NYX: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types

Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz, *Ruhr-Universität Bochum*

https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo

This paper is included in the Proceedings of the 30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the 30th USENIX Security Symposium is sponsored by USENIX.

NYX: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types

Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner and Thorsten Holz

Ruhr-Universität Bochum

Abstract

A hypervisor (also know as virtual machine monitor, VMM) enforces the security boundaries between different virtual machines (VMs) running on the same physical machine. A malicious user who is able to run her own kernel on a cloud VM can interact with a large variety of attack surfaces. Exploiting a software fault in any of these surfaces leads to full access to all other VMs that are co-located on the same host. Hence, the efficient detection of hypervisor vulnerabilities is crucial for the security of the modern cloud infrastructure. Recent work showed that blind fuzzing is the most efficient approach to identify security issues in hypervisors, mainly due to an outstandingly high test throughput.

In this paper we present the design and implementation of NYX, a highly optimized, coverage-guided hypervisor fuzzer. We show how a fast snapshot restoration mechanism that allows us to reload the system under test thousands of times per second is key to performance. Furthermore, we introduce a novel mutation engine based on custom bytecode programs, encoded as directed acyclic graphs (DAG), and affine types, that enables the required flexibility to express complex interactions. Our evaluation shows that, while NYX has a lower throughput than the state-of-the-art hypervisor fuzzer, it performs competitively on simple targets: NYX typically requires only a few minutes longer to achieve the same test coverage. On complex devices, however, our approach is able to significantly outperform existing works. Moreover, we are able to uncover substantially more bugs: in total, we uncovered 44 new bugs with 22 CVEs requested. Our results demonstrate that coverage guidance is highly valuable, even if a blind fuzzer can be significantly faster.

1 Introduction

As the digital landscape shifts more and more towards cloud computing, the security of hypervisors becomes increasingly vital for our society. At the same time, hypervisors are complex pieces of software that deal with very low-level details of the underlying hardware. To properly understand the behavior of a hypervisor both for security analysts and off-the-shelf analysis tools, we need a large amount of information on low-level hardware details. In combination with the fact that they are typically running in a highly privileged setting, this makes it difficult to properly test and analyze hypervisors for potential (security) bugs.

Fuzzing has been studied as an effective way to uncover bugs in hypervisors [7, 12, 13, 23, 24, 28, 38, 47, 48, 52, 53]. The state of-the-art methods are VDF [28] and HYPER-CUBE [48]. The former is based on isolating individual QEMU device drivers into harnesses that can be fuzzed by AFL [65], while the latter does not use any coverage feedback but is a *blind* fuzzer. Surprisingly, HYPER-CUBE still outperformed VDF on nearly all benchmarks. This is due to the fact that the device emulation in VDF is way too slow. In addition, the increased test throughput of a blind fuzzer outweighs the advantages of coverage feedback. Generally speaking, all existing fuzzers either isolate a part of the hypervisor code base into a standalone ring-3 application (harness) to obtain code coverage feedback, or are not guided by coverage feedback at all. Existing approaches have significant drawbacks, as we explain in the following.

One the one hand, isolating individual device emulators requires significant manual effort. Hence, such methods do not scale across different hypervisor implementations. Even worse, they risk introducing bugs that cannot be triggered in the original version (false positive) and more importantly, they can hide bugs that would otherwise be found (false negatives). To avoid this, one has to be very careful to reproduce the original environment of the device emulator faithfully. In addition, this approach is unable to test the parts that cannot easily be extracted. On the other hand, blind fuzzing is very efficient if a precise generator of the expected behavior is given. Unfortunately, this method fails to uncover "interesting" (especially security-critical) behaviors in complex devices and interfaces within a hypervisor. Most recent research in the general area of fuzzing has focused on coverage guidance as a feedback loop [2, 6, 10, 14, 42, 43, 62, 65]. Time and time again,

experiments have shown that coverage-guided fuzzing can drastically improve the ability to find software faults. Modern coverage-guided fuzzers can "learn" how interesting inputs look like *without* a precise specification. Even in the case where a specification is given, coverage-guided fuzzing can greatly increase the ability to test interesting behavior over blind fuzzing [1, 40, 43].

In this paper, we present NYX, a novel fuzzer that is able to test hypervisors (and, in fact, arbitrary x86 software) using coverage-guided fuzzing. As we will later see, our approach significantly outperforms HYPER-CUBE on complex devices, re-establishing the observation that coverage guidance offers significant advantages over blind fuzzing. This holds even if it comes with a significant reduction in test throughput.

Implementing coverage-guided hypervisor fuzzing without relying on manually created harnesses introduces its own set of challenges. Most significantly, in a full system setting, we typically cannot compile all relevant components with a custom compiler to obtain code coverage. Additionally, we need to be able to run the target—even in the presence of memory corruptions and crashes. Lastly, we need to be able to interact with a diverse set of interfaces.

To handle crashes and to perform introspection efficiently, we run the target component (i.e., the hypervisor we want to test) in our own hypervisor. Consequently, a large number of components are running at the same time: The host OS, running the host hypervisor, in which we run the target OS with the target hypervisor, in which the agent OS is running. Overall, this setup consist of three different operating systems and two different hypervisors. This introduces additional problems, mainly related to complexity: each of these components has a significant amount of state and seemingly non-deterministic behaviors such as timing interrupts.

To tackle all these challenges, we propose a new design that builds upon features of two existing fuzzing projects. By using Intel-PT (*Processor Trace*), we obtain code coverage information on the code running in our host hypervisor similar to KAFL [49]. Furthermore, we use a modified version of HYPER-CUBE's custom OS [48] to run inside the target hypervisor. Based on this basic setup, we built our coverage-guided hypervisor called NYX that relies on two main features. First, to handle the inherent statefulness and non-determinism of this complex stack, we develop an extremely fast snapshot restoration mechanism that allows us to reload a whole VM image in the host hypervisor many thousands of times per second. Second, to effectively generate inputs for diverse sets of interfaces, we design a new mutation engine that uses user-provided specifications. The mutation engine generates and mutates inputs that are effectively expressing highly optimized "test playbooks" of multiple interactions. These inputs are custom bytecode programs, encoded as directed acyclic graphs (DAG). The user can provide a specification to the fuzzer that describes the semantics of the bytecode and, implicitly, the shape of the graphs produced. Additionally, we use the idea of affine types, a class of typesystems that ensure each value is used at most once. This allows the specifications to properly handle cases where resources are freed or closed during testing. Using this highly flexible approach, we demonstrate adapting the fuzzer to multiple targets. We first implement a generic fuzzing specification for emulated devices, similar to state-of-the-art fuzzers. To demonstrate the strength and flexibility of our approach, we also build more precise specifications for some of the more complex devices, and even demonstrate that targeting modern paravirtualized VirtIO devices becomes possible.

Our evaluation shows that this approach consistently outperforms both coverage-guided and blind state-of-the-art hypervisor fuzzers. During the evaluation, we found 44 new bugs in current versions of hypervisors that were previously tested by state-of-the-art fuzzers. At the time of writing, 22 CVEs have been requested from which 5 vulnerabilities have already been fixed by the maintainers.

In summary, we make the following three key contributions:

- We present the design and implementation of NYX, a coverage-guided, full-system hypervisor fuzzing tool that found 44 new software faults in current hypervisors.
- We show how a highly optimized, full VM reload mechanism can be used to significantly accelerate fuzzing by reloading a whole VM image many thousands of times per second.
- We introduce the concept of an affine typed, structured mutation engine and demonstrate the benefits and flexibility of such mutations.

To foster research on fuzzing, we release NYX under an open source license at https://github.com/RUB-SysSec/nyx.

2 Technical Background

We now discuss some of the technical properties of hypervisors that make fuzzing hypervisors challenging, and introduce the techniques needed for efficient hypervisor fuzzing.

2.1 x86 Hypervisors

Hypervisors (sometimes called *Virtual Machine Monitors*) manage sharing hardware resource to Virtual Machines (VMs), also termed *guest*, within a host operating system running on a physical machine. In modern systems, this is usually implemented with the help of specific CPU features such as specialized instructions and access protection schemes that separate the memory and CPU states used by different VMs. Similar protection schemes can be used to prevent VMs directly accessing the hardware. Instead, generally speaking, emulated hardware is provided by the hypervisor. In some cases, real hardware that cannot be emulated easily can be "passed-through" (e.g., graphics cards).

2.2 Trap-`VM-Exit` and Paravirtualization

Any privileged operation (such as interaction with emulated hardware) that happens inside of the VM is trapped and control is transferred back to the hypervisor (via a VM-Exit transition). The hypervisor can emulate the privileged operation and return to the VM. This allows the hypervisor to emulate non-existing devices and to apply additional security checks. Generally speaking, the VM accesses emulated devices either via Memory-Mapped I/O (MMIO) or by using Port I/O (PIO). Hypervisor can set a trap condition for entire MMIO region. Upon access request to the MMIO region, the VM exits to the hypervisor. For port I/O operation, hypervisor uses a different strategy. Generally, to access port I/O devices, the VM has to use an `in` or `out` instruction. These instructions allow interaction with the port I/O address space and port I/O devices. Hypervisors typically configure the CPU to trap on `in`/`out` instructions. Either way, the hypervisor captures the VM-Exit, inspects the exit reason, and calls the corresponding device emulator. Device emulators are typically the largest (but not the only) attack surface of hypervisors.

Since Trap-and-Exit emulation can be slow, many modern hypervisors contain the ability to emulate hardware that does not have physical pendants, but reduce communication overhead. If the OS running inside the hypervisor is aware that it is running in a virtualized environment, it can use these special "paravirtualized" interfaces. In contrast to real devices that are typically emulated, the protocols used to interact with paravirtualized devices typically use complex structures prepared in the guests memory, containing instructions to execute whole sequences of interactions. This way, most expensive context switches can be avoided.

2.3 Challenges for Fuzzing Hypervisors

Hypervisors are a cornerstone of modern cloud infrastructures. As such, their security is of utmost importance in practice. As noted above, most previous research on fuzzing hypervisors used blind fuzzing [7, 12, 13, 23, 24, 38, 47, 52, 53]. While it is much easier to get a basic blind fuzzer to work compared to a coverage-guided fuzzer, they often struggle to explore complex devices, unless a lot of work is put into specific generators. The only exception is VDF [28], a project in which individual device emulators from QEMU were extracted and fuzzed with AFL [65] in ring-3. This helps with complex devices, however the extraction process is very labor intensive and cannot easily be performed for closed-source hypervisors.

Overall, hypervisors are challenging targets for fuzzing, as they typically run with very high privileges, making it hard to obtain *code coverage* information and to *handle crashes*. Additionally, hypervisors are *highly stateful*, as they keep all the state of each guest VM, themselves, and the emulated hardware. Consequently, during fuzzing, it is difficult to isolate the effect of one single test case (input). Previous test cases can heavily affect the result of a new test case. To prevent this, the fuzzer has to take great care to ensure that the state of the hardware is not affected by previous test cases. For example, if one test case disables some emulated hardware, subsequent test cases will not be able to interact with it. Lastly, hypervisors do not consume a single well-formed input. Instead, they provide a wide variety of different *interactive interfaces*. Some of these interfaces require the guest OS to setup complex, highly advanced structures in its own memory. Most existing general-purpose fuzzers aim at targeting programs that consume a single binary string. Now that we have identified existing challenges in coverage-guided fuzzing for hypervisors, we are going to discuss them individually.

2.3.1 Code Coverage and Handling Crashes

To handle highly-privileged code, fuzzers typically make use of virtualization to create an isolated, externally controlled environment. For instance, there are various fuzzers that are built upon KAFL [49], such as REDQUEEN [2] or GRIMOIRE [6]. These fuzzers use a modified hypervisor (KVM-PT) that allows to trace the code that runs inside of the VM. Furthermore, these fuzzers use QEMU-PT, an extension that, amongst other things, allows to decode the traces and obtain coverage information by utilizing hardware-assisted trace features such as Intel-PT (*Processor Trace*). Since the fuzzers have full control of the VM and any code running inside it, they can gracefully handle crashes of complex components such as closed-source operating systems.

Nested Virtualization Since we aim to fuzz hypervisors inside of KVM-PT, we need to enable nested virtualization. Nested virtualization describes the ability of a hypervisor, in this terminology known as *Level-0* (*L0*), to run an unmodified guest hypervisor (*L1 guest*) and all of its associated guests (*L2 guests*) in a virtual machine. Unfortunately, current x86 virtualization extensions, such as Intel VMX or AMD SVM, do not provide the nested virtualization capability in hardware. They only allow one hypervisor to be executed on one logical CPU core at the same time. Hence, the support for nested virtualization has to be implemented in software.

In modern hypervisors such as KVM, nested virtualization is implemented via emulation. Similar to emulated devices, the hypervisor traps all VMX instructions and emulates them at L0. That is, to handle a write access to a port I/O address at L2, L0 has to handle the trap first, pass on the PIO exit reason to L1, and trap the VM re-entry at L1 and emulate it to continue execution in L2. In theory, this adds a significant overhead to nested guests. However, this can be accelerated by multiple techniques [3]. KVM provides an efficient nested virtualization implementation, which we also use for NYX.

2.3.2 Fuzzing Stateful Applications

Many applications are to some extend stateful. That is, the execution of one test case is not independent of all previously executed test cases. In many instances, this statefulness is rather obvious: a target that writes the content of the test case to a file on the hard disc and fails if the file already exists is obviously stateful. However, it also manifests in much more subtle effects. For example, many standard hash table implementations use the time to derive a key used to calculate hashes. We observed that this would occasionally cause some amount of non-determinism in the code coverage, depending on whether the given keys collide or not.

In the context of hypervisors, a significant amount of state is stored in the emulated devices such as timers in the interrupt controller. These are often very relevant for the behavior of the emulated devices. Thus, for reproducible test cases, it is paramount to control the full state of the hypervisor at the beginning of the execution. This is a very hard task. Previous approaches typically tackled this problem in one of two ways: most blind fuzzers such as HYPER-CUBE tried to ignore this aspect by booting into a controlled state and then only execute a single, very long, test case, and reduce overall environment noise. However, this does not work for coverage-guided fuzzing and also causes problems when a crash is found after a long time of fuzzing. Lastly, sometimes the fuzzer might get stuck by inevitably disabling some device, rendering all future interactions pointless. The only previous coverage-guided fuzzer (VDF) tested only a small fraction of the hypervisor (such as a single device emulator) in a ring-3 QEMU process. This allowed them to restart the whole process to reset the device state. The obvious downside is that this approach does not work for large amounts of the attack surface of a typical hypervisor.

In this paper, we propose to use another approach: we implement our own fork-like mechanism for a whole VM. This has multiple advantages. First, it works independently of the target. We can use this to overcome statefulness in user-space applications, kernel components, and of course hypervisors running nested inside of our hypervisor. Additionally, as we reset the whole VM, we can also reset the emulated devices, including tricky components such as timer interrupts. This also applies for all nested VMs.

2.3.3 Fuzzing Interactive Interfaces

Most current fuzzers provide the target application with one unstructured array of bytes. While this approach is very well suited to target binary file format parsers and similar programs, it is far less useful for interactive applications that follow a well-known pattern of inputs over time (even though the format of each input might be unknown). A surprisingly large number of relevant applications actually behave like this.

Most importantly for us, hypervisors support a multitude of different interfaces that can be interacted with—each with a different format. Similarly, most kernels provide a large number of different interactions points via interfaces such as *syscalls* and *ioctls*. Lastly, even ordinary ring-3 applications, such as network services, applications with a user interface, or libraries that provide an API, require complex input formats.

```
obj = malloc_obj();
//use only after it was created
use(&obj)
//obj must not be used after free
free(obj);
```

Listing 1: Example demonstrating lifetime constraints for interactive targets.

Consider a simple API where a resource is first created, then any number of operations are performed, and lastly the resource is freed and must not be used afterwards. A similar pattern emerges with most interactive interfaces. One hypothetical test case that the fuzzer could generate is shown in Listing 1. If the fuzzer generates inputs that free non-existing objects, or accesses from objects that were not created yet, most of the generated inputs are trivially invalid, and the time spent to generate and run them is wasted. Even worse, while this is unlikely in the context of hypervisor fuzzing, they might lead to false positive crashes. For example, when fuzzing a library that provides these functions, handing an invalid pointer to the library causes a crash that is not indicating a bug in the library. To properly explore this kind of interfaces, the fuzzer should be aware of the temporal relations between creating, using, and destroying resources during input generation.

Grammar-based fuzzers (e.g., [1, 40, 43]) use context-free grammars to approximately describe inputs with such relations. However, while context-free grammars can encode the overall structure of individual interactions, they cannot readily express the temporal properties (e.g., it would not be possible to express the create/use/delete/do-not-reuse constraints explained above). On a high level, this is due to the fact that, by the definition of context-free grammars, they fundamentally only produce tree-shaped data structures. However, the data flow, resulting from chaining multiple interactions, fundamentally creates directed acyclic graphs (DAGs). This is well-known in the world of JavaScript fuzzers. Hence, many modern JavaScript fuzzers use more complex formats, which ensure that only previously initialized variables with correct types can be used [25, 63]. Additionally, current implementations of grammar-based fuzzers are typically not very effective at expressing binary data.

Another interesting example is SYZKALLER [55]. It was designed specifically to fuzz kernel interfaces via *syscalls*. These format specification typically can express initialization / use patterns. However, they are typically designed for one specific use case, and cannot express the temporal properties such as that closed resources are not to be reused later.

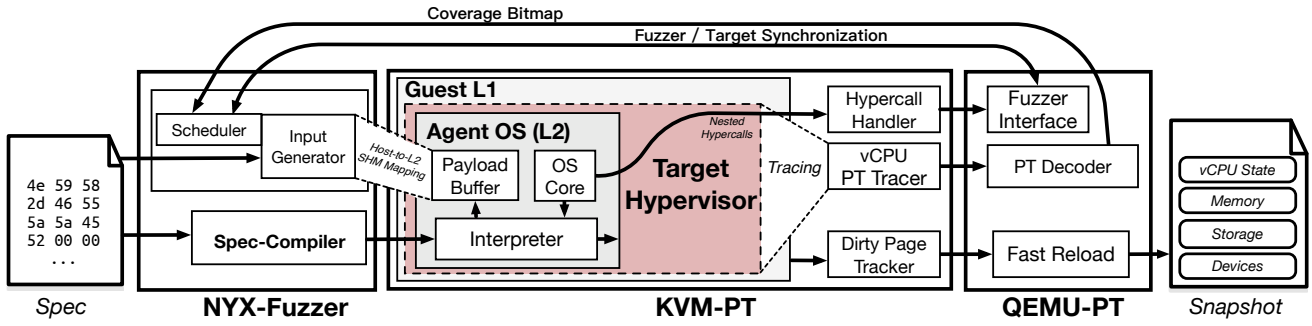

Figure 1: Overview of NYX's architecture. The architecture consists of three main components: (i) NYX's novel fuzzing engine, (ii) a highly modified version of KVM-PT which enables nested hypercall communication and hypervisor fuzzing, and (iii) a modified version of QEMU-PT to support fast snapshot reloads.

2.4 Affine Types

In many cases, the inability to express that a closed resource is not reused later on is not a big problem. In other cases, this can cause false positive crashes. For example, ignoring a library's contract will lead to false positive crashes (e.g., by passing a previously freed pointer to the library). These are not interesting to us, as they do not represent a security issue in the library, but rather simply our inability to properly use the library. One can address this issue by using *affine types*. Affine types are a class of type systems that ensure each value is used at most once. Consequently, they ensure that a resource is not reused after it was closed.

In this paper, we design a new formalism based on affine types that allows to express these kinds of constraints with a focus on versatility. The user specifies a set of opcodes. Each opcode represents a single function call, and can take any number of arguments, and return any number of values. The arguments can either be consumed or borrowed. Once a value was consumed, the fuzzer ensures that it will not be used in future opcodes. Thus, one can effectively specify sequences of affinely typed function calls. In a way, this approach is rather similar to how the programming language *Rust* uses move and borrow semantics. Using this mechanism, it becomes trivial to express well-formed interactions such as the ones seen in Listing 1. Note that this mechanism does not take away our ability to express invalid sequences if we chose to do so, it merely gives us the option to avoid them. For example, we can still express the fuzzing algorithm of AFL by only having a single handler with a vector of bytes. Consequently, this approach allows us to find all kinds of bugs that other current fuzzer can find. Yet, we can narrow down the search drastically to achieve greater coverage and find bugs faster.

3 Design

In the following, we describe the design and the reasoning behind the design choices of NYX. We start by giving an informal threat model for hypervisor security. Based on this threat model, we describe our fuzzing approach.

3.1 Threat Model

As hypervisors are used to enable provisioning of new VMs in the cloud, they are a cornerstone of the modern Internet and computing landscape. Whenever a user requires a new cloud instance, a VM is created on demand, and the user has full privileges inside the VM. To ensure scalability, many such VMs run on the same physical host and the hypervisor is the security boundary that separates different VMs. To compromise other users' VM, it suffices to escape one's own VM: once the attacker obtains hypervisor privileges, she also typically has full control over all other machines running on the same physical host. Consequently, we assume that the attacker is able to run her own kernel and tries to exploit a software vulnerability in the hypervisor.

3.2 Architecture Overview

To efficiently identify such security vulnerabilities by fuzzing hypervisors, we have to tackle a number of challenges that most current fuzzers do not address. More specifically, we need a way to explore complex interfaces with multiple back and forth interactions, while maintaining a deterministic and controlled environment that allows us to observe the test coverage. On a high level, our basic architecture is a virtual machine introspection (VMI) based fuzzer similar to KAFL and REDQUEEN, with a custom operating system similar to HYPER-CUBE used as the agent component. We introduce multiple novel techniques to make coverage-guided fuzzing applicable to highly interactive targets. An overview of NYX's architecture is shown in Figure 1.

3.3 High Performance, Coverage-Guided Fuzzing

Broadly speaking, there are two approaches to obtain the coverage information necessary to perform feedback-guided fuzzing: (i) compile-time instrumentation based approaches and (ii) binary-only based approaches. We choose to use binary-only coverage tracing, as we believe that requiring

a custom compiler toolchain severely increases the effort to obtain a working setup for fuzzing. With our setup, for example, the binaries as published by major distributors can be used with no further complications. Besides avoiding to deal with the various build systems and compilers in existence, this also ensures that we test the real software as it is delivered, with the original compiler flags and patch sets. Since we fuzz privileged code, the usual options such as dynamic binary instrumentation (DBI) are excluded. Consequently, we use Intel-PT based tracing to obtain code coverage information with only a small performance overhead.

Stable and Deterministic Fuzzing To gracefully recover from crashes in privileged code, we run the target software inside a KVM VM. As our fuzzer is outside the VM, we can restore the VM to a prior state after triggering a crash. Even beyond handling crashes, we found that fuzzing real hypervisors is very difficult: Both the target OS and the target hypervisor maintain a significant amount of state that will produce exceedingly noisy coverage traces results. To overcome this issue, we extended QEMU-PT and KVM-PT with the ability to perform very fast VM reload operations that fully restore the state of the emulated hardware—including all device state such as timing interrupts and clocks.

By using a hardware acceleration features called *Page Modification Logging* (PML), KVM can efficiently identify only those page frames in memory that need to be reset. We maintain a full copy of the original state and an additional *dirty page tracker* that allows us to quickly reset only the dirty pages. In a similar manner, we circumvent the usual device loading code used by QEMU-PT to speed up resetting the device state. This way, we overcome most of the non-determinism issues, even when tracing a whole hypervisor. Lastly, we used a modified version of HYPER-CUBE OS [48] to serve as the agent running inside of the target hypervisor. This agent communicates with our fuzzer via the host hypervisor (KVM-PT) by using hypercalls to bypass the target hypervisor.

Communication with Nested Virtualization To be able to directly communicate with the fuzzer from our agent OS, we need to provide hypercalls from the agent running in L2 directly to KVM-PT. Due to the way nested virtualization is implemented, hypercalls are passed to the host (KVM-PT) first, and later forwarded to the target hypervisor running in L1. Consequently, we implemented special hypercalls and corresponding handlers that avoid being forwarded to the target hypervisor. Additionally, the fuzzing logic and the agent need to set up a section of shared memory to efficiently pass the inputs from the logic to the agent.

3.4 Generic Fuzzing of Interactive Targets

Our fuzzing agent consumes a form of bytecode that describes the actions it should take to interact with the target hypervisor. In contrast to HYPER-CUBE OS, where the bytecode is generated randomly in a blind fashion, in our case the fuzzer generates and mutates the bytecode. To this end, the user provides specifications that describe the bytecode format. This approach is somewhat similar to grammar-based fuzzers [1, 40, 43]. However, we found that for specifying the interfaces for interactive targets, context-free grammars are not a very useful abstraction. Typed, bytecode-like specifications are much more useful, as they allow to properly refer to existing and initialized variables. Similar designs were already pioneered by JavaScript fuzzers such as FUZZILI [25] and SYZKALLER. However, instead of a highly-specialized format, we choose to develop a more general description mechanism akin to context-free grammars. In contrast to context-free grammars, our specification format allows to express types and temporal usage patterns. As a consequence, NYX can be directly applied to other targets such as kernels and ring-3 applications as well. This approach has also proven very helpful in practice by allowing an efficient test-evaluate-adapt cycle when developing specifications for new interfaces.

Affine Typed Specification Engine To allow generic fuzzing of interactive systems, we provide the user with a simple mechanism to describe a "grammar" of possible interactions. As our goal behind this fuzzing engine was to be as generic as possible, we aimed to build a mechanism as general as context-free grammars, incorporating the constraints discussed in Section 2.3.3. Specifically, we aim to express general interactions with temporal create/use/delete/do-not-reuse constraints. We achieve this by building a formalism that can be used to describe strongly typed bytecodes. We then use a custom compiler that generates C code from those bytecode specifications. Special care is taken to make sure this C code is easily embeddable into any target (no use of `malloc` etc.). Each input is represented by a directed acyclic graph (DAG). Each node is a single function call and each edge is a typed value returned by the source function and passed to the target function. Functions can take arguments either as a value or as a reference. If an argument is used as value, it can not be used later on by any other nodes. Thus, the value is effectively deleted. If the value is passed as a reference, it can later be re-used by other calls. Any node or function can take an arbitrary number of inputs both as reference and value, and return any number of values. In addition to those inputs and outputs, each function can have an additional data argument that can contain arbitrary tree-shaped data structures. We now present a small example for the previously discussed use case of opening, writing and closing files to illustrate our approach.

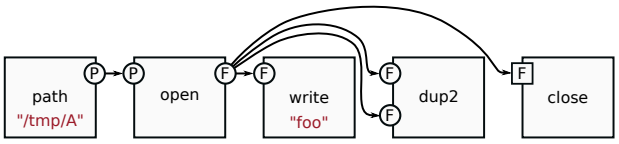

Figure 2: The graph encoding of the input shown in Listing 1. Borrowing arguments are shown as circles containing the type. Arguments that consume the value are shown as square. The tree shaped structural data attached to each node, is shown in red.

Example 1. *In this case we consider 3 opcodes:* `open`*,* `write`*, and* `close`*. The first opcode* `open(data: Vec<u8>) -> File` *has no moved or ref arguments. It only consumes a path (data string) and produces a file object. The second opcode,* `write(file: &File, data: Vec<u8>)` *takes a reference to a file object and again some data that will be written and returns no value. Any number of such write opcodes can reuse the same File object. The last opcode* `close(file: File)` *consumes the File object, and no further operations are possible on the file.*

The graph encoding the test case shown in Listing 1 can be seen in Figure 2. The input graphs generated from this bytecode specification are stored in a very compact serialized format. During fuzzing, they are stored, generated, and mutated directly in the memory shared between the fuzzer and the agent. Consequently, we avoid unnecessary copy operations and perform no allocations to generate the graphs.

The target component parses the graph stored in the shared memory. To ease this task, we automatically compile the bytecode specification to a single C header file that implements a bytecode interpreter. To compile the bytecode, the user has to provide a C implementation of the behavior of each node. As the tree-shaped data needs to be mutated, the fuzzer needs to be aware of the structure and thus, they need to be described in the specification. Consequently, the C structs representing these values can be generated automatically. On the other hand, the fuzzer does not need to modify or use the values that are created in the edges. Hence, the user can use arbitrary C types as edge types.

3.5 Applications beyond Hypervisor Fuzzing

While this paper focuses on hypervisor fuzzing, all of the techniques described here are working with any other kind of software as well. Our prototype is capable of fuzzing hypervisors, operating systems, and ring-3 applications in a unified framework. This kind of structural specification can be used to express many different kind of fuzzing scenarios. For example, in an offline experiment, we ported some of the SYZKALLER specifications to our fuzzer. We also built a harness that allows to explore the impact of fuzzing environment variables, commandline arguments, as well as, STDIN and multiple files as inputs to a ring-3 application at the same time.

4 Implementation Details

To be able to evaluate the impact of our design choices, we implemented a prototype of our design. In this section, we start by describing the steps we took to implement a high performance, coverage-guided fuzzer backend which allows us to run stable and deterministic fuzzing sessions. This includes getting coverage information, providing fast snapshot reloads, and facilitating communication between the agent and the fuzzer. We then describe the implementation details of the fuzzing frontend that generates and mutates our affine typed bytecode programs. The prototype implementation is available at https://github.com/RUB-SysSec/nyx.

4.1 Backend Implementation

The backend basically has to provide three features to the frontend: (i) It has to measure the coverage produced by a given test input, (ii) it has to provide a stable environment that can handle misbehaving targets, and (iii) it has to provide communication channels. We build upon QEMU-PT and KVM-PT as released in REDQUEEN and extended the implementation with the capabilities discussed in Section 3. We now discuss how we implemented these three components.

4.1.1 Fast Coverage

To obtain coverage information from the target hypervisor, we use the Intel-PT decoder released by Aschermann et al. [2] as a basis for our coverage measurement. However, we added some improvements on top of the original code that aim to increase the decoding performance. The decoder consists of two components: the Intel-PT parser, and the disassembler that follows the trace through a disassembled control flow graph taken from a memory snapshot. We rewrote the decoder to utilize an optimization technique known as "computed-gotos". As tracing the control flow through the disassembled control flow graph is expensive, we also introduced a caching layer. This layer can turn Intel-PT data directly into coverage information (AFL-style bitmap entries [64]) if the same trace fragments have been observed previously.

4.1.2 Fast Snapshot Reloads

Starting each test case from a clean snapshot is important to obtain deterministic coverage results. If previous test cases can affect the coverage produced by later test cases, coverage-guided fuzzing performs significantly worse. One of the major features of NYX is the ability to restore VM snapshots many thousands of times per second. To implement rapid snapshot reloads, we need to reload three components of the VM. First of all, the register state of the emulated CPU itself has to be reset. Secondly, we also need to reset all modified pages of the memory used by the virtual machine. Lastly, the state of all devices emulated in QEMU (including hard disks) needs to

be reset. We now describe the details of the mechanisms used to reset these components except for resetting the register (which is trivial).

Fast Memory Resets To create a snapshot of the VM memory, we create a snapshot file that contains a dump of the whole memory of the VM. We also implement a delta mechanism that allows to create incremental update of this snapshot file. Typically, we create one full snapshot per OS type, and then use the delta snapshots at the start of the first input. To create this snapshot, we implemented a hypercall that the agent uses to inform the fuzzer that it should create the incremental snapshot from which each test case will be started.

To quickly reset the memory of the VM, we use our own dirty page logger in KVM-PT. By default, KVM already provides the capabilities to log which pages have been dirtied since the last time the CPU entered the VM (VM-Entry). However, since KVM's technique requires us to walk a large bitmap to find all dirty pages, we extended KVM-PT with the capability to store the addresses of dirty pages in an additional stack-like buffer. This can significantly accelerate the memory restoration process, especially in cases where only a few pages have been dirtied. Additionally, we need to ensure that memory that is changed by the devices emulated by QEMU-PT is also reset. To this end, we track a second map where VM pages modified by QEMU-PT are also noted. Before we start the next execution, each page that was changed either inside the VM (as tracked by KVM-PT) or by QEMU-PT is reset to the original content from the snapshot.

Fast Device Resets Resetting the device state is a much more involved procedure compared to resetting the memory of the VM. As noted before, QEMU manages a multitude of devices. QEMU also provides a serialization/deserialization mechanism for each device, which is used to store snapshots of running VMs on the hard disk. Each device emulator provides a specification for its state in form of a specific data structure. QEMU iterates this data structure to identify fields, integers, arrays, and so on. During serialization, these fields get converted into a `JSON` string that can later be loaded during deserialization. The whole process is painfully slow, but ensures that VM snapshots can be loaded even on different machines (where the compiler may change the in-memory layout). To increase the performance, we mostly ignore these device structure specifications. Instead, we log all writes once during this process and obtain a list of all memory used by the devices. Using this list, we can now reset the device's memory from our snapshot with a series of calls to `memcpy`. It should be noted that a small subset of devices cannot be reset like this, as they require to run some custom code after each reset. We manually identified these devices in QEMU-PT and call the original deserialization routine for these devices specifically. Note that physical hardware which is used by the guest via pass-through cannot be reset, as it is not possible to access that state stored in real hardware.

Fast Disk Reset QEMU handles hard disks differently from other devices. As their state is very large—potentially larger than the available memory—the guest's hard disk content is stored on the host's hard disk in a so-called `qcow` file. To ensure we can handle targets that write files to hard disk, we create our own overlay layer on top of QEMU's `qcow` handling. During the execution, we create a hashmap that stores the content of modified sectors. This hashmap is stored in memory and uses a fixed set of buffer of pages. Every read access to the disk image is first checked against this hashmap, and then against the original `qcow` file. We place an upper limit on the number of sectors to be written during one test case to ensure that misbehaving processes do not destroy the overall fuzzing performance, similar to how AFL places limits on the time and memory used per test case. Resetting the disk image is then as easy as zeroing out the small hashmap. Critically, we do not need to overwrite the actual disk data, as removing the indices in the map suffices. Overall, this makes the reset process highly efficient and effective.

4.1.3 Nested Hypervisor Communication

To intercept and distinguish our fuzzing hypercalls from normal hypercalls directed to the target hypervisor, we implemented an additional, simple check in the host's `vmcall` handler routine. If a special value is placed in the `RAX` register by the guest, the hypercall request is handled by KVM-PT. Otherwise, this request is passed to the target hypervisor. To set up a shared memory mapping between the host and the agent OS, we need to allocate this memory region in L2 first. Using our hypercall interface, we pass all physical addresses of our allocated memory region to the host by executing a special hypercall. The host translates all guest physical addresses to host virtual QEMU-PT addresses and creates a shared memory mapping. A visualization of this procedure is given in Figure 3 ①. This shared memory region is later used by the fuzzing logic to receive messages from the agent OS or to pass new generated inputs to the agent. Prior to entering the fuzzing loop, the agent OS (L2) executes a special hypercall to create the snapshot for the fuzzing loop. The hypercall is handled by KVM-PT, and instead of relaying it to the target hypervisor (L1), another VM exit reason is passed. On the next VM entry transition from the target hypervisor to the agent OS, the snapshot will be created by QEMU-PT. This procedure is visualized in Figure 3 ②. Once the fuzzing engine has generated a new input, the snapshot is restored, and the execution is continued in the agent OS running in L2. On each transition from L2 to L1, Intel PT tracing is enabled, and disabled vice versa. This communication is shown in Figure 3 ③.

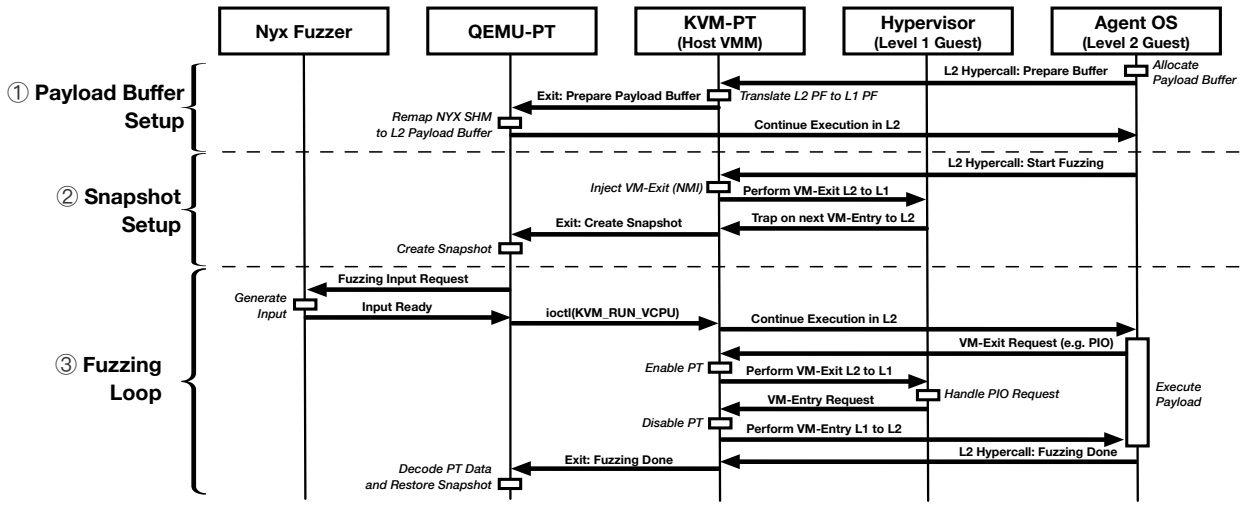

Figure 3: Overview of NYX's hypercall interaction between the various components: fuzzing logic, QEMU-PT, KVM-PT, L1 guest, and agent OS.

4.2 Fuzzing Frontend for Affine Typed Bytecode Programs

The main task of the fuzzing frontend is to generate candidate inputs and to pass the inputs to the agent OS. We implemented our own fuzzing frontend in Rust. This frontend is specifically designed to generate and mutate the bytecode inputs and we now describe the relevant details of our implementation.

4.2.1 Representation of the Bytecode

As noted earlier, we take great care in NYX to enable fast and effective input generation. Each input is stored in two arrays. The graph layout is stored in one array of `u16` integers. The additional tree-shaped data arguments are stored in a byte array. This flat, pointerless format allows fast generation and sharing via shared memory. Each node/opcode has a fixed number of arguments and outputs. We allow up to 2^{16} different node types, each with a unique ID. To encode a given node, we first push the type ID, and then one edge ID for each argument and return value. All edge IDs introduced as a return value can then be used as argument IDs for later nodes.

Example 2. *Consider the input in Listing 1. Assume the ID for the variable path is* `p` *and the ID for the variable file is* `f`*. The graph would be encoded into the following array:* `[n_new_path_id, p, n_open_id, p, f,n_write_id, f, n_dup2_id, f, f, n_close_id, f]`*. To encode the first opcode* `path=new_path("/tmp/A")`*, we first push the ID of* `new_path` (`n_new_path`)*, then we push the ID of the only return value (*`p`*). Note that we ignore the additional data argument for now. We encode the remaining nodes in the same fashion by pushing the node ID and then the edge IDs for each argument or return value.*

The additional tree-shaped/binary data attached to each node is stored in a second buffer. As we know what kind of data is attached to each node, the values are simply concatenated. For binary data that is dynamically sized (e.g., strings or byte vectors), the size is prefixed.

Example 3. *When considering the graph representing the input in Listing 1, we would encode the binary data used as additional arguments to* `new_path` *and* `write` *as:* `[7,"/tmp/A\0",4,"foo\0"]`*. Here, 7 and 4 are the lengths of the following strings. The strings are stored as raw bytes.*

4.2.2 Generating Bytecode Interpreters

To interpret the results, we automatically transpile the specifications into a single C-header interpreter for the bytecode. The user simply has to fill in the functions for each opcode. This interpreter uses the information provided in the specification to iterate both memory buffers, keep track of the values that are passed along the edges in the graph, and call the user-provided functions for each node. In our example, we used HYPER-CUBE and linked this interpreter into HYPER-CUBE to produce a fuzzing agent for NYX.

5 Evaluation

We use our prototype implementation of NYX to evaluate the results of our design choices. In particular, we aim to answer the following five research questions:

- **RQ 1.** How does NYX compare to state-of-the-art approaches such as HYPER-CUBE and VDF?
- **RQ 2.** Does coverage guidance improve generative fuzzing?

- **RQ 3.** What are the performance gains provided by the structured mutation engine?
- **RQ 4.** What is the performance impact of fast reloads?
- **RQ 5.** Can NYX find previously unknown vulnerabilities in well-tested parts of hypervisors?

As we will see, NYX drastically outperforms VDF on almost all devices and performs comparable or better than HYPER-CUBE on all but one device. In four cases, NYX drastically outperforms HYPER-CUBE, using specifications that are chosen to mirror the behavior of HYPER-CUBE. If we use properly customized specifications, the results are improved further. We were able to uncover 44 new bugs, many of which represent serious security issues. Using the fast snapshot restoration allows us to reset the whole VM with a performance characteristics comparable to AFL's fork server.

5.1 Evaluation Setup

All experiments were performed on Intel Xeon Gold 6230 CPUs. Each machine had 40 physical cores and 192GB of memory as well as an SSD. We pinned each fuzzer to one physical core and did not use hyper-threading. Each experiment was repeated ten times to obtain statistically significant results [32]. In all plots, the lines mark the median of the ten runs, and the shaded area display the best and worst run respectively. We targeted QEMU `5.0.0` and bhyve `12.1-RELEASE`. VDF was evaluated on older versions of QEMU and we can only compare with the numbers reported in the paper. While this slightly reduces the strength of the comparison to VDF, we believe it is much more meaningful to fuzz modern, well-tested software. Additionally, VDF was already shown to be significantly slower than HYPER-CUBE. We also repeat the HYPER-CUBE experiments using the newer version of QEMU and observe very similar results.

5.2 Fuzzing Device Emulators

In the first experiment, we compare NYX against HYPER-CUBE and VDF to answer **RQ 1.**. We used the open-source version of HYPER-CUBE, but unfortunately VDF is not openly available. Therefore, we follow the authors of HYPER-CUBE and compare against the numbers published in the VDF paper. While the authors of VDF evaluated for approximately 60 days, the authors of HYPER-CUBE managed to beat VDF in both terms of coverage found and bugs found in only ten minutes. As we are not able to reproduce the exact hardware that VDF used for their experiments, we too, choose to drastically reduce the time for evaluation. However, since NYX performs many complex operations such as minimizing new inputs found, we also extended the experiments to 24 hours each. To compare fairly against HYPER-CUBE, we created specs that very closely represent HYPER-CUBE's operations (NYX-Legacy) and used both fuzzers to target QEMU/KVM. As we will later see, NYX can perform even better using custom specifications for specific targets. To demonstrate the impact of specs on NYX's performance, we also added another complex device (XHCI).

We ran the target VM with Gcov, and restarted it every 10 minutes or after each crash, to dump the coverage. This way, we could obtain coverage plots over time, as otherwise only the final coverage could be reported. The coverage found over time is shown in Figure 4. Note that this figure only contains those devices where non-trivial differences in performance where observed. The full set of results can be found in the Appendix. We also display the overall results in Table 1. As can be seen, our approach easily surpasses VDF in all (but two) scenarios. After manual inspection, we believe that the difference in coverage between VDF and NYX is due to the fact that the code changed since VDF performed their experiments and that the observed difference does not represent a real difference in performance. Compared to the blind fuzzer HYPER-CUBE, we see that in all but six cases, NYX and HYPER-CUBE perform identical or nearly identical (Though NYX might sometimes need a few more minutes to reach the same coverage). Since many device emulators have rather simple control flows (many do not even contain loops), this is not entirely surprising.

However, on the more complex devices, the advantages of coverage-guided fuzzing begin to show. Over a reasonable time frame (typically the first few hours), the advantages begin to outweigh the additional cost. This effect is particularly pronounced in the complex examples where HYPER-CUBE stops making any progress very early. Hence, NYX produces drastically more coverage on four of the six devices, which also answers **RQ 2.**. On the other two devices (SoundBlaster and E1000), HYPER-CUBE performs better. We investigate SoundBlaster and believe this is due to interrupt handlers which are triggered after specific timeout interrupts occur. These timeouts are never triggered due to the short time span of our test cases and the subsequent VM resets. We believe a similar mechanism affects our performance on E1000.

5.3 Structured and Coverage

To further substantiate the impact of proper structure definitions (**RQ 3.**), we studied the Intel specifications for the eXtensible Host Controller Interface (XHCI) and built specifications that specifically target this device. Besides the usual MMIO operations that are required to actually interact with the device emulators, this also includes setting up complex data structures in the guest's memory. For example, the XHCI USB Host Controller uses multiple linked list for different purposes to be handled. The MMIO access then only writes the pointer to the head of the list, and the device iterates the list on its own. We created a specification that allows to setup such memory structures in the guest. Using this specification, we performed another set of runs. To answer **RQ 3.**, we compare the results of the legacy specification that emulates the

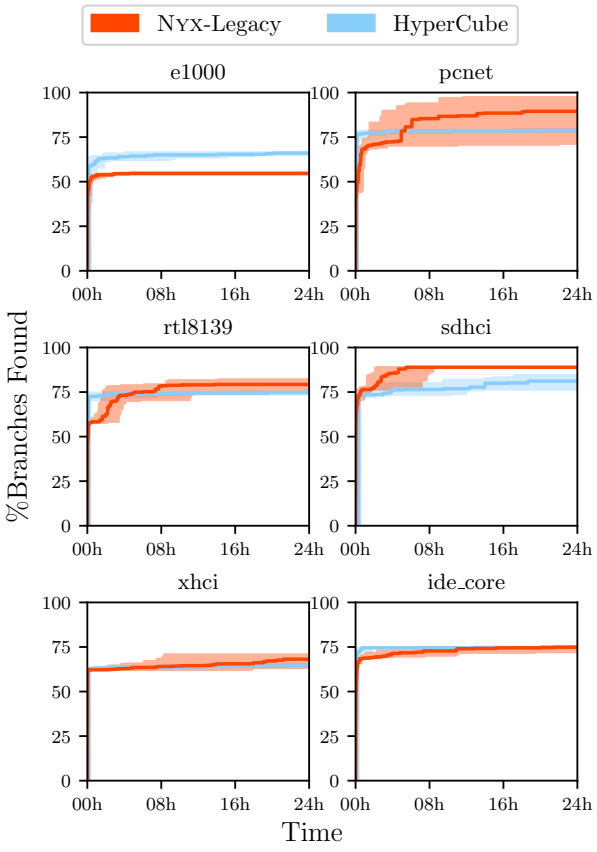

Figure 4: Overview of the median, best, and worst branch coverage across 10 experiments. We only display the 6 devices with relevant differences between Nyx using the legacy spec and Hyper-Cube.

behavior of Hyper-Cube and our more detailed specification. The results can be seen in Figure 5. As the experiment results show, using more detailed specifications drastically increases the performance of the fuzzer. While in the previous experiment, coverage guidance on helped in the long term, and Hyper-Cube dominated for the first hour of fuzzing, here we can see that such specifications are showing drastic improvements in performance from the very first moment on.

To further confirm our claim that coverage guidance is in fact helpful (**RQ 3.**), we perform a second experiment comparing coverage-guided and non-coverage-guided fuzzing with these more detailed specifications. As we could not integrate them into Hyper-Cube, we instead used Nyx, but disabled the coverage guidance mechanism. Thus, we compare a blinded version of Nyx with the normal, coverage-guided version of Nyx. This allows us to specifically identify the impact of coverage guidance in the presence of structured fuzzing. The result are also shown in Figure 5. As we can see, without coverage guidance, the more complex specifications added very little coverage. However, in combination with coverage guidance, the ability to find deeper code paths increased massively. While it might seem somewhat surprising that the

Table 1: Branch coverage by Nyx using a legacy specification and Hyper-Cube in 24 hours (compared to VDF with multi-month experiments). If the differences between Nyx and Hyper-Cube are statistically relevant with p<0.01 according to a Mann-Whitney-U test, the better result is printed bold. Δ denotes the difference in percentage points between Nyx and Hyper-Cube.

	VDF	Hyper-Cube	Nyx	
Device	Cov	Cov	Cov	Δ
AC97	53.0%	**100.00%**	98.92%	-1.62
CS4231a	56.0%	74.76%	74.76%	-
ES1370	72.7%	91.38%	91.38%	-
Intel-HDA	58.6%	**79.17%**	78.33%	-0.84
SoundBlaster	81.0%	**83.80%**	81.34%	-2.46
Floppy	70.5%	**84.51%**	83.10%	-1.41
Parallel	42.9%	38.61%	38.61%	-
Serial	44.6%	73.76%	73.76%	-
IDE Core	27.5%	74.87%	74.69%	-0.18
EEPro100	75.4%	83.82%	83.82%	-
E1000	81.6%	**66.08%**	54.55%	-11.53
NE2000 (PCI)	71.7%	71.89%	71.89%	-
PCNET (PCI)	36.1%	78.71%	**89.49%**	+10.78
RTL8139	63.0%	74.68%	**79.28%**	+4.60
SDHCI	90.5%	81.15%	**88.93%**	+7.78
XHCI	-	64.70%	**69.93%**	+5.23

specifications offer so little without coverage guidance, this can actually be explained by the fact that a significant number of integer parameters need to be chosen properly to generate interesting structures from the specification. Without the coverage feedback, picking the right shape and the right values is exceedingly unlikely.

5.4 Fast Snapshot Reload Performance

To quantify the performance impact of our fast VM reloads, and to answer **RQ 4.**, we perform two experiments on the reload performance. Since reloading the register- and device-state is independent of the fuzzing target, the reload performance is primarily determined by the number of dirty pages that need to be restored. As our fuzzer is also able to fuzz ring-3 applications, we created a small test application that dirties a given number of pages on each execution. To inspect the behavior, we perform measurements with different numbers of dirty pages. The results can be seen in Figure 6. Device reloads create an additional performance cost, even when no pages need to be reset. As expected, as more and more pages are reset, the performance gets gradually worse. Overall, for large resets we approach the memory throughput.

To put these numbers in relation to similar mechanisms, we also compare with AFL's *forkserver* and QEMU's normal snapshot restoration mechanism. We use the same ring-3 application as before and note the number of executions AFL's forkserver achieves depending on the number of dirty pages. As expected, for very small deltas, the forkserver is slightly more effective, yet as the number of modified pages grows, the performance differences shrink. In contrast, QEMU always restores the full snapshot. Hence, the performance remains

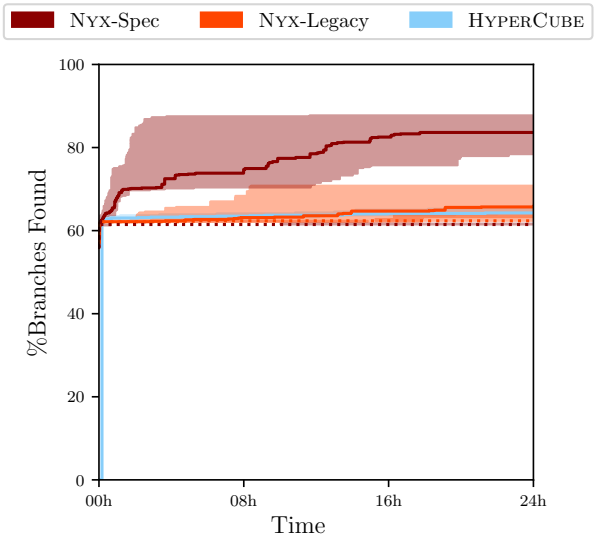

Figure 5: Comparing the code coverage found on XHCI by the legacy specification and more detailed specifications across 10 runs. The dotted lines show the performance that NYX achieved, if used as a blind fuzzer.

constant, until running the application which accesses large amounts of memory begins to affect the performance.

For realistic workloads, our snapshots reloads are multiple orders of magnitude faster than QEMU's internal snapshot restoration mechanism, and we are able to perform about 60% as many test cases compared to AFL's forkserver. While obtaining similar performance, NYX reloads perform a lot more tasks than the fork server: we observe that when the target only dirties ten pages, we reload almost a 100 pages in the kernel. We also reset all of the devices' state, including hard discs. This also shows up in the number of pages reloaded: When fuzzing more complex targets that modify the disc state, this becomes fundamental.

When using NYX in offline experiments, we observed that fuzzing programs like *Bash* with AFL is very hard: great care has to be taken to ensure that script interpreters do not overwrite or remove any relevant files. Similarly, they do tend to quickly fill up the disc with junk. All of these issues are mitigated by the snapshot restoration process. Lastly, we observed similar performance when fuzzing target programs under Windows. This is a significant advantage, as Windows does not offer the performance gains of a forkserver, which significantly slows down the fuzzing process.

5.5 New Vulnerabilities

Besides analyzing the coverage, we also used our fuzzer to find novel bugs. To this end, we picked all the devices from Section 5.3 as well as some additional ones that we could not use to compare coverage for various reasons. For example, we evaluated various VirtIO devices on bhyve such as (`virtio_blk`, `virtio_net`, and `virtio_serial`) that are not readily supported by HYPER-CUBE.

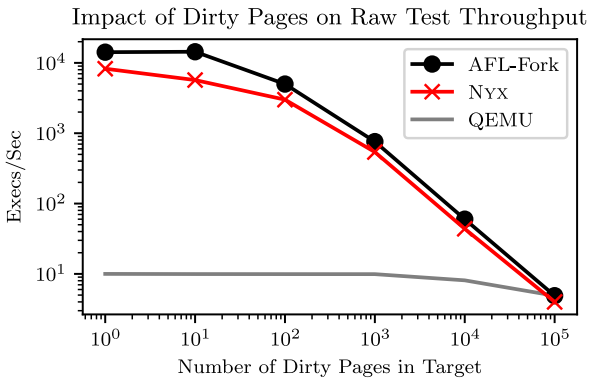

Figure 6: Comparing raw executions per second for targets that dirty *N* pages, with an AFL forkserver, QEMU's `loadvm` snapshot restore functionality, and NYX's fast full-system reloads.

Table 2: Overview of vulnerabilities found by NYX in our targets.

Hypervisor	Type	#Bugs
QEMU	Use-After-Free (Write)	1
	Heap-based Buffer Overflow (Write)	1
	Stack Overflow	1
	Infinite Loop	1
	Segmentation Fault	3
	Abort/Assertion	9
BHYVE	Segmentation Fault	14
	Infinite Loop	1
	Assertion	13

During the evaluation, we identified 44 manually verified, unique crashes. An overview of the types of crashes found is shown in Table 2, a full list of the crashes with more details on the exploitability can be found in the Appendix. All bugs were reported in a coordinated way and CVEs were requested for all memory corruption issues. Many of the bugs were fixed and some are still being actively discussed on the maintainers' mailing lists. Even after QEMU and bhyve were fuzzed by HYPER-CUBE, NYX finds a significant number of serious issues in both hypervisors, answering **RQ 5.**.

In the following, we provide a more in-depth look at some of the bugs found.

Case Study: bhyve Infinite `XHCI TRB` Loop. The `XHCI` device implementation of the bhyve hypervisor is vulnerable to a denial-of-service attack via an infinite loop in the host. According to the `XHCI` specification, the guest's driver has to setup and maintain multiple memory regions in its physical memory to communicate with the `XHCI` USB controller and its attached USB devices. A set of data structures called TRBs (Transfer Ring Blocks) are used for bi-directional communication. Link TRBs are used to link multiple memory chunks together to implement rings across non-continues memory regions. By configuring a crafted TRB ring array containing a Link TRB pointing to itself, the emulator gets stuck in an infinite loop in the function `pci_xhci_trb_next`.

Case Study: QEMU `EE100Pro` Stack Overflow via Recursive DMA Requests. NYX uncovered a stack-overflow vulnerability in QEMU's DMA mechanism used by the EE100Pro device emulator. The EE100Pro device relies on the CU (Command Unit) and RU (Receive Unit) to send commands and receive data from its guest. By configuring the CU `base` and `offset` register to point to its own PCI MMIO BAR with a specific offset and a write accesses to the command register thereafter, the device emulator will perform a DMA write access to the same MMIO register and initiate the same DMA access again. This will lead to stack exhaustion and a crash, which can be exploited by a malicious guest.

Case Study: QEMU `SDHCI` Heap-based Buffer Overflow. The `SDHCI` device performs read and write operations in blocks. The size of these blocks can be set with the `SDHC_BLKSIZE`. Each read and write command moves the `data_count` cursor of the data buffer `fifo_buffer` forward until the `blksize` is reached. For larger data, the `SDHC_SYSAD` command allows multi-block transfers and starts at the `data_count` cursor.

When a new block size is set with the `SDHC_BLKSIZE` command, the `data_count` cursor is not reset and the block size is also not checked against `0`. This allows an attacker to first set a high block size, move the cursor at an arbitrary position, then set the block size to `0` and issue a multi-block transfer. The length is calculated as `0 - data_count`, which results in an arbitrary heap out-of-bounds write up to the size of the `uint16_t` or the maximum buffer size, whatever is lower.

Coordinated Disclosure. In total, we reported 44 bugs to the maintainers. 7 security issues were directly reported to and acknowledge by the QEMU security team according to their security process. Currently, the QEMU security team assigned four CVEs (CVE-2020-25084, CVE-2020-25085, CVE-2020-25741, CVE-2020-25743) for fixed and published issues. While in general it is hard to evaluate the exact security impact of bugs found without actually spending time to write an exploit, we believe that most memory corruption issues could be exploited under the right circumstances. Another 15 security issues in bhyve were reported to the FreeBSD security team with pending CVEs assigned by MITRE.

Other non-critical security issues, such as assertion failures, were publicly reported through `launchpad.net` for QEMU bugs (#1883728, #1883729, #1883732, #1883733, #1883739, #1525123, #697510, #1681439, #1810000) and the FreeBSD bug tracker for bhyve findings.

6 Related Work

In recent years, fuzzing has shown exceptional results on uncovering bugs in software systems. This trend was started by a coverage-guided fuzzer named AFL [65]. To improve upon AFL, a large number of researcher tried to improve AFL's input mutation algorithm [1, 2, 27, 40, 43] and its ability to identify bugs [4, 5, 31, 37, 39, 58]. Other approaches focused on improving feedback mechanism in coverage-guided fuzzers [16, 19, 30, 33, 57]. Additionally, improved scheduling algorithms have been researched extensively [8–11, 46, 59]. A more in depth discussion on various recent advances in fuzzing can be found in Manès et al.'s overview [35].

Next to generic improvement over AFL's design and implementation, some research proposed a hybrid software testing method which combines feedback fuzzing with concolic execution [20–22, 26, 36, 50, 56, 62, 66]. Similar to the concolic execution based approaches, others tried to improve fuzzing by adding taint tracking [14, 45]. Lastly, various researchers focused on improving the raw throughput of various components of modern fuzzers [51, 61].

Snapshots were already used in the context of testing. AFL's fork server can be seen as a primitive ring-3 snapshot mechanism. Dong et al. used snapshots for testing Android apps [17]. However, their approach takes approximately nine seconds to restore a single snapshot, rendering them infeasible for our purposes. Recently, Falk used a similar mechanism to quickly reset the memory of VMs [18], however that implementation does not support emulated devices.

To apply fuzzing to a wider set of targets, coverage-guided fuzzers for ring-0 targets were developed [29, 41, 49, 55, 60]. Additionally, some recent research expanded the fuzzing approach into the IoT and embedded systems domain [15, 34]. Beyond ring-0, fuzzing was also applied to hypervisors [23, 28, 48, 52]. For example, VDF [28] implements a coverage-guided hypervisor fuzzing approach. Recently, Schumilo et al. introduced HYPER-CUBE, a blind fuzzer for hypervisors [48]. Various researchers also implemented other blind hypervisor fuzzers [12, 23, 38, 47].

7 Discussion

In this paper, we describe an approach to fuzz hypervisors using coverage guidance. The recent success of HYPER-CUBE put the viability of coverage-guided fuzzing for hypervisors into question. Our evaluation shows that coverage-guided fuzzing is indeed working as expected. Consequently, the fundamental problem behind VDF is not the overhead of coverage-guided fuzzing per se, but their implementation. A properly implemented and sufficient optimized whole-system fuzzer design is capable of outperforming HYPER-CUBE. However, to this end, current fuzzers need to apply a set of changes: first, we need a way to obtain code coverage from all code regardless of the protection ring it is running under. Second, they need to handle the high non-determinism using fast snapshot reloads. Last, the mutator needs to understand the interactive nature of the inputs. As the authors of HYPER-CUBE already noted, coverage-guided fuzzing adds a lot of value when fuzzing more complex devices.

While our approach is versatile and much faster and easier to use than VDF, and in many cases outperforms even HYPER-CUBE, it also has some drawbacks: it is slightly more complex to setup than HYPER-CUBE, as the target hypervisor needs to run inside KVM-PT. For most hypervisors, this is not particularly challenging, as KVM-PT fully supports nested virtualization. However, using nested virtualization allows us to easily recover from crashes. HYPER-CUBE needs to restart the whole process after each crash, and typically has a very hard time to overcome early crashes triggered by overzealous assert statements.

Creating Specifications Additionally to running the target hypervisor in a nested configuration, the user also needs to provide a specification. While we have demonstrated that even the uninformed specification that closely mirrors HYPER-CUBE's behavior is already quite useful, most of the times a more precise specification is helpful. Designing a specification is quite similar to designing a grammar for well-known fuzzers such as NAUTILUS [1], PEACH [54], or SULLEY [44]. The biggest part of the effort is not to produce the specification, but to obtain a sufficient understanding of the target. In our case, we spent about two days on our most complex specification. Understanding the structures required to perform VirtIO took by far the biggest amount of work. Writing the specification based on this understanding took only a very small fraction of the time (around two hours).

Long-Running Interactive Fuzzing Our fuzzer still maintains one aspect of current coverage-guided fuzzers: each small input is tested in isolation after a mutation. It would be very interesting to explore long-running interactive fuzzing: instead of generating small inputs outside of the VM, a large stream could be generated from a given seed inside the target VM. While the original HYPER-CUBE logic generates interactions within the VM, KVM-PT would observe the coverage from the outside until new coverage is found.

8 Conclusion

In this paper, we introduced an approach to fuzz highly complex and stateful interactive targets. While this paper focuses on hypervisor fuzzing as one example of such systems, all the techniques introduced here work as well to fuzz any other kind of software. We are convinced that both super fast, full VM reloads and structured fuzzing of interactive applications are valuable additions to current fuzzers, no matter of the target. We have demonstrated how coverage-guided fuzzing can beat blind fuzzing, even when the blind fuzzer is able to produce far more interactions per second. While blind fuzzers such as HYPER-CUBE are conceptually much simpler, and—if implemented properly—can provide a much larger number of such interactions, they will struggle to sufficiently test the less common parts of the application. Using fast snapshots provides near-perfect reproducibility. By using coverage guidance, the hard-to-hit parts of the target are explored much more thoroughly. As a consequence, we find more bugs and in most cases more coverage while using the same specification. Similarly, using our affinely typed bytecode specification format, it becomes simple to generate much more complex specifications for any given use case, further increasing the coverage and number of bugs found.

Acknowledgements We would like to thank our shepherd Byron Williams and our anonymous reviewers for their valuable feedback. This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2092 CASA – 390781972. In addition, this work was supported by the European Union's Horizon 2020 Research and Innovation Programme (ERC Starting Grant No. 640110 (BASTION) and 786669 (REACT)). The content of this document reflects the views only of their authors. The European Commission/Research Executive Agency are not responsible for any use that may be made of the information it contains.

References

[1] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for Deep Bugs with Grammars. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[3] Muli Ben-Yehuda, Michael D Day, Zvi Dubitzky, Michael Factor, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[4] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[5] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing. 2020.

[6] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and

Thorsten Holz. GRIMOIRE: Synthesizing Structure while Fuzzing. In *USENIX Security Symposium*, 2019.

[7] Sören Bleikertz. XenFuzz. https://www.openfoo.org/blog/xen-fuzz.html. Accessed: October 6, 2020.

[8] Marcel Böhme, Valentin JM Manès, and Sang Kil Cha. Boosting Fuzzer Efficiency: An Information Theoretic Perspective. In *esec-fse*, 2020.

[9] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

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

[11] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *IEEE Symposium on Security and Privacy*, 2015.

[12] Amardeep Chana. MWR-Labs: Ventures into Hyper-V - Fuzzing hypercalls. https://labs.mwrinfosecurity.com/blog/ventures-into-hyper-v-part-1-fuzzing-hypercalls/. Accessed: October 6, 2020.

[13] Amardeep Chana. Viridian Fuzzer. https://github.com/mwrlabs/ViridianFuzzer. Accessed: October 6, 2020.

[14] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *IEEE Symposium on Security and Privacy*, 2018.

[15] 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 *USENIX Security Symposium*, 2020.

[16] S. Dinesh S. Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *IEEE Symposium on Security and Privacy*, 2020.

[17] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. Time-travel Testing of Android Apps. In *icse*, 2020.

[18] Brandon Falk. Chocolate Milk. https://github.com/gamozolabs/chocolate_milk. Accessed: October 6, 2020.

[19] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: Path Sensitive Fuzzing. In *IEEE Symposium on Security and Privacy*, 2018.

[20] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[21] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[22] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *Symposium on Network and Distributed System Security (NDSS)*, 2008.

[23] Mikhail Gorobets, Oleksandr Bazhaniuk, Alex Matrosov, Andrew Furtak, and Yuriy Bulygin. Attacking hypervisors via firmware and hardware. *Black Hat USA*, 2015.

[24] Mikhail Gorobets, Oleksandr Bazhaniuk, Alex Matrosov, Andrew Furtak, and Yuriy Bulygin. Attacking hypervisors via firmware and hardware. *Black Hat USA*, 2015.

[25] Samuel Groß. FuzzIL: Coverage Guided Fuzzing for JavaScript Engines. Master's thesis, Karlsruhe Institute of Technology, Karlsruhe, Germany, 2018.

[26] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *USENIX Security Symposium*, 2013.

[27] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[28] Andrew Henderson, Heng Yin, Guang Jin, Hao Han, and Hongmei Deng. VDF: Targeted Evolutionary Fuzz Testing of Virtual Devices. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2017.

[29] Jesse Hertz and Tim Newsham. Project Triforce: Run AFL on Everything! https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/. Accessed: October 6, 2020.

[30] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. INSTRIM: Lightweight Instrumentation for Coverage-guided Fuzzing. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, 2018.

[31] Yuseok Jeon, Wookhyun Han, Nathan Burow, and Mathias Payer. FuZZan: Efficient Sanitizer Metadata Design for Fuzzing. In *USENIX Annual Technical Conference*, 2020.

[32] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[33] Li, Yuekang and Chen, Bihuan and Chandramohan, Mahinthan and Lin, Shang-Wei and Liu, Yang and Tiu, Alwen. Steelix: Program-state Based Binary Fuzzing. In *Joint Meeting on Foundations of Software Engineering*, 2017.

[34] Dominik Maier, Lukas Seidel, and Shinjo Park. BaseSAFE: Baseband SAnitized Fuzzing through Emulation. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2020.

[35] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. In *IEEE Transactions on Software Engineering*, 2019.

[36] David Molnar, Xue Cong Li, and David Wagner. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs. In *USENIX Security Symposium*, 2009.

[37] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities. In *International Conference on Software Engineering (ICSE)*, 2020.

[38] Tavis Ormandy. An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments. In *CanSecWest 2007*, 2007.

[39] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *usenix-security*, 2020.

[40] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Zest: Validity Fuzzing and Parametric Generators for Effective Random Testing. *arXiv preprint arXiv:1812.00078*, 2018.

[41] Hui Peng and Mathias Payer. USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation. In *USENIX Security Symposium*, 2020.

[42] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: Fuzzing by Program Transformation. In *IEEE Symposium on Security and Privacy*, 2018.

[43] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. Smart Greybox Fuzzing. *arXiv preprint arXiv:1811.09447*, 2018.

[44] Aaron Portnoy and Pedram Amini. Sulley. https://github.com/OpenRCE/sulley. Accessed: October 6, 2020.

[45] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[46] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan M Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *USENIX Security Symposium*, 2014.

[47] Microsoft Security Research and Defense. Fuzzing para-virtualized devices in Hyper-V. https://blogs.technet.microsoft.com/srd/2019/01/28/fuzzing-para-virtualized-devices-in-hyper-v/. Accessed: October 6, 2020.

[48] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[49] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *USENIX Security Symposium*, 2017.

[50] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[51] Robert Swiecki and Anestis Bechtsoudis. Security oriented fuzzer with powerful analysis options. https://github.com/google/honggfuzz. Accessed: October 6, 2020.

[52] Jack Tang and Moony Li. When Virtualization Encounters AFL. https://www.blackhat.com/docs/eu-16/materials/eu-16-Li-When-Virtualization-Encounters-AFL-A-Portable-Virtual-Device-Fuzzing-Framework-With-AFL-wp.pdf. Accessed: October 6, 2020.

[53] Microsoft Virtualization Security Team. Fuzzing para-virtualized devices in Hyper-V. https://blogs.technet.microsoft.com/srd/2019/01/28/fuzzing-para-virtualized-devices-in-hyper-v/. Accessed: October 6, 2020.

[54] Peach Tech. Peach. http://www.peachfuzzer.com/. Accessed: October 6, 2020.

[55] Dmitry Vyukov. syzkaller: Linux syscall fuzzer. https://github.com/google/syzkaller. Accessed: October 6, 2020.

[56] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy*, 2010.

[57] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. In *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[58] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. Memlock: Memory usage guided fuzzing. In *International Conference on Software Engineering (ICSE)*, 2020.

[59] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[60] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data Race Fuzzing for Kernel File Systems. In *IEEE Symposium on Security and Privacy*, 2020.

[61] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing New Operating Primitives to Improve Fuzzing Performance. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[62] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium*, 2018.

[63] Soyeon Park Wen Xu Insu Yun and Daehee Jang Taesoo Kim. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *IEEE Symposium on Security and Privacy*, 2020.

[64] Michael Zalewski. Technical whitepaper for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed: October 6, 2020.

[65] Michał Zalewski. american fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed: October 6, 2020.

[66] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

Appendix

A List of Bugs

Table 3: Bugs found by NYX in our targets. QEMU CVEs were assigned by the maintainers if the issues was fixed. The remaining issues marked as *requested* are still under investigation. The BHYVE maintainers have not yet assigned CVEs, and CVEs were reserved by MITRE instead.

Hypervisor	Type	CVE
QEMU	Use after free (write) in `usb_process_one`	CVE-2020-25084
	Heap buffer overflow (write) in `sdhci_sdma_transfer_multi_blocks`	CVE-2020-25085
	Stack overflow in `eepro100_write_command`	requested
	Infinite loop in `start_xmit`	requested
	Segmentation fault in `blk_inc_in_flight`	CVE-2020-25741
	Segmentation fault in `pci_change_irq_level`	CVE-2020-25742
	Segmentation fault in `blk_bs`	CVE-2020-25743
	Abort in `xhci_alloc_device_streams`	-
	Assertion in `address_space_unmap`	-
	Assertion in `usb_packet_copy`	-
	Assertion in `xhci_find_stream`	-
	Assertion in `xhci_kick_epctx`	-
	Assertion in `usb_ep_get`	-
	Assertion in `lsi_do_dma`	-
	Assertion in `ide_cancel_dma_sync`	-
	Assertion in `ide_dma_cb`	-
BHYVE	Infinite loop in `pci_xhci_trb_next`	RESERVED
	Segmentation fault in `pci_xhci_cmd_eval_ctx`	RESERVED
	Segmentation fault in `pci_xhci_cmd_reset_device`	RESERVED
	Segmentation fault in `pci_xhci_cmd_address_device`	RESERVED
	Segmentation fault in `pci_xhci_complete_commands`	RESERVED
	Segmentation fault in `pci_xhci_insert_event` at `pci_xhci.c`	RESERVED
	Segmentation fault in `pci_xhci_insert_event` at `pci_xhci.c`	RESERVED
	Segmentation fault in `pci_xhci_insert_event` at `pci_xhci.c`	RESERVED
	Segmentation fault in `ahci_handle_slot` at `pci_ahci.c`	RESERVED
	Segmentation fault in `ahci_handle_slot` at `pci_ahci.c`	RESERVED
	Segmentation fault in `vq_has_descs`	RESERVED
	Segmentation fault in `vq_kick_disable`	RESERVED
	Segmentation fault in `pci_vtcon_notify_tx`	RESERVED
	Segmentation fault in `vq_endchains`	RESERVED
	Segmentation fault in `pci_vtcon_control_tx`	RESERVED
	Assertion in `pci_xhci_cmd_config_ep`	-
	Assertion in `pci_xhci_cmd_reset_ep` at `pci_xhci.c`	-
	Assertion in `pci_xhci_cmd_reset_ep` at `pci_xhci.c`	-
	Assertion in `pci_xhci_cmd_set_tr` at `pci_xhci.c`	-
	Assertion in `pci_xhci_cmd_set_tr` at `pci_xhci.c`	-
	Assertion in `pci_xhci_get_dev_ctx`	-
	Assertion in `ahci_build_iov`	-
	Assertion in `pci_vtblk_proc` at `pci_virtio_block.c`	-
	Assertion in `pci_vtblk_proc` at `pci_virtio_block.c`	-
	Assertion in `pci_vtblk_proc` at `pci_virtio_block.c`	-
	Assertion in `pci_vtblk_proc` at `pci_virtio_block.c`	-
	Assertion in `pci_vtblk_proc` at `pci_virtio_block.c`	-
	Assertion in `pci_vtblk_proc` at `pci_virtio_block.c`	-

B Coverage Plots

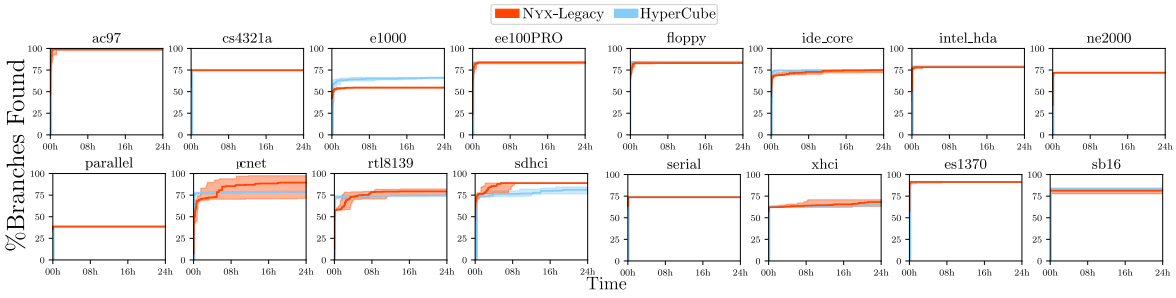

Figure 7: The median, best, and worst branch coverage of 10 runs (24h each).