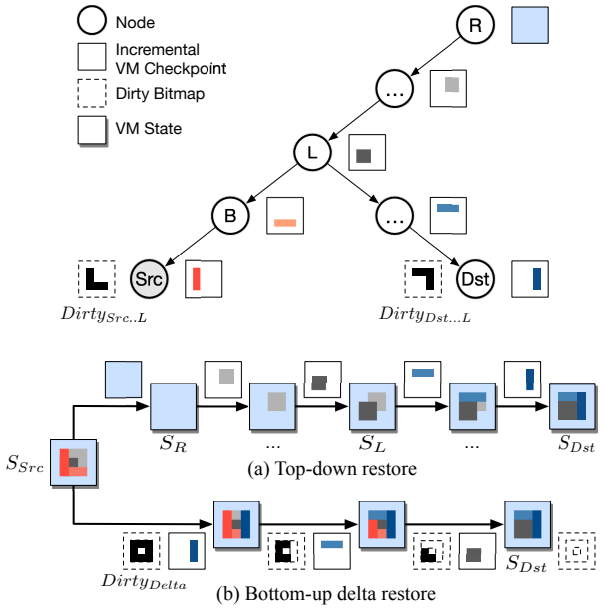


Figure 5: Top-down restore vs. Bottom-up delta restore.

Our delta restore algorithm first locates the lowest common ancestor node (node Ⓛ) of the node Ⓢ and Ⓓ, and computes a bitmap of modified memory pages (or a dirty bitmap) of each node with respect to the node Ⓛ, denoted by $Dirty_{Src..L}$ and $Dirty_{Dst..L}$, respectively. We take the union of these two dirty bitmaps, which we call a *delta* dirty bitmap, denoted by $Dirty_{Delta}$. $Dirty_{Delta}$ contains a complete list of memory pages that need restoring. Then, starting from the node Ⓓ, we traverse the checkpoint tree *backwards* to the root node. At each node during the traversal, we restore only the memory pages that are in $Dirty_{Delta}$ and clear their corresponding bits in $Dirty_{Delta}$ to ensure that each dirty page is restored only once. The traversal stops when $Dirty_{Delta}$ is fully cleared. The strawman approach, by contrast, restores all pages stored in incremental checkpoints starting from the node Ⓡ.

3.6 I/O Interception for Fuzzing

Fuzzing driver code paths that can be reached through a given peripheral interface requires interception and redirection of the driver's I/O requests. We find two common models for driver I/O interception and redirection:

- **User-Space Device Emulation.** I/O requests coming from a kernel driver are redirected to a user-mode program through the system call interface. This approach typically requires kernel source code modifications for intercepting and redirecting driver I/O requests.
- **Device Virtualization.** Device virtualization techniques allow the virtual machine monitor to intercept I/O requests coming from the corresponding kernel driver.

Syzkaller's USB fuzzing mode takes the user-space device emulation approach. It adds a kernel module that intercepts and redirects USB driver I/O requests to a program running in user space via the system call interface. Since Syzkaller already contains many smart fuzzing features such as structure-awareness of USB packets, we modified Syzkaller such that Agamotto can be applied. Our key modification was moving Syzkaller's fuzzer outside of the virtual machine so that the fuzzer survives virtual machine restorations as well as kernel crashes. We also modified the communication channels between Syzkaller's components. The fuzzing algorithm and other aspects of Syzkaller were left unmodified.

For fuzzing the PCI interface, we developed our own fuzzer, which uses a device virtualization approach to intercept the driver's I/O requests at the virtual machine monitor level. A key benefit of this approach is that it does not require kernel modifications; a virtual device can be implemented within the virtual machine monitor without modifying the guest OS kernel. We created a "fake" virtual PCI device in QEMU, and plugged it into QEMU's virtual PCI bus. Our fake PCI device attached to the PCI bus gets recognized by the PCI bus driver as the guest OS kernel boots, and, once the target PCI driver gets loaded, it intercepts all memory-mapped I/O (MMIO) requests coming from the target driver. We fuzzed these MMIO requests by sending fuzzer-generated data to the driver as a response to each driver I/O request.

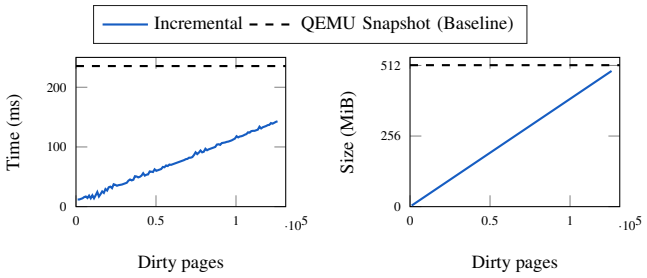
4 Implementation

We implemented Agamotto on top of QEMU 4.0.0 running in an x86 Linux environment [8]. We used the Linux Kernel Virtual Machine (KVM) for hardware accelerated virtualization [43]. We used Syzkaller[3] for USB fuzzing [24], and American Fuzzy Lop (AFL) version 2.52b for PCI fuzzing [65].

Dirty Page Logging. We used KVM's dirty page logging to identify modified pages, as required for our incremental checkpointing and delta restoration techniques. KVM's dirty page bitmap was looked up upon a checkpoint creation request and a virtual machine restoration request. We cleared KVM's dirty page bitmap after each checkpoint creation and virtual machine restoration. We note that KVM's dirty page logging can transparently be accelerated as hardware support—e.g., Page Modification Logging in Intel x86 CPUs—becomes available. Using this dirty page logging, we implemented our own optimized versions of virtual machine checkpointing and restoration mechanisms, since QEMU's snapshot implementation was found to be slower than we expected.

Inter-Component Communication. We used a variety of commodity virtual machine introspection (VMI) mechanisms

[3] Specifically, the commit number: ddc3e85997efdad885e208db6a98bca86e5dd52f



(a) Run-time overhead (b) Memory overhead

Figure 6: Overheads of incremental checkpointing.

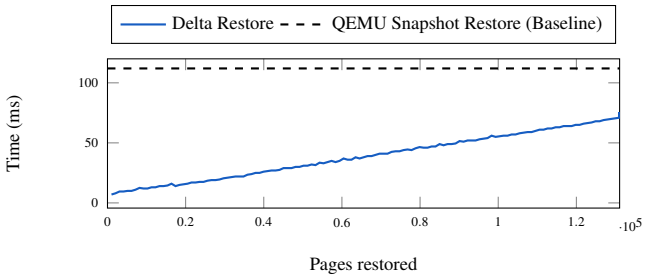


Figure 7: Run-time overhead of delta restore.

to implement inter-component communication channels. Control channels were implemented via hypercalls and VIRTIO pipes established between QEMU and the guest virtual machine [44]. Data channels for bulk data transfer were implemented via direct reads and writes to the guest memory or by using a separate shared memory device.

Syzkaller and AFL Support. Agamotto was designed to support multiple fuzzers, and the current prototype supports two different fuzzers. When running Agamotto with Syzkaller for fuzzing the USB interface, we used Syzkaller's fuzzer (`syz-fuzzer`) as Agamotto's fuzzer component and Syzkaller's executor (`syz-executor`) as Agamotto's guest agent. They were both modified such that they use our VMI-based communication channels. When running Agamotto with AFL for fuzzing the PCI interface, we ran an AFL fuzzer thread as Agamotto's fuzzer component and used a shell script as the guest agent, which simply loads the target PCI driver.

5 Evaluation

We conducted all of our experiments on a machine equipped with AMD EPYC 7601 CPU and 500GB of memory. We targeted device drivers in Linux v5.5-rc3 in our fuzzing experiments. We enabled Kernel AddressSanitizer to expose more bugs [35]. We first evaluate Agamotto's individual primitives, and then the performance of kernel driver fuzzers augmented with Agamotto in both USB and PCI fuzzing scenarios.

5.1 Incremental Checkpointing

We compare the run-time and memory overheads of our incremental checkpointing implementation with the overheads of QEMU's non-incremental snapshot approach [1]. To measure the overheads conservatively, we disabled QEMU's zero page optimization, a checkpoint size reduction technique that handles a page filled with zeros by storing a fixed-size entry in the checkpoint image, instead of storing 4KiB of zeros.

Run-Time Overhead. The run-time overhead of checkpointing primarily depends on the number of pages copied into the checkpoint image. Figure 6a shows the overhead of our incremental checkpointing mechanism, and that of the baseline, when checkpointing a 512MiB memory guest virtual machine. As the number of dirty pages increases, the run-time overhead of incremental checkpointing increases linearly. In contrast, the overhead of the baseline, a non-incremental approach, remains constant regardless of the number of dirty pages. In addition, QEMU's non-incremental checkpoint approach adds an additional overhead due to its implementation and the full inclusion of the device memory, of which only a few pages are dirtied during fuzzing. A full restore can, therefore, take more than 200ms per checkpoint for copying all 131,072 pages, whereas our incremental checkpointing, for a typical range of the number of dirty pages (see Section 5.3), takes less than 20ms on average as it only copies the dirty pages.

Memory Overhead. Figure 6b shows how the size of each checkpoint correlates to the number of dirty pages when checkpointing a 512MiB memory virtual machine. As expected, the size of an incremental checkpoint increases in proportion to the number of pages that have been modified since the last checkpoint. Given the distribution of the number of modified pages, which typically ranges from 0 to 8,000 (see Section 5.3), each checkpoint should take no more than 64MiB. With the zero page optimization enabled, the size of each checkpoint observed in actual fuzzing runs, on average, is less than 32MiB. This is a reduction of 90% or more in size from the baseline.

5.2 Delta Restore

Run-Time Overhead. Figure 7 shows the run-time overhead of our implementation of the delta restore algorithm depending on the number of pages that are restored when restoring a 512MiB memory virtual machine. We used QEMU's default restoration mechanism as the baseline, which restores a virtual machine state from a non-incremental, full snapshot image. The smaller the number of restored pages as computed by our delta restore algorithm, the less time it takes to restore

Table 2: USB and PCI fuzzing targets.

Target	USB (§5.3)	PCI (§5.4)	Path (/drivers/...)
RSI	✓		net/wireless/rsi
MWIFIEX	✓		net/wireless/marvell/mwifiex
AR5523	✓		net/wireless/ath/ar5523
BTUSB	✓		bluetooth/btusb.c
PN533	✓		nfc/pn533
GO7007	✓		media/usb/go7007
SI470X	✓		media/radio/si470x
USX2Y	✓		sound/usb/usx2y
ATLANTIC		✓	net/ethernet/aquantia
RTL8139		✓	net/ethernet/realtek
STMMAC		✓	net/ethernet/stmicro
SNIC		✓	scsi/snic

a virtual machine state. The number of restored pages, as observed in actual fuzzing runs, is significantly lower than the total number of pages in memory (see Section 5.3). With an average number of under 8,000 restored guest and device memory pages, our delta restore implementation can restore the virtual machine in 12.5ms on average, 8.9 times faster than the baseline, QEMU's implementation of the full snapshot restore approach, which takes 112ms on average.

5.3 Syzkaller-USB Fuzzing

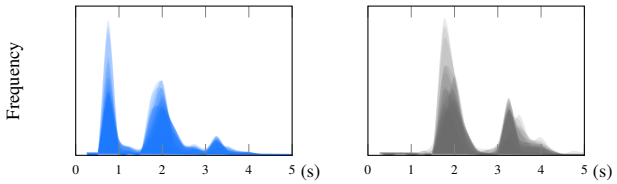
Experimental Setup. We fuzzed USB drivers individually, one in each experiment. We chose 8 USB drivers, as shown in Table 2, which include drivers (i) of 5 different classes, (ii) of different numbers of source lines of code, and (iii) from different vendors. We ran 32 fuzzing instances for three hours in fuzzing each driver. Each instance fuzzed the driver running in a 512MiB memory virtual machine.

We enabled all USB related functions and constrained the parameters of `syz_usb_connect`—i.e., device and interface descriptors—to fuzz the drivers individually in each experiment. To minimize the effects of non-determinism in our experiment, we limited coverage instrumentation to the driver code as well as generic kernel code that drivers call into.[4]

The fuzzing algorithm of Syzkaller was not modified. We only increased Syzkaller's default five-second timeout to ten seconds to encourage deeper exploration.[5] We started fuzzing without any seed input to eliminate its impact on the results. To minimize the randomness inherent in fuzzing algorithms, we used different but fixed sets of PRNG seed values for different instances, using the equation, $\{id_{inst} + \#crashes_{inst} * 128\}$ where $inst = \{0, 1, ..., 31\}$. This equation ensures that seed values (i) are always unique across instances, and (ii) change after each kernel crash. With these adjustments, the randomness of Syzkaller's fuzzing algorithm was controlled; note, however, that the randomness originating in the target system, e.g., coverage signal, was not controlled. To account for this randomness, we ran each experiment three times.

[4] We instrumented the source code under the following directories: drivers, sound/{usb, core}, and net/{bluetooth, nfc, wireless}.

[5] We followed Syzkaller's default timeout model, where each test case can execute for at most three seconds, but, as long as the last action has returned within last one second, it can execute up to ten seconds.



(a) Agamotto execution time (b) Normal execution time

Figure 8: Distribution of the execution time per test case in Syzkaller-USB fuzzing.

We ran two different versions of Agamotto—(i) a full-fledged Agamotto and (ii) Agamotto with only the root checkpoint enabled (Agamotto-R)—to quantify the effectiveness of checkpoints dynamically created by Agamotto. We used Syzkaller as a baseline, only with the aforementioned changes for controlling timeout and randomness. We configured Agamotto with the following additional parameters: The checkpoint pool size was configured to be 12GiB per instance, and we used 500ms as the initial checkpoint creation interval.

Execution Time of Individual Test Cases. Figure 8 shows how much time Agamotto skips in executing each test case. By using fine-grained checkpoints created by Agamotto, the initial parts of many test cases were skipped. We measured each test case's execution time in all experiments (Figure 8a) and computed each test case's normal execution time, the time each test case execution could have taken if fine-grained checkpoints were not used (Figure 8b). Agamotto successfully reduced the execution time of many test cases—a large portion of test cases took less than a second with Agamotto, as shown in Figure 8a.

Overall Fuzzing Throughput. Figure 9 illustrates how much Agamotto improves Syzkaller's USB fuzzing throughput. This overall fuzzing throughput includes the overhead of Agamotto itself. One common trend observed in all experiments is that Agamotto's fuzzing throughput peaks in the first 10 minutes. This is because, as fuzzing instances are started, lots of test cases producing new coverage were discovered and minimized. Each minimized test case was then mutated 100 times and executed in a row. During this period of time in which new inputs were frequently discovered, a large number of *similar* test cases were executed in a row, the throughput of which was greatly improved by Agamotto. As the fuzzing continued, coverage-increasing test cases were seldom discovered, stabilizing the throughput. Still, Agamotto's throughput

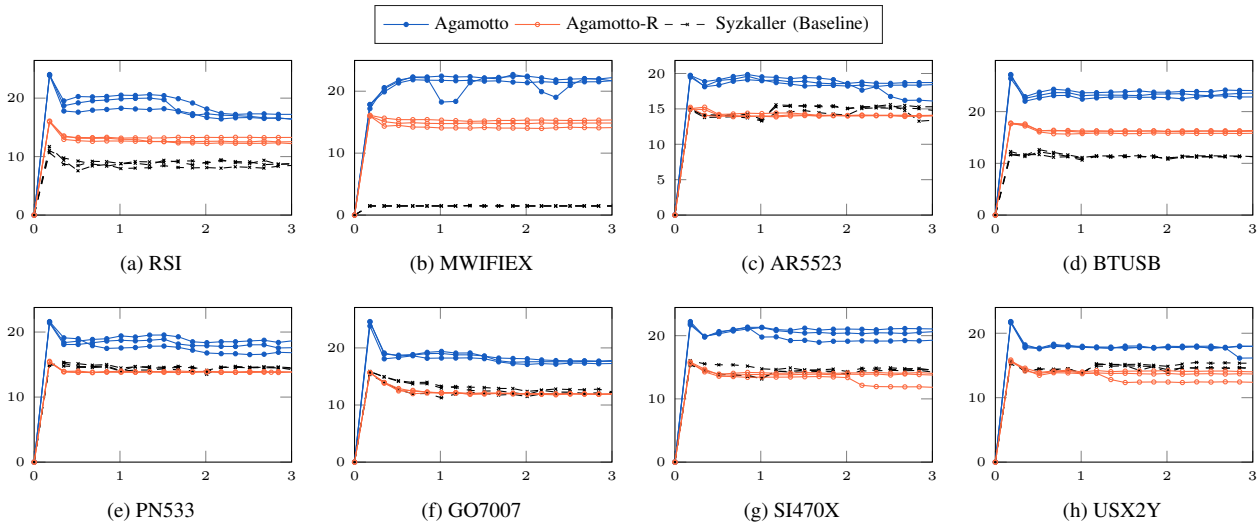


Figure 9: Syzkaller-USB fuzzing throughput (execs/second) measured every 10 minutes for 3 hours.

Table 3: Checkpoint hit and guest execution time statistics.*

	# Checkpoints		# Executions		Guest Exec. Time	
	Created	Evicted	Total	Hit (Rate)	Total	Skipped (Rate**)
RSI	87k	63k	201k	120k (59%)	90.3h	42.1h (31%)
MWIFIEX	19k	9.8k	236k	60k (25%)	28.0h	18.3h (39%)
AR5523	91k	71k	201k	116k (57%)	95.0h	38.6h (28%)
BTUSB	74k	59k	254k	145k (57%)	94.7h	47.1h (33%)
PN533	89k	65k	199k	116k (58%)	95.2h	39.7h (29%)
GO7007	105k	83k	201k	126k (62%)	95.1h	44.5h (31%)
SI470X	88k	67k	223k	130k (58%)	94.9h	43.6h (31%)
USX2Y	92k	76k	195k	90k (46%)	95.0h	29.4h (23%)
Geo. Mean				51.5%		30.9%
ATLANTIC	8.4k	0.6k	191k	43k (22%)	95.2h	18.5h (22%)
RTL8139	17.9k	6.5k	272k	128k (47%)	91.5h	78.9h (46%)
STMMAC	4.8k	0.3k	160k	23k (14%)	95.2h	15.9h (14%)
SNIC	4.0k	0.2k	153k	8.3k (5.4%)	95.3h	5.35h (5.3%)
Geo. Mean				17.0%		16.7%

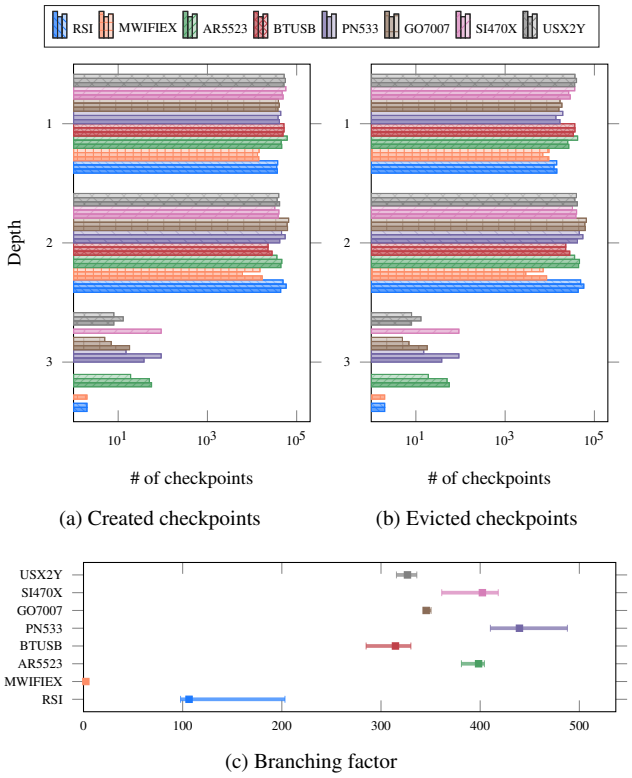
* Median values from 3 independent runs.
** Skipped/(Skipped+Total)

was consistently higher than the baseline. Of the eight analyzed drivers only two experienced kernel crashes (MWIFIEX and RSI). The performance improvement of the remaining targets is therefore solely due to the reduced average execution time by using the checkpoints created by Agamotto.

Checkpoint Utilization and Effectiveness. We identify *a checkpoint hit* as selecting a non-root checkpoint in executing a test case, and *a checkpoint miss* as selecting the root checkpoint. The *hit rate* refers to the portion of executions that had a checkpoint hit among all executions. At each checkpoint hit, a different amount of time is skipped depending on the checkpoint used. Table 3 summarizes the hit rates, as well as the amounts of the guest execution time skipped in each fuzzing experiment. The hit rates and time skip rates vary depending on the driver targeted in each experiment; on average, we achieved 51.5% of hit rate, saving 30.9% in guest execution time.

To quantify the effectiveness of multiple checkpoints created by Agamotto, we compare the throughput of Agamotto and Agamotto-R; the throughput was improved by 38% on average. The shape of the checkpoint tree used to achieve this improvement is characterized in Figure 10. The depths of the checkpoint nodes—i.e., the number of edges from the root node—created and evicted by Agamotto ranged from 1 to 3, and the resulting checkpoint trees had an average branching factor of 175. This large branching factor reflects (i) how Syzkaller explores the input space, and (ii) that our checkpoint management policies favor checkpoint nodes of lower depths in the checkpoint tree. In these checkpoint trees, the length of the restoration path—i.e., the path from the node representing the dirty system state after each test case execution to the node being restored—ranged from 1 to 6, as shown in Figure 11. The widely ranging lengths of the restoration paths mean that different checkpoints created at various depths were actively used for virtual machine restoration, which also supports the utility of multiple checkpoints created by Agamotto.

Resilience to Kernel Panics. Agamotto found several known bugs in RSI and MWIFIEX that were already found and reported in earlier kernel versions by Syzbot [62], but left unfixed. Agamotto found one unknown bug in MWIFIEX. This bug was not found in the baseline (nor Syzbot), as it was obscured by a known, shallow bug in MWIFIEX, which repeatedly caused immediate kernel panics in the baseline. In contrast, since Agamotto puts the fuzzer outside the virtual



(a) Created checkpoints (b) Evicted checkpoints

(c) Branching factor

Figure 10: Distribution of the depths of all the (a) created and (b) evicted checkpoints in the checkpoint trees, as well as (c) the resulting branching factors of the trees, measured in Syzkaller-USB fuzzing.

machine, Agamotto continuously generated and ran test cases despite kernel panics, eventually getting past the known bug to discover this unknown bug. Moreover, Agamotto maintains the fuzzing throughput, even when it frequently hits these bugs. In fuzzing MWIFIEX as well as RSI, where Agamotto encountered bugs more than 6,000 and 200 times in every 10 minutes, their baseline throughput is significantly lower than the ones observed in fuzzing other drivers. Agamotto, in contrast, maintained a similar level of throughput across all experiments.

Dirty Page Statistics. To show that our incremental checkpointing and delta restore techniques are effective in practice, we measured the number of pages that are restored and dirtied in each iteration of the fuzzing loop. The results are shown in Figure 12a and 12b. In our experiments, the number of pages dirtied after executing a test case has an upperbound near 8,000 pages. The number of restored pages is similarly bounded, but often exceeds this limit when the modified pages of the checkpoint being restored do not completely overlap with the current set of dirty pages. This means that, as discussed in Section 5.1 and 5.2, the run-time overhead of virtual machine checkpointing and restoration was greatly reduced.

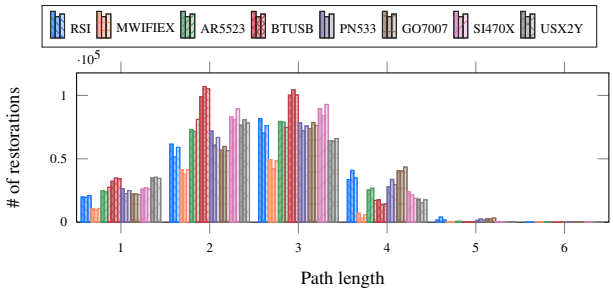
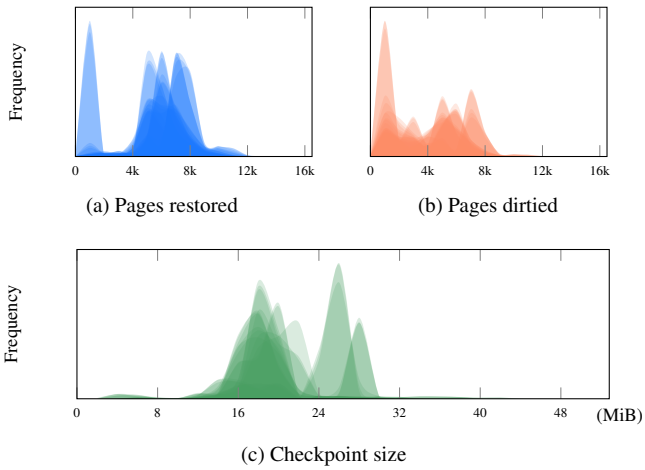


Figure 11: Distribution of the length of the restoration path in Syzkaller-USB fuzzing.



(c) Checkpoint size

Figure 12: Distribution of the number of pages (a) restored and (b) dirtied per iteration, and (c) the size of checkpoints in Syzkaller-USB fuzzing.

Also, with the zero page optimization enabled, most of the checkpoints were found to be smaller than 32MiB, as depicted in Figure 12c.

5.4 AFL-PCI Fuzzing

Experimental Setup. To evaluate our device-virtualization-based PCI fuzzer augmented with Agamotto, we fuzzed four PCI drivers. We used AFL as the fuzzer this time, with its fuzzing algorithm unmodified again; we note that AFL implements a different input generation and scheduling algorithm than Syzkaller. With our own PCI fuzzer, we used a conservative baseline, where Agamotto was applied, but the creation of non-root checkpoints was disabled. In effect, our PCI experiments measured the effectiveness of fine-grained checkpoints created by Agamotto in improving the performance of kernel driver fuzzing.

To avoid introducing randomness through the seed input, we started fuzzing with a single input as the seed, which contains an eight-byte string—"Agamotto" in the ASCII format—and without any dictionary entries. Randomness in the fuzzing

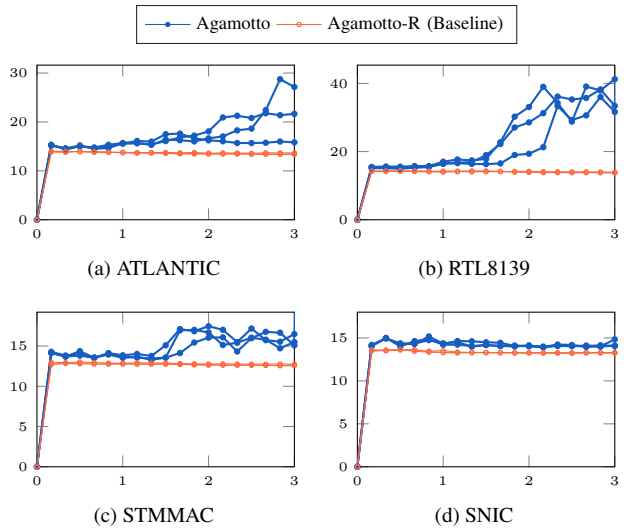


Figure 13: AFL-PCI fuzzing throughput (execs/second) measured every 10 minutes for 3 hours.

algorithm was also controlled the same way as in the USB experiments. We fuzzed each driver using 32 instances for three hours. Since the driver's interactions with a PCI device were faster than what we observed in USB fuzzing, we reduced the starting checkpoint interval to 50ms. We used 100ms as a timeout value; we terminated each iteration 100ms after the driver's last access to the I/O mappings.

Fuzzing Throughput. Although AFL uses a fuzzing algorithm different from Syzkaller's, Agamotto again improved the throughput by 21.6% on average, as shown in Figure 13. We emphasize that neither AFL's nor Syzkaller's fuzzing algorithm produces a sequence of test cases that are optimal for Agamotto to accelerate. In particular, AFL's fuzzing algorithm is not tailored to fuzzing of event-driven systems (e.g., it always mutates each test case in the corpus from the first byte). Still, Agamotto consistently improved the fuzzing throughput in all experiments, and has potential to improve it further when the checkpoint management policies are optimized together with other aspects of the fuzzing algorithm.

Path Coverage. Table 4 shows, in fuzzing each driver, the maximum number of code paths discovered among all fuzzing instances. Agamotto's effectiveness is far more pronounced when the underlying fuzzer keeps discovering new, deeper code paths; the more checkpoints created by Agamotto in deep code paths, the more time it saves. In fuzzing ATLANTIC, RTL8139, and STMMAC, Agamotto covered substantially more paths than the baseline did in the same amount of time; by executing 32.8% more test cases on average, Agamotto covered 47.8% more paths. In fuzzing SNIC, however, AFL only discovered only a limited number of paths. Still, Agamotto did execute 6.2% more test cases than the baseline.

Table 4: Number of executions and discovered paths in AFL-PCI fuzzing.*

	# Executions		# Paths Discovered	
	Agamotto-R	Agamotto **(Increase)**	Agamotto-R	Agamotto **(Increase)**
ATLANTIC	147k	191k **(30.1%)**	112	142 **(18.7%)**
RTL8139	152k	259k **(70.5%)**	71	153 **(115.4%)**
STMMAC	137k	160k **(16.6%)**	87	121 **(50.5%)**
SNIC	144k	153k **(6.2%)**	8	8 **(0%)**

* Median values from 3 independent runs.

6 Discussion

Checkpoint-Aware Fuzzing Algorithm. Our checkpointing primitive introduces a new dimension in the optimization space of fuzzing kernel drivers or other event-driven, reactive systems in general. We conservatively evaluated Agamotto without modifying the underlying fuzzing algorithm; that is, we only leveraged spatial and temporal localities that are already present in the fuzzing algorithm of state-of-the-art fuzzers. Thus, various aspects of the fuzzing algorithm such as input selection and mutation strategies can be revisited. We intend to explore checkpoint-aware or -oblivious fuzzing algorithms as future work.

Supporting Other OSs. Agamotto itself does not require any modification to the OS. Agamotto interacts with the virtual machine using standard virtual machine introspection mechanisms—hypercalls, VIRTIO, and shared memory devices—which are also readily available in closed-source, proprietary operating systems such as Windows [45]. Syzkaller's USB fuzzing component requires kernel modifications, in order to redirect a USB driver's I/O requests to user space via the system call interface. Our modifications to Syzkaller only pertain to its user-space components. Due to its OS-independence Agamotto can be used in conjunction with general kernel fuzzing approaches [53, 54].

Fuzzing the System Call Interface. OS kernels have an event-driven system that processes incoming inputs from peripherals and user-space programs. Agamotto can make fuzzing the system call attack surface more efficient. The degree to which system call fuzzing can benefit from Agamotto, however, can vary depending on the kernel subsystem being targeted. Kernel subsystems that pose similar challenges to those that Agamotto addresses may benefit more than others. Device drivers themselves can also be tested more thoroughly by simultaneously fuzzing both their system call (e.g., *ioctl*) and peripheral attack surface. A local attacker having access to both of these attack surfaces can compromise the OS kernel by exploiting vulnerabilities found by such two-dimensional fuzzing. Prior work showed that two-dimensional fuzzing is

effective at finding bugs in file systems [64]. The same idea can be applied to device drivers, and Agamotto can facilitate an in-depth exploration of their two-dimensional input space.

Fuzzing Other Event-Driven Systems. Virtualization techniques have also been used for running and fuzzing IoT firmware [17, 22, 25, 66]. Although this paper focuses on fuzzing kernel-mode device drivers running in a full-fledged OS kernel, Agamotto's core techniques can be applied to fuzzing IoT firmware running in a virtual machine. Event-driven systems running in user mode can also benefit from Agamotto. For example, when fuzzing a multi-process system where processes interact with each other, Agamotto, as it transparently captures the states of all running processes at the virtual machine level, can facilitate a deep exploration of the state space of such systems as a whole.

Further Optimizations. In a multi-instance fuzzing setup, one can deduplicate checkpoints across fuzzing instances via shared memory. Deduplication allows Agamotto to store more checkpoints in memory, which in turn may prevent thrashing and result in a higher hit rate of checkpoints. One can also explore different checkpointing and eviction policies that are either generic (e.g., the ones we presented), or tailored to certain classes of fuzzing algorithms. Virtual machine introspection primitives can also be further optimized via software and hardware techniques [4].

Limitations. Syzkaller supports a *multi-proc* mode, which runs multiple instances of a fuzzer within a single guest OS, increasing the fuzzing throughput. Agamotto does not support this mode currently, but we believe that this mode can be supported with a finer-grained checkpointing mechanism, e.g., via finer-grained virtual machine introspection or in-kernel checkpoints with kernel modifications [30]. We intend to explore this direction as future work. We emphasize, however, that other aspects of Agamotto, e.g., checkpoint management and optimization techniques, would still apply even with such finer-grained checkpointing mechanism. We also emphasize that our choice of checkpointing at the virtual machine level allows Agamotto to support other VM-based kernel driver fuzzers as we demonstrated with PCI-AFL experiments.

7 Related Work

Peripheral Attacks and Defenses. Malicious peripherals have long been a threat to OS kernel security. A well-known example are malicious USB devices, which often appear as benign USB flash drives [42]. Peripherals other than USB devices, even non-pluggable ones hardwired in an SoC, can also potentially turn malicious via peripheral attacks [9, 11]. Many defenses against malicious peripherals have been proposed [5, 13, 58–60], though securing the peripheral attack surface is still an ongoing effort [39]. With the performance improvements that Agamotto provides, the exploration of the peripheral attack surface via fuzzing can be made more efficient, reducing the time for discovering new vulnerabilities.

Kernel Fuzzing. Many fuzzers exist that find vulnerabilities in kernel subsystems [2, 3, 18, 24, 26, 27, 29, 32, 40, 46, 48, 53–55, 57, 64]. A line of work used various snapshot techniques [2, 64], which we already discussed in detail in Section 2. Other lines of work generally focused on the fuzzing algorithm, e.g., generating coverage-increasing test cases; Agamotto complements these efforts, as it transparently creates and uses checkpoints to save time in executing the generated test cases. Hybrid fuzzing, a combination of symbolic execution and fuzzing, has also been used to find bugs in OS kernels [33, 36, 49, 51]. Since both Agamotto and symbolic execution systems maintain different forms of snapshots, by devising switching mechanisms between the two forms of snapshots, Agamotto could also augment hybrid kernel fuzzing.

8 Conclusion

We presented Agamotto, a system which transparently improves the performance of kernel driver fuzzers using a highly-optimized dynamic virtual machine checkpointing primitive. During a fuzzing run, Agamotto automatically checkpoints the virtual machine at fine-grained intervals and restores the virtual machine from these checkpoints allowing it to skip reboots on kernel panics and to "fast forward" through the time-consuming parts of test cases that are repeatedly executed. We evaluated Agamotto in various USB and PCI fuzzing scenarios with two different fuzzers, and demonstrated the performance benefit that Agamotto can provide, as well as its adaptability.

Acknowledgments

The authors would like to thank our shepherd, Manuel Egele, and the anonymous reviewers for their valuable feedback. The authors also thank Paul Kirth for his help with proofreading this paper. This material is based upon work partially supported by the Defense Advanced Research Projects Agency under contracts FA8750-15-C-0124 and FA8750-15-C-0085, by the United States Office of Naval Research under contract N00014-17-1-2782, by the National Science Foundation under awards CNS-1619211 and CNS-1513837, by the European Commission under the Horizon 2020 Programme (H2020) as part of the LOCARD project (G.A. no. 832735), by the IITP under contract 20190015700021001, and by the NRF under contract 2017R1A2B3006360. Any opinions, findings, and conclusions or recommendations expressed in this

material are those of the authors and do not necessarily reflect the views of our funding agencies.

References

[1] QEMU system emulation user's guide.

[2] TriforceAFL: AFL/QEMU fuzzing with full-system emulation, 2016.

[3] Trinity: Linux system call fuzzer, 2019. https://github.com/kernelslacker/trinity.

[4] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[5] S. Angel, R. S. Wahby, M. Howald, J. B. Leners, M. Spilo, Z. Sun, A. J. Blumberg, and M. Walfish. Defending against malicious peripherals with Cinch. In *Proceedings of the USENIX Security Symposium*, 2016.

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

[7] I. Beer. pwn4fun spring 2014 - Safari - part II, 2014. https://googleprojectzero.blogspot.com/2014/11/pwn4fun-spring-2014-safari-part-ii.html.

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

[9] G. 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.

[10] G. 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.

[11] G. 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.

[12] G. 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.

[13] S. Boyd-Wickizer and N. Zeldovich. Tolerating malicious device drivers in Linux. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2010.

[14] O. Chang. Attacking the Windows NVIDIA driver, 2017. https://googleprojectzero.blogspot.com/2017/02/attacking-windows-nvidia-driver.html.

[15] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[16] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the USENIX Symposium on Networked Systems Design & Implementation*, 2005.

[17] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer. HALucinator: Firmware re-hosting through abstraction layer emulation. In *Proceedings of the USENIX Security Symposium*, 2020.

[18] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna. DIFUZE: Interface aware fuzzing for kernel drivers. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[19] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the USENIX Symposium on Networked Systems Design & Implementation*, 2008.

[20] A. Davis. USB – undermining security barriers. *Black Hat USA*, 2011.

[21] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. 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.

[22] B. Feng, A. Mera, and L. Lu. P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *Proceedings of the USENIX Security Symposium*, 2020.

[23] Google. Found Linux kernel USB bugs, 2019. https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs_usb.md.

[24] Google. syzkaller - kernel fuzzer, 2019. https://github.com/google/syzkaller.

[25] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, and G. Vigna. Toward the analysis of embedded firmware through automated rehosting. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2019.

[26] H. Han and S. K. Cha. IMF: Inferred model-based fuzzer. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[27] J. Hertz and T. Newsham. A Linux system call fuzzer using TriforceAFL, 2016. https://github.com/nccgroup/TriforceLinuxSyscallFuzzer.

[28] Intel. Intel 64 and IA-32 architectures software developer's manual - Chapter 23 Introduction to Virtual Machine Extensions. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.pdf.

[29] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[30] A. Kadav, M. J. Renzelmann, and M. M. Swift. Fine-grained fault tolerance using device checkpoints. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[31] A. Kadav and M. M. Swift. Understanding modern device drivers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[32] S. Keil and C. Kolbitsch. Stateful fuzzing of wireless device drivers in an emulated environment. *Black Hat Japan*, 2007.

[33] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim. CAB-Fuzz: Practical concolic testing techniques for COTS operating systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.

[34] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2005.

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

[36] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2010.

[37] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(2):190–222, 1983.

[38] LLVM Developers. libFuzzer – a library for coverage-guided fuzz testing, 2019. https://llvm.org/docs/LibFuzzer.html.

[39] A. T. Markettos, C. Rothwell, B. F. Gutstein, A. Pearce, P. G. Neumann, S. W. Moore, and R. 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.

[40] M. Mendonça and N. Neves. Fuzzing Wi-Fi drivers to locate security vulnerabilities. In *Proceedings of the European Dependable Computing Conference (EDCC)*, 2008.

[41] S. Nagy and M. Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[42] K. Nohl and J. Lell. BadUSB – on accessories that turn evil. *Black Hat USA*, 2014.

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

[44] Open Virtualization Alliance. Virtio. https://www.linux-kvm.org/page/Virtio.

[45] Open Virtualization Alliance. Windows guest drivers. https://www.linux-kvm.org/page/WindowsGuestDrivers.

[46] S. Pailoor, A. Aday, and S. Jana. Moonshine: Optimizing OS fuzzer seed selection with trace distillation. In *Proceedings of the USENIX Security Symposium*, 2018.

[47] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten years later. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[48] J. Pan, G. Yan, and X. Fan. Digtool: A virtualization-based framework for detecting kernel vulnerabilities. In *Proceedings of the USENIX Security Symposium*, 2017.

[49] J. Patrick-Evans, L. Cavallaro, and J. Kinder. POTUS: Probing off-the-shelf USB drivers with symbolic fault injection. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[50] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Current trends in Concurrency*, pages 510–584. Springer, 1986.

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

[52] J. Rutkowska. Why do I miss Microsoft BitLocker?, 2009. http://theinvisiblethings.blogspot.com/2009/01/why-do-i-miss-microsoft-bitlocker.html.

[53] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In *Proceedings of the USENIX Security Symposium*, 2017.

[54] S. Schumilo, R. Spenneberg, and H. Schwartke. Don't trust your USB! how to find bugs in USB device drivers. *Black Hat Europe*, 2014.

[55] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz. PeriScope: An effective probing and fuzzing framework for the hardware-OS boundary. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[56] J. V. Stoep and S. Tolvanen. Year in review: Android kernel security. *Linux Security Summit*, 2018.

[57] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *Proceedings of the USENIX Security Symposium*, 2018.

[58] D. J. Tian, A. Bates, and K. Butler. Defending against malicious USB firmware with GoodUSB. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2015.

[59] D. J. Tian, G. Hernandez, J. I. Choi, V. Frost, P. C. Johnson, and K. R. B. Butler. LBM: A security framework for peripherals within the Linux kernel. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[60] D. J. Tian, N. Scaife, A. Bates, K. Butler, and P. Traynor. Making USB great again with USBFILTER. In *Proceedings of the USENIX Security Symposium*, 2016.

[61] M. Tischer, Z. Durumeric, S. Foster, S. Duan, A. Mori, E. Bursztein, and M. Bailey. Users really do plug in USB drives they find. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2016.

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

[63] W. Xu, S. Kashyap, C. Min, and T. Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[64] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim. Fuzzing file systems via two-dimensional input space exploration. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[65] M. Zalewski. American Fuzzy Lop, 2019. http://lcamtuf.coredump.cx/afl.

[66] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun. Firm-AFL: High-throughput greybox fuzzing of IoT firmware via augmented process emulation. In *Proceedings of the USENIX Security Symposium*, 2019.