HyperFuzzer: An Efficient Hybrid Fuzzer for Virtual CPUs

Xinyang Ge
Microsoft Research
xing@microsoft.com

Ben Niu
Microsoft
beniu@microsoft.com

Robert Brotzman
Penn State University
rcb44@cse.psu.edu

Yaohui Chen
Facebook
yaohway@gmail.com

HyungSeok Han
KAIST
hyungseok.han@kaist.ac.kr

Patrice Godefroid
Microsoft Research
pg@microsoft.com

Weidong Cui
Microsoft Research
wdcui@microsoft.com

ABSTRACT

In this cloud computing era, the security of hypervisors is critical to the overall security of the cloud. In particular, the security of CPU virtualization in hypervisors is paramount because it is implemented in the most privileged CPU mode. Blackbox and graybox fuzzing are limited to finding shallow virtual CPU bugs due to its huge search space. Whitebox fuzzing can be used for systematic analysis of CPU virtualization, but existing implementations rely on slow hardware emulators to enable dynamic symbolic execution.

In this paper, we present HyperFuzzer, the first efficient hybrid fuzzer for virtual CPUs. Our key observation is that a virtual CPU's execution is determined by the VM state. Based on this observation, we design a new fuzzing setup that uses complete VM states as fuzzing inputs, and a new fuzzing technique we call Nimble Symbolic Execution to enable dynamic symbolic execution for CPU virtualization running on bare metal. Specifically, it uses the hardware to log the control flow efficiently, and then reconstructs an approximate execution trace from only the control flow and the fuzzing input. The reconstructed execution trace is surprisingly sufficient for precise dynamic symbolic execution of virtual CPUs.

We have built a prototype of HyperFuzzer based on Intel Processor Trace for Microsoft Hyper-V. Our experimental results show that HyperFuzzer can run thousands of tests per second, which is 3 orders of magnitude faster than using a hardware emulator. When compared with a baseline using full (control+data) execution traces, HyperFuzzer can still generate 96.8% of the test inputs generated by the baseline. HyperFuzzer has found 11 previously unknown virtual CPU bugs in the Hyper-V hypervisor, and all of them were confirmed and fixed.

CCS CONCEPTS

• **Security and privacy → Software and application security**; **Systems security**.

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea.

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484748

KEYWORDS

fuzzing, symbolic execution, virtualization, hypervisor

ACM Reference Format:
Xinyang Ge, Ben Niu, Robert Brotzman, Yaohui Chen, HyungSeok Han, Patrice Godefroid, and Weidong Cui. 2021. HyperFuzzer: An Efficient Hybrid Fuzzer for Virtual CPUs. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3460120.3484748

1 INTRODUCTION

Hardware-assisted virtualization is one of the most disruptive technologies in the past two decades. The increasing growth of the cloud computing industry makes virtualization more prevalent than ever. As a result, most people today run their computation tasks on top of virtualization either explicitly or implicitly.

	Root mode	Non-root mode
Ring 0	**Virtual CPU (Both)**	Virtual I/O (Type-1)
Ring 3	Virtual I/O (Type-2)	Virtual I/O (Type-1)

Table 1: The CPU mode where virtual CPU and virtual I/O are implemented in type-1 and type-2 hypervisors on x86.

Hypervisors implement the abstraction of virtual machines (VMs) and allow them to share resources on a single physical machine. This makes the security of hypervisors paramount for the cloud—Microsoft offers up to $250,000 bounty for a single bug in Hyper-V [7]. A hypervisor has two main tasks: CPU virtualization and I/O virtualization as listed in Table 1. While I/O virtualization can be implemented in a less-privileged user-mode process [11] or offloaded to dedicated hardware [2], CPU virtualization is at the heart of a hypervisor and implemented in the most privileged CPU mode (e.g., root mode, ring 0 on x86). Therefore, a security vulnerability in CPU virtualization can lead to catastrophic consequences (e.g., allowing a malicious VM to take control of the entire physical machine).

Searching for CPU virtualization bugs is challenging because the search space consists of all possible architectural states. This huge search space is reflected in the almost 5,000 dense pages of the latest Intel Software Developer Manual [32] that describes the interface implemented by a modern CPU. For example, one bug

we found in this work requires that (1) the guest VM runs in the 16-bit protected mode, (2) attempts to execute an instruction from a memory-mapped I/O region, and (3) places a specific instruction at the guest physical address 0. It is almost impossible for random testing to hit all these conditions at once to trigger the bug.

Existing solutions for securing hypervisors are insufficient. Formal verification techniques have been applied to individual components of commercial hypervisors [37] or to simpler non-commercial hypervisors [13, 21, 31, 36], but they currently do not scale to full-fledged commercial hypervisors like those deployed in the public cloud. Manually-written [6, 14] or randomly-generated [4, 8–10, 44, 45] tests are unlikely to catch bugs involving complex trigger conditions (like the one mentioned above) due to their *blackbox* or *graybox* nature. For example, the state-of-the-art hypervisor fuzzer called Nyx [45] employs a coverage-guided graybox fuzzing setup, and only finds ring-3 I/O virtualization bugs in KVM/QEMU. Dynamic symbolic execution (for *whitebox* fuzzing) can be used to search for bugs with complex trigger conditions, but existing whitebox fuzzing systems for hypervisors have low fuzzing throughput. For instance, MultiNyx [24] incurs significant slowdown by executing the hypervisor on a hardware emulator [3] which itself runs on a binary instrumentation framework [38].

This motivates us to build an efficient hybrid fuzzer tailored for CPU virtualization in hypervisors. Hybrid fuzzing [17, 39, 48, 50, 53] is an automatic test generation technique that combines coverage-guided random input mutation with precise input generation based on path constraints derived from dynamic symbolic execution [30]. Hybrid fuzzing has shown its effectiveness in finding security vulnerabilities in user-mode programs: graybox fuzzing quickly explores easy-to-find program paths with simple random input mutations, while whitebox fuzzing extends the search frontier to harder-to-find program paths with precise path constraint solving.

We introduce HyperFuzzer, the first efficient hybrid fuzzer for virtual CPUs. In the rest of the paper, we refer to hypervisor as its virtual CPU implementation unless specified otherwise. The key observation driving the design of HyperFuzzer is that *a virtual CPU's execution is determined by the VM's state, not by the hypervisor's internal state*. This is similar to a real CPU whose execution is driven by the software state (e.g., registers and memory), not by the CPU's internal state.

Based on this observation, we design a new fuzzing setup for HyperFuzzer. First, we use a complete VM state as the fuzzing input. This allows HyperFuzzer to mutate both the instruction to be executed by the VM and the architectural environment in which the instruction is executed. Mutating the full VM state is crucial for catching CPU virtualization bugs that depend on some uncommon architectural state (e.g., 16-bit protected mode). Second, we construct a VM's state as a fuzzing input from scratch. We only include data required for a valid architectural state in a fuzzing input, which allows it to be as small as a few hundred bytes. The small size of fuzzing inputs is important for effective input mutation and fast VM restore. Third, we only fuzz the (short) virtual CPU execution of the *first* VM trap triggered by a fuzzing input. We do not need to fuzz the execution of multiple VM traps because each subsequent trap can be explored by a different fuzzing input. The short execution size of fuzzing sessions is important for precise dynamic symbolic execution.

To make HyperFuzzer hybrid, we must support both coverage-guided random mutation and precise input generation based on dynamic symbolic execution. To make HyperFuzzer efficient, we must run the hypervisor natively on a real CPU instead of on a hardware emulator. This is seemingly contradictory because hybrid fuzzing usually requires instrumentation or emulation to record the execution of the fuzzing target. The key enabling technology for HyperFuzzer is a new dynamic symbolic execution technique we call Nimble Symbolic Execution (NSE). NSE uses *hardware tracing*, such as Intel Processor Trace (PT) [32, Chap. 35], to record the complete control flow of the virtual CPU's execution in the hypervisor with low performance overhead. The recorded control flow is obviously sufficient for coverage-guided fuzzing [18, 46, 52]. Unfortunately, it does not include data values for registers or memory locations, and is thus insufficient for dynamic symbolic execution that requires both the *control* and *data* flows.

To overcome this limitation, NSE reconstructs an approximate execution trace based on the fuzzing input and the recorded control flow of the virtual CPU. In principle, this reconstruction is incomplete because the hypervisor's memory and register values during the execution are not recorded and thus unknown. However, thanks to our specific fuzzing setup and our key observation that the VM state determines the execution of a virtual CPU, NSE reconstructs execution traces with high precision sufficient for dynamic symbolic execution. This allows HyperFuzzer to generate the majority of new fuzzing inputs that would be generated by a baseline system that uses a hardware emulator to record full-fidelity execution traces but is orders-of-magnitude slower than HyperFuzzer.

We have implemented a prototype of HyperFuzzer based on Intel PT for Microsoft Hyper-V. By launching a VM to directly trigger a hypervisor's native execution on a real CPU, HyperFuzzer can run thousands of tests per second, which is 3 orders of magnitude faster than using a hardware emulator like Bochs [3]. When comparing NSE against a baseline with full-fidelity execution traces (control+data), we find that NSE can generate 96.8% of the new fuzzing inputs generated by the baseline while only relying on the recorded control flows and the fuzzing inputs. Finally, HyperFuzzer has found 11 previously unknown virtual CPU bugs in the Hyper-V hypervisor, including 6 security critical ones that allow a malicious guest VM to compromise the underlying physical machine. *All* 11 bugs were confirmed and fixed.

In particular, we make the following contributions:

- The first efficient hybrid fuzzer for virtual CPUs without using a slow hardware emulator.
- The Nimble Symbolic Execution technique that enables whitebox fuzzing for virtual CPUs with only a control-flow trace recorded by the commodity hardware.
- An effective prototype of HyperFuzzer that has found 11 previously unknown virtual CPU bugs in the Hyper-V hypervisor.

2 MOTIVATION

In this section, we use a real-world bug found by HyperFuzzer to motivate our design. We first describe the bug, then explain the root cause, and finally discuss the requirements for HyperFuzzer.

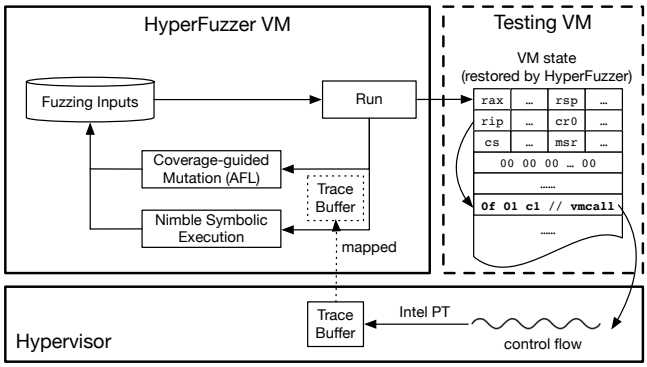


Figure 1: Overview of HyperFuzzer's design.

This particular virtual CPU bug is triggered when the guest VM is in the following state. It runs in the 16-bit protected kernel mode with paging disabled and attempts to execute an instruction from the memory-mapped I/O region of the Advanced Programmable Interrupt Controller (APIC), which is mapped at the physical address `0xFEE00000` on x86. The hypervisor traps all the guest's APIC accesses for emulation. In this case, the hypervisor detects that the guest VM runs in the 16-bit mode, so it truncates the guest's instruction pointer to the lower 16 bits. This causes the hypervisor to erroneously emulate the guest's instruction at the guest physical address 0. This inconsistency crashes the hypervisor eventually.

The root cause of this bug is a misunderstanding of the architectural difference between the 16-bit *real* mode and the 16-bit *protected* mode on x86. While the effective address for data operands is 16-bit for both modes, the instruction pointer is 16-bit in the former but 32-bit in the latter. Furthermore, this difference is not explicit in Intel's Software Developer Manual [32, Chapter 3]. The complexity of the modern CPU architecture not only makes the virtual CPU implementation error-prone, but also makes it difficult to find bugs in its huge search space.

In order to catch tricky CPU virtualization bugs like the one described above, we argue that HyperFuzzer must satisfy the following two requirements.

First, HyperFuzzer must mutate a VM's entire state rather than just the instructions it executes. When we construct the initial set of fuzzing inputs for HyperFuzzer, we do not have any input that is in the 16-bit protected mode. This mode is controlled by two bits in the Global Descriptor Table (GDT) (`CS.L = 0` and `CS.D = 0`). Without mutating the entire VM state that includes the GDT, it is impossible to trigger the bug described above.

Second, HyperFuzzer must enable precise input generation based on dynamic symbolic execution. To generate a new VM state in the 16-bit protected mode, HyperFuzzer needs to negate the two bits in the guest GDT. This requires it to precisely track the path constraints for the hypervisor's check on the guest VM mode.

3 OVERVIEW

HyperFuzzer is an efficient hybrid fuzzer for CPU virtualization in hypervisors. We focus on CPU virtualization because it is implemented in the most privileged CPU mode (e.g., root mode, ring 0 on x86), and its bugs have serious security implications for the whole system. We assume an adversary has complete control inside a guest VM, which is consistent with the threat model used by cloud operators. Specifically, we assume the adversary controls the virtual disk image and thus controls the booting code and the guest OS. This would allow the adversary to directly boot the guest VM into a bug-inducing VM state to trigger a bug in the hypervisor.

We show the end-to-end fuzzing setup of HyperFuzzer in Figure 1. HyperFuzzer begins with a set of fuzzing inputs. Each fuzzing input is a *complete* VM state including the VM's registers and entire memory. This allows HyperFuzzer to mutate both the instruction to be executed by the VM and the architectural environment in which the instruction is executed (e.g., the interrupt descriptor table, the segment attributes, the page tables). Instead of taking a snapshot of a traditional VM, we construct the seed fuzzing inputs from scratch for two reasons. First, this allows us to make a VM's state as small as a few hundred bytes by only including data required architecturally. The small size of the fuzzing inputs is important for HyperFuzzer's efficiency. Second, it allows us to construct an uncommon but architecturally valid VM state against the hypervisor. Such a VM state may not be reached by a traditional operating system.

HyperFuzzer runs its fuzzing loop in the management VM of Hyper-V (also known as the root partition [41]). For each fuzzing input, HyperFuzzer creates a dedicated testing VM and resumes its execution from the state specified in the input. The VM will trigger some virtual CPU execution. HyperFuzzer then halts the VM after its first trap into the hypervisor. During this process, HyperFuzzer leverages efficient hardware tracing, such as Intel PT [32, Chap. 35], to record the control flow of the virtual CPU's execution during the first trap.

For coverage-guided fuzzing, HyperFuzzer derives the hypervisor branch coverage from the recorded control flow, and feeds the information to a coverage-guided fuzzer such as AFL [1]. The coverage-guided fuzzer will maintain a list of interesting VM states that have triggered new code coverage, and repeatedly apply random mutations to them.

For whitebox fuzzing, HyperFuzzer marks the entire VM state as symbolic, and performs Nimble Symbolic Execution (NSE) based on the recorded control flow and the fuzzing input. NSE iterates over the recorded instruction sequence, detects the hypervisor's accesses to the symbolic VM state, tracks the path constraints for input-dependent conditional branches, and generates new inputs by solving the negated constraints to flip these branches. The key challenge for NSE is to perform these steps with only the recorded control flow (i.e., no register or memory values are recorded).

4 DESIGN

In this section, we present the design of HyperFuzzer by focusing on how it enables efficient dynamic symbolic execution for virtual CPUs in the hypervisor. We first describe the fuzzing setup of HyperFuzzer that enables hypervisor-only symbolic execution. Then we present how it performs symbolic execution based on *only* the control flow and the fuzzing input.

4.1 Fuzzing Setup

In HyperFuzzer, we use the complete VM state as the fuzzing input. In general, a guest VM runs most of its instructions natively on the

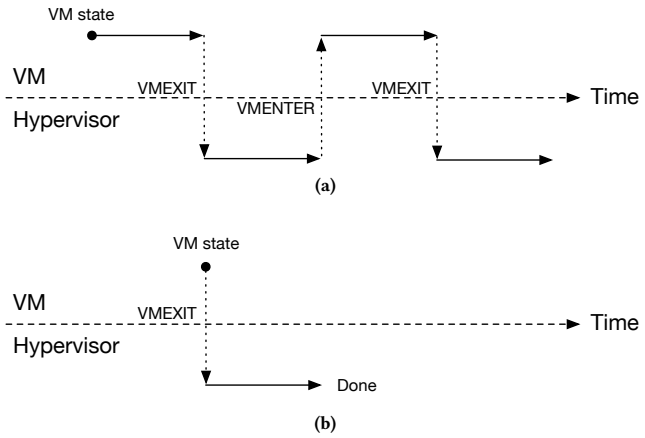


Figure 2: (a) shows a generic scenario where the VM starts execution from a specified state and constantly traps to the hypervisor for emulation. HyperFuzzer realizes hypervisor-only symbolic execution by forcing no execution inside the VM and only analyzing the virtual CPU execution of the first VM trap in the hypervisor as shown in (b).

CPU, and traps into the hypervisor only when the guest operation needs to be intercepted for emulation, such as executing a special instruction or accessing memory-mapped I/O regions. Therefore, when the testing VM is resumed from a specified VM state, its execution will alternate between the guest VM and the hypervisor, and can potentially run indefinitely as shown in Figure 2a.

This is not ideal for dynamic symbolic execution because HyperFuzzer would have to analyze not only the hypervisor's execution but also the guest VM's execution as both executions can update the symbolic VM state. MultiNyx [24] presents a multi-level analysis to enable symbolic execution across the VM and the hypervisor, but their analysis adds complexity and performance overhead for maintaining state across two CPU modes in symbolic execution.

HyperFuzzer uses a different fuzzing setup to enable a *hypervisor-only* symbolic execution. As shown in Figure 2b, the VM state is set up in such a way that the VM is immediately trapped into the hypervisor to trigger its virtual CPU execution without any execution in the VM. Furthermore, HyperFuzzer only traces the virtual CPU execution of the first trap in the hypervisor, and halts the VM after it.

This fuzzing setup has two advantages. First, by forcing no execution in the guest VM, it avoids performing symbolic execution to track state changes in the VM. Second, it allows HyperFuzzer to perform symbolic execution on a *short* virtual CPU execution history, which helps make its analysis efficient and precise.

One concern here is whether analyzing the virtual CPU execution over a single VM trap fundamentally limits HyperFuzzer's code reachability in the hypervisor. We argue that the majority of virtual CPU implementation can be tested in this fashion. Since a virtual CPU's execution is determined by the VM state, a subsequent VM trap can be potentially explored by a different VM state. For example, to test the second VM trap in Figure 2a, we can conceptually capture

```
1  ; VMEXIT entrypoint
2  push rcx
3  mov  rcx, gs:[0x10]     ; load vCPU pointer
4  mov  vcpu.rax[rcx], rax ; save guest GPRs
5  mov  vcpu.rbx[rcx], rbx ;
6  ...
7  ; VMEXIT event emulation
8  mov  rbx, vcpu.rbx[rdi] ; read guest RBX
9  cmp  rbx, 5
10 je   some_branch
```

Figure 3: An example of data propagation.

the VM state right before that trap, and use it as another fuzzing input in Figure 2b.

The remaining question is how to ensure every VM state immediately traps to the hypervisor when resumed. While we can ensure the initial fuzzing inputs satisfy the requirement via careful construction, it cannot be guaranteed after a VM state is randomly mutated. HyperFuzzer detects such cases by running the testing VM in the single-step mode to force a VM trap *after* the first instruction is executed in the VM. When such a single-step trap happens, HyperFuzzer knows the VM state did not trigger an immediate VM trap when resumed, and simply removes the fuzzing input from further symbolic execution and/or mutations.

4.2 Nimble Symbolic Execution

HyperFuzzer performs dynamic symbolic execution based on the fuzzing input and the control flow of the virtual CPU execution recorded in our fuzzing setup (§4.1). Since it is different from the traditional symbolic execution that requires a full execution trace, we call it Nimble Symbolic Execution (NSE) to emphasize its efficiency. What the control flow provides is a sequence of machine instructions *without* any register or memory values. At a high level, NSE introduces symbolic inputs and their concrete values when the hypervisor accesses the VM state, emulates every instruction in the sequence to update the concrete and symbolic stores, and solves the negated path constraints at input-dependent branches to generate new fuzzing inputs.

NSE faces two unique challenges. First, the recorded virtual CPU execution may involve the hypervisor's internal state. Since we do not have a full execution trace, the concrete values of some internal state may be unknown. Some unknown virtual CPU state may affect NSE's analysis if the unknown values are involved with symbolic inputs. Second, the physical CPU can perform checks on the VM state before trapping to the hypervisor, and NSE has to account for these hidden checks when solving constraints. Next, we describe how NSE tackles these two challenges.

4.2.1 Unknown Virtual CPU State. Some unknown virtual CPU state may affect NSE's analysis in two ways. First, if an unknown value is used to decide the memory location to store a symbolic input, NSE will lose track of the symbolic input since the memory location is unknown. Second, if an unknown value is used in the path constraint for an input-dependent conditional branch, NSE will not be able to negate the path constraint to flip the branch. The good news is that the second case is *rare*—our evaluation shows

that over 98% of the input-dependent conditional branches do not depend on any unknown virtual CPU state (§6.3.1). This matches our observation that the VM state determines the virtual CPU execution. NSE simply ignores those input-dependent branches that involve unknown virtual CPU state. In the rest of the section, we focus on how NSE solves the first case.

We show an example in Figure 3. When a guest triggers a VM trap, the hypervisor first saves all its general-purpose registers in an internal data structure representing the current virtual CPU (Line 2-6). When the hypervisor handles the trapping event for the guest, it may fetch the guest's register state from its virtual CPU data structure, and emulates the operation accordingly (Line 8-10). The main issue here is that the virtual CPU pointer is an internal state of the hypervisor, so it cannot be derived from the fuzzing input. Without knowing the pointer value, NSE will lose track of the concrete value and symbolic expression for `rbx` at Line 8. Our insight is that we do not need to know the *actual* pointer value. We can just give it some concrete value `v` so that NSE can propagate the symbolic expression and the concrete value of `rbx` through the memory location specified by `vcpu.rbx[v]`.

In general, for any missing memory address that is only used for data propagation but not involved in path constraints, NSE can assign an arbitrary value to it to keep the concrete value and the symbolic expression propagated. However, these arbitrary values for memory addresses must meet the underlying aliasing requirement (i.e., `rcx` at Line 3 and `rdi` at Line 8 are aliased addresses and must have the same value). In the context of CPU virtualization, we find that the aliasing relationship between the virtual CPU's internal data structures does *not* depend on the guest VM state. Based on this observation, we mitigate the problem of missing memory addresses in the following way. We first take a memory dump of the virtual CPU right before it starts processing a guest VM trap. We only need to take the memory dump once. The intuition is that this memory dump captures the aliasing relationship of the virtual CPU's internal data structures. When NSE emulates an instruction that accesses some virtual CPU's internal state, it takes the concrete value from the memory dump. To ensure the values from the memory dump are not used in path constraints but only used for input data propagation, NSE tracks if a concrete value or a symbolic expression is derived from the memory dump.

4.2.2 Hidden Constraints. The hardware performs a series of checks on the VM state before the execution is trapped into the hypervisor. Since these checks are done inside the hardware, they are invisible to NSE and thus cannot be captured in the path constraint generated by symbolic execution. Therefore, when NSE solves a path constraint to find a new fuzzing input, this new input might violate some (hidden) hardware check and be rejected by the hardware without triggering any virtual CPU execution in the hypervisor. Existing fuzzing solutions for user-mode code do not have this problem because all constraints are (visible) software constraints.

Ideally, NSE should emulate and derive path constraints from these implicit hardware checks described in the software developer manual. However, it requires non-trivial engineering efforts because these checks are complex and vary across CPU generations. Instead, we find that NSE can apply the following two techniques to reduce

```
1  void emulate_taskswitch()
2  {
3      // To reach here, the processor has already
4      // verified (guest_eflags & 0x0001) != 0.
5      // Now, the hypervisor is checking another bit
6      // in guest_eflags.
7      if (guest_eflags & 0x4000) {
8          ...;
9      } else {
10         ...;
11     }
12 }
```

Figure 4: An example of the challenge caused by hardware checks on the VM state.

```
1  void some_hyp_func()
2  {
3      if (a > 5) {
4          if (b == 8) { // <-- negate this one
5              ...; // this branch was taken originally
6          } else {
7              ...;
8          }
9      }
10 }
```

Figure 5: An example of unrelated constraint elimination.

unnecessary input-value changes when solving constraints and effectively mitigate the problem of hidden hardware checks.

First, NSE enables *bit-wise* precision in its symbolic execution—every bit in the fuzzing input has its own symbolic variable [47]. We show how bit-wise precision reduces input-value changes in Figure 4. Suppose the hardware has already checked bit 1 in `guest_eflags` is set (otherwise it would have rejected the input). During the hypervisor's execution, it checks bit 14 of `guest_eflags` and enters the `else` branch at line 10 as a result. To negate this branch, NSE needs to find a new value that satisfies `guest_eflags & 0x4000 != 0`. If NSE models `guest_eflags` as one symbolic variable, it is possible for the constraint solver to find a new `guest_eflags` whose bit 14 is set but bit 1 is *cleared* which violates the hardware check. With per-bit symbolic variables, the constraint would become `guest_eflags_bit14 & 0x1 != 0`, so only the bit 14 will be modified in the new input.

Second, NSE applies the unrelated constraint elimination technique introduced in SAGE [30] to remove unrelated symbolic variables from the path constraint. Take the example shown in Figure 5. If we want to flip the branch at line 4, the full path constraint would be `(a > 5)`$\wedge$`(b != 8)`. However, the input `a` is unrelated to the branch constraint `(b != 8)`. If we use the full path constraint, the constraint solver may find an arbitrary value for the input `a` as long as its new value is greater than 5. This could be a problem if the hardware has a hidden check to require it to be less than some constant value. Therefore, we eliminate branch constraints that do not share symbolic variables with the negated branch in a transitive manner to avoid unnecessary changes. In this example, the path constraint would become `(b != 8)`, which avoids changing the

Component	Lang	SLoC
Fuzzing Control (§5.1)	C/C++	6K
Hypervisor Tracing (§5.2)	C	4K
Nimble Symbolic Execution (§5.3)	C++	38K

Table 2: Implementation of HyperFuzzer (SLoC)

input a. While prior systems [30, 50] leverage this technique for performance optimization, it is *essential* to NSE in accounting for hidden hardware checks and reducing hardware rejections.

5 IMPLEMENTATION

In this section, we describe the implementation of HyperFuzzer. We implement a prototype of HyperFuzzer on the Intel VMX platform [32, Chap. 23] for Microsoft Hyper-V [41]. The prototype has three major components: fuzzing control, hypervisor tracing, and Nimble Symbolic Execution (as shown in Table 2). Next, we describe these components in detail.

5.1 Fuzzing Control

The fuzzing control component runs inside the host's management VM. It is responsible for initializing the testing VM, loading a fuzzing input to set up the testing VM, configuring the hypervisor for tracing, and passing the control flow trace to graybox and whitebox fuzzing.

Each fuzzing input is a binary file specifying a VM state that contains register and memory values. We construct these initial fuzzing inputs *manually* based on expert knowledge. This allows us to craft fuzzing inputs arbitrarily without following modern operating system conventions. We make the size of the initial fuzzing inputs as small as possible to make random mutation more effective in coverage-guided fuzzing. We do so by eliminating data structures that are unnecessary for the virtual CPU function being fuzzed. For instance, paging is not required for the 32-bit protected mode, so we can eliminate the guest page tables and make the fuzzing input as small as a few hundred bytes when constructing the state for a 32-bit VM. For virtual CPU features only reachable from a 64-bit VM, we allocate 2 page table pages to setup 512GB identity mapping using 1GB huge pages in the guest VM. This helps minimize the number of memory pages required for guest page tables.

To support hypervisor-only analysis, we set up the initial fuzzing inputs in such a way that the testing VM traps into the hypervisor upon executing the first instruction. However, this cannot be guaranteed for new fuzzing inputs after they are mutated. To handle this case, we use the Monitor Trap Flag (MTF) [32, Chap. 25.5.2] provided by Intel VMX to force the testing VM to trap into the hypervisor after executing the first instruction. When HyperFuzzer detects an MTF VM exit from the testing VM, it simply removes the fuzzing input from further analysis.

We use AFL [1] for graybox fuzzing and implement whitebox fuzzing based on NSE. We use the Z3 SMT solver [22] in whitebox fuzzing to do constraint solving. We implement HyperFuzzer as a hybrid fuzzer by integrating whitebox fuzzing into the main fuzzing loop of AFL. The pseudo code is shown in Figure 6. Specifically, we modify AFL's code to invoke whitebox fuzzing every time when

```
1  list interesting_input_queue;
2
3  void save_if_interesting(input)
4  {
5      coverage = test(input);
6      if (has_new_coverage(coverage))
7          interesting_input_queue.push_back(input);
8  }
9
10 void fuzz_one(input)
11 {
12     list new_white_inputs = WhiteboxFuzzing(input);
13     foreach (new_input in new_white_inputs)
14         save_if_interesting(new_input);
15
16     list new_gray_inputs = GrayboxFuzzing(input);
17     foreach (new_input in new_gray_inputs)
18         save_if_interesting(new_input);
19 }
20
21 void AFL(init_input_list)
22 {
23     foreach (init_input in init_input_list)
24         save_if_interesting(init_input);
25
26     while (true) {
27         foreach (input in interesting_input_queue)
28             fuzz_one(input);
29     }
30 }
```

Figure 6: Pseudo code for HyperFuzzer's hybrid fuzzing.

AFL mutates an "interesting" input that triggers new code coverage. The whitebox fuzzing runs symbolic execution using NSE on the given input and generates some new fuzzing inputs. These new fuzzing inputs are tested, and those that trigger new code coverage are added into the queue of interesting fuzzing inputs. Note that AFL initializes this queue by testing all initial fuzzing inputs.

5.2 Hypervisor Tracing

We record the control flow of the hypervisor using Intel PT [32, Chap. 35]. We modify the Hyper-V hypervisor to enable the control flow tracing for each virtual CPU. Specifically, we allocate a trace buffer for each virtual CPU, and instrument the virtual CPU switch routine to swap the trace buffers when a different virtual CPU is scheduled onto the physical processor. We do not need to trace the VM execution in HyperFuzzer, so we resume/pause Intel PT tracing when the execution enters/leaves the hypervisor. Parsing an Intel PT trace requires the hypervisor's code. We retrieve the code pages from the hypervisor memory dump (cf. Section 4.2.1).

5.3 Nimble Symbolic Execution

We implement NSE for Intel x86 ISA from scratch because existing symbolic execution engines require full execution traces and cannot handle *unknown* values. Our prototype supports common x86 instructions such as arithmetic, logical, memory access, AVX, and branch instructions. It also has a basic understanding of other

instructions (e.g., what are the source and destination operands) and implements a default policy for instructions that are not specially handled (e.g., clear the value and symbolic expression of the destination operand).

NSE initializes symbolic variables when it detects the hypervisor accessing the VM state during instruction emulation. The values of general-purpose registers in the VM *fall through* to the hypervisor on the trap. Thus NSE simply marks them as symbolic in the symbolic store and initializes the concrete store with their values specified in the fuzzing input when starting the symbolic execution. To access a memory page in the VM, the hypervisor maps the underlying physical page to its own address space. To handle such memory accesses, NSE detects the invocation of the memory mapping function during symbolic execution. Then it marks the mapped page as symbolic in the symbolic store, and populates the concrete store with the memory page's content specified in the fuzzing input. Finally, we need some special handling for *system registers* that are passed by the hardware to the hypervisor through the Virtual Machine Control Structure (VMCS) [32, Chap. 24]. To access a field in VMCS, the hypervisor uses the dedicated `VMREAD` instruction. NSE emulates this instruction to initialize the symbolic expression and the concrete value for a system register when it is accessed by the hypervisor.

NSE captures a memory dump of the hypervisor when it is rebooted to leverage its internal state in dynamic symbolic execution. First, NSE uses data stored in read-only memory pages directly because they are not updated since the memory dump is captured. Second, NSE uses data from writable pages in a conservative manner. Specifically, NSE maintains a flag to track if a concrete value or a symbolic expression contains data from writable pages in the hypervisor memory dump. This has two benefits: it allows NSE to recover more memory addresses used for input propagation, and NSE can use the flag to ignore path constraints that contain dynamic values from the hypervisor dump (see Section 4.2.1).

6 EVALUATION

In this section, we present our experimental results with HyperFuzzer. For evaluation purposes, we implement a baseline system based on the Bochs emulator [3] that can collect *full* execution traces of the hypervisor including both its control and data flows. When running our experiments, we aim to answer the following questions.

(1) Efficiency (§6.2):
   - Run time (§6.2.1): How long does it take to run a single test?
   - Throughput (§6.2.2): What is HyperFuzzer's fuzzing throughput?
(2) Precision (§6.3):
   - Completeness (§6.3.1): What fraction of input-dependent conditional branches can NSE identify?
   - Divergences (§6.3.2): What fraction of inputs generated by NSE can correctly flip their targeted branches?
(3) Coverage (§6.4): How is HyperFuzzer's code coverage compared with graybox-only or whitebox-only fuzzing?
(4) Bugs (§6.5): Can HyperFuzzer find previously unknown virtual CPU bugs in the Hyper-V hypervisor?

	Initial	Expanded
Hypercalls	157	1091
Task Switch	5	186
APIC Emu.	56	521
MSR Emu.	2	476

Table 3: The number of fuzzing inputs in the initial and expanded fuzzing sets for different virtualization interfaces.

In the rest of this section, we first present our experimental methodology, and then describe our experimental results in detail.

6.1 Experimental Methodology

6.1.1 Experiment Setup. We run all experiments on a workstation with a quad-core Intel i7-6700K processor and 16GB RAM. We focus our experiments on four virtualization interfaces: hypercalls, hardware task switch [32, Chap. 7] emulation, advanced programmable interrupt controller (APIC) emulation, and model-specific register (MSR) emulation. We pick these interfaces because they either have a big attack surface (e.g., hypercalls) or previously-reported bugs (e.g., Task Switch). We do not cover all virtualization interfaces due to the manual efforts required for understanding and constructing initial fuzzing inputs for these.

We manually create a number of initial fuzzing inputs [12] for Task Switch, APIC and MSR based on Intel Software Developer Manual [32]. For instance, we have one fuzzing input for reading and the other for writing an MSR. For hypercalls, we generate a single initial fuzzing input for each hypercall API. Specifically, we leverage an existing tool [5] to randomly generate parameters based on the parameter's type.

Our evaluation requires a large number of fuzzing inputs to conduct the experiments. We run HyperFuzzer on the initial seed fuzzing inputs for 120 minutes to generate a new set of fuzzing inputs. In this new set, some inputs are generated by whitebox symbolic executions, and others are generated by graybox random mutations. We then keep all the new inputs that triggered new code coverage, and discard others. We refer to the remaining set as the expanded fuzzing set (see Table 3). Note that we divide fuzzing inputs in the expanded fuzzing set based on the virtualization interface it exercises for the purpose of evaluation. In practice, we do not need to differentiate what virtualization interface a fuzzing input exercises.

As described in §5.1, we implement HyperFuzzer by integrating the symbolic execution-based input generation into the main fuzzing loop of AFL [1] that performs coverage-guided random mutation. In the rest of this section, hybrid fuzzing refers to this implementation, graybox fuzzing includes only AFL's coverage-guided random mutation, and whitebox fuzzing includes only the symbolic-execution-based input generation.

6.1.2 Bochs-Based Baseline. We implement a baseline system based on the Bochs emulator [3] for our experiments. We use this baseline system to demonstrate HyperFuzzer's performance improvement (§6.2), and to provide the ground truth for evaluating NSE's effectiveness when only given a control-flow trace (§6.3). As shown

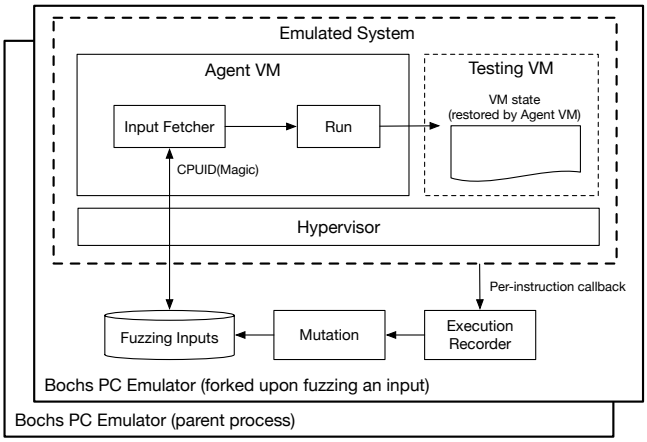


Figure 7: The architecture of the Bochs-based baseline.

in Figure 7, we run the Hyper-V hypervisor inside Bochs and instrument Bochs to enable dynamic symbolic execution for the hypervisor. We choose Bochs because it emulates Intel's modern hardware virtualization extension (VMX) with high fidelity while other hardware emulators (e.g., QEMU) either do not emulate VMX at all or in a very limited fashion. We implement per-instruction callbacks in Bochs to record both the control and data flow of the hypervisor's execution.

The fuzzing loop of the baseline system is similar to HyperFuzzer except that it runs inside an emulator. We run an agent inside the management VM to load a fuzzing input and trigger the hypervisor's execution. The agent requests a fuzzing input by executing `CPUID` with a special leaf value, and Bochs then copies the fuzzing input to the agent's memory space. The agent triggers the hypervisor's execution in a way similar to HyperFuzzer by launching a testing VM and setting up its state based on the fuzzing input.

We optimize the performance of the baseline system by *forking* the Bochs emulator process when it fuzzes a new input. This enables it to quickly rollback the entire emulated system including the hypervisor and the agent. This optimization improves the fuzzing throughput of the baseline system by two orders of magnitude compared to a naive approach that restarts the Bochs emulator from an on-disk snapshot every single time.

6.2 Efficiency

We evaluate HyperFuzzer's efficiency by comparing it with the Bochs-based baseline system in two experiments. In the first experiment, we measure the average run time for performing a single test (without any mutation) or a single symbolic execution (with constraint solving) in HyperFuzzer and the baseline system. For this experiment, we integrate the Triton symbolic execution engine [43] into the Bochs emulator. In the second experiment, we measure the throughput of hybrid, graybox and whitebox fuzzing.

6.2.1 Run Time. We measure the average run time of a single test or symbolic execution for HyperFuzzer and the baseline system on the expanded fuzzing set and show the result in Table 4. The average run time for a single test is less than 0.5ms for HyperFuzzer, while the baseline system is 3 orders of magnitude slower. The average run time for a single symbolic execution is between 212ms and 782ms, while the baseline system is 4 to 10 times slower. The ratio difference changes because both systems spend a significant amount of time on constraint solving. However, HyperFuzzer is still more efficient than the baseline system. We attribute this to the implementation differences between the two systems.

	HyperFuzzer		Bochs-Based Baseline	
	Testing	NSE	Testing	Triton [43]
Hypercalls	0.48	781.89	683.73	4861.84
Task Switch	0.33	457.83	690.94	3499.39
APIC Emu.	0.36	212.66	696.31	2074.09
MSR Emu.	0.36	387.51	687.87	1871.94

Table 4: The efficiency comparison between HyperFuzzer and Bochs-based baseline system in performing a single test and symbolic execution. Numbers in this table are in milliseconds.

	Hybrid	Graybox	Whitebox
Hypercalls	979.40	2945.03	35.63
Task Switch	1369.95	4378.22	106.88
APIC Emu.	1774.05	3650.68	210.85
MSR Emu.	1171.24	3967.35	76.67

Table 5: The throughput (# tests/sec) of hybrid, graybox and whitebox fuzzing.

	Input-Dep Branches	New Inputs
Hypercalls	98.1%	96.8%
Task Switch	98.8%	98.9%
APIC Emu.	98.2%	99.2%
MSR Emu.	98.3%	99.2%

Table 6: The completeness evaluation of NSE. The percentages listed are the fraction of input-dependent conditional branches and new fuzzing inputs identified/generated by NSE (based on the control flow and fuzzing input) compared to the baseline set (based on full execution traces).

6.2.2 Throughput. We evaluate HyperFuzzer's throughput by running hybrid, graybox and whitebox fuzzing on the initial fuzzing inputs for 120 minutes and reporting the average number of tests per second in Table 5. We can see that hybrid fuzzing can run 1000's of tests per second, and graybox fuzzing's throughput is 2 to 3 times higher than that. The reported throughput for whitebox fuzzing is higher than the number calculated based on the average run time of symbolic execution. This is because not every tested fuzzing input is passed to the symbolic execution in the allotted time (120 minutes). HyperFuzzer's fuzzing throughput is 3 orders of magnitude higher than the baseline system because only a small fraction of fuzzing inputs that trigger new code coverage are tested with NSE.

6.3 Precision

We evaluate NSE's precision by measuring what fraction of input-dependent conditional branches it can correctly *identify* and then *flip*. The former defines *completeness*, while the latter is measured by counting *divergences*. A divergence occurs whenever a new input generated to exercise a specific program path actually takes an unintended path.

6.3.1 Completeness. We use the expanded fuzzing set to measure completeness. We pick this data set to reduce the bias of using NSE generated inputs to evaluate its effectiveness because this data set contains fuzzing inputs generated by graybox fuzzing as well. For each fuzzing input, we record the hypervisor's full execution trace (control+data) by using the Bochs-based baseline system. We then run our core symbolic execution engine over the full execution trace in order to compute a baseline set of input-dependent conditional branches and new fuzzing inputs. To run NSE, we extract the control flow from full execution traces as if it were logged by Intel PT. Then to measure completeness, we compare the baseline set of the input-dependent conditional branches and the new fuzzing inputs with those computed by NSE based on the extracted control-flow-only trace.

In other words, in order to isolate the effects of full (control+data) versus control-flow-only execution traces, we run our core symbolic execution engine on both Bochs-based traces and Intel PT traces to measure completeness. In contrast, we deliberately do not use Triton on the Bochs-based traces since such a comparison would then obfuscate the results with other factors such as the quality of two different symbolic execution engines, constraint generators and solvers.

We show the experimental results in Table 6. We can see that NSE can identify at least 98.1% of input-dependent conditional branches and generate at least 96.8% of new fuzzing inputs in the baseline set that are obtained from full execution traces. Note that we count each occurrence of an input-dependent branch instruction in the trace separately. The common reason for NSE to miss an input-dependent conditional branch is that the hypervisor mixes the input with its internal state, which is unknown to NSE. When this happens, NSE sets the new value to unknown and stops tracking its symbolic expression. This under-approximation of the symbolic state is also the reason why NSE never misidentifies an input-dependent branch that is not in the baseline set (no false positives, by construction). Furthermore, the main reason for NSE to miss a new fuzzing input is that NSE fails to identify its corresponding branch as an input-dependent branch.

6.3.2 Divergences. In this section, we describe our experimental results on divergences by measuring what fraction of inputs generated by NSE can actually flip its targeted branch. In this experiment, we run whitebox fuzzing on the expanded fuzzing set. For every newly generated fuzzing input, we test it and save its control flow. Then we compare it with the control flow of its "parent" input (the one we used to generate the new fuzzing input) to check if it flips the targeted branch successfully.

We show the experimental results in Figure 8. We run the experiment in three setups to evaluate the effectiveness of mitigations

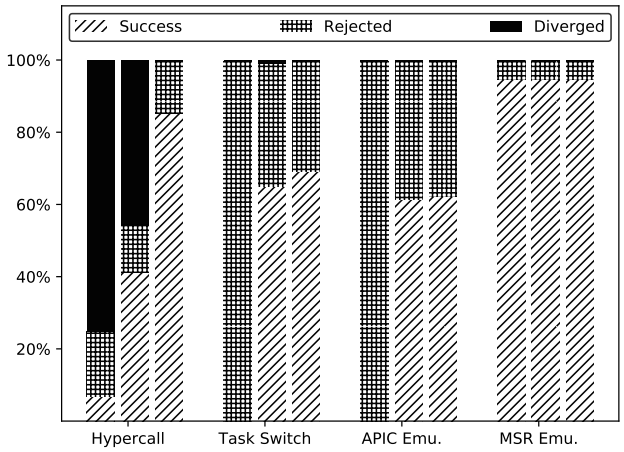


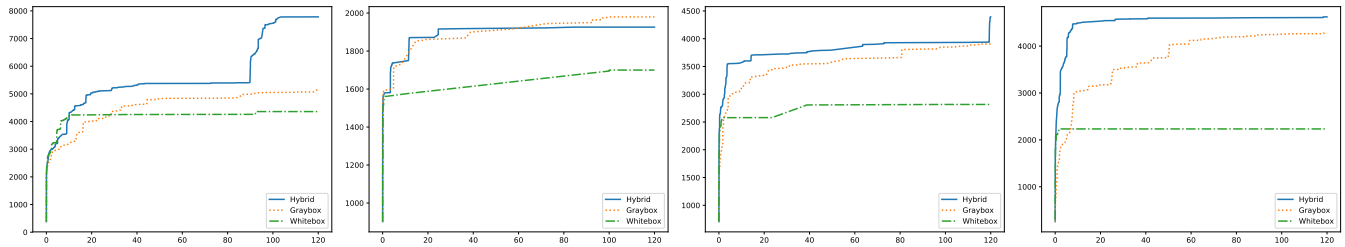
Figure 8: HyperFuzzer's divergence rate on newly generated fuzzing inputs. Success means the newly generated fuzzing input successfully flips its targeted branch. Rejected means it is rejected by hardware checks. Diverged means it fails to flip its targeted branch. For each virtualization interface, we progressively show the results from left to right: (1) without unrelated constraint elimination and bit-wise symbolic variables, (2) with unrelated constraint elimination but without bit-wise symbolic variables, and (3) with both.

described in §4.2.2 for hidden hardware checks: (1) without unrelated constraint elimination and bit-wise symbolic variables, (2) with unrelated constraint elimination but without bit-wise symbolic variables, and (3) with both. We present their results from left to right for each virtualization interface, respectively. For each setup, we measure the percentages of newly generated fuzzing inputs that flip the input-dependent branches as intended (Success), get rejected due to violating hardware checks (Rejected), or fail to flip the intended branches (Diverged).

The results show that the two techniques for minimizing changes to the original fuzzing input are effective in reducing hardware rejections. For task switch, 100% of newly generated fuzzing inputs are rejected by the hardware when neither mitigation is enabled. This is because Intel CPUs perform 11 different checks before passing the control to the hypervisor for emulation [32, Chap. 25.4.2].

Another interesting observation here is that the two techniques help reduce both hardware rejections and software divergences. In our experiment, unrelated constraint elimination significantly reduces software divergences when fuzzing hypercalls. The root cause of these divergences is related to an implementation detail of NSE. Specifically, NSE always uses a concrete memory address when emulating a memory operation even if its address is symbolic (i.e., NSE does not support symbolic pointers [16, 23]). This can cause inconsistencies when the constraint solver assigns a new value to a symbolic variable previously used as a memory address.

In the particular case of hypercalls, the hypercall number is first range-checked by the hypervisor and then used as an index to fetch the corresponding hypercall handler from a function pointer table. When flipping a conditional branch inside the hypercall handler,



(a) Hypercalls (b) Task Switch (c) APIC Emu. (d) MSR Emu.

Figure 9: Edge coverage of hybrid, graybox and whitebox fuzzing. X axis spans the time range from 0 to 120 minutes. Y axis represents the aggregated number of unique control-flow edges covered by each fuzzing configuration.

NSE can assign a different hypercall number as long as it satisfies the aforementioned range check *if the range check is included in the path constraint*. This eventually leads to a divergence as the subsequent test can no longer reach the same hypercall handler, not to mention the conditional branch it intends to flip. Both eliminating unrelated constraints and using bit-wise symbolic variables can help exclude the range check from the path constraint. It is obvious that eliminating unrelated constraints can do it when the range check is indeed unrelated to the branch we try to flip. Using bit-wise symbolic variables also helps because the hypercall number is specified as a bit field and mixed with other bit-level flags. When symbolic variables are bit-wise, we avoid including the hypercall number in the path constraint for a branch that checks the other bit-level flags.

Divergences remain after we apply both techniques. The main reason for these divergences is that NSE misses some input-dependent conditional branches. When such a branch is missed, its branch constraint is also missed in the path constraint when NSE tries to flip a subsequent branch. Then the generated new input may violate the missing branch's constraint and thus lead to a divergence.

6.4 Coverage

In this section, we report our experiments on coverage. The goal is to check if, and by how much, HyperFuzzer achieves a better coverage than graybox-only or whitebox-only fuzzing. In our experiments, we run the graybox, whitebox and hybrid fuzzing separately on the initial fuzzing input set for 120 minutes. Then we count the aggregated number of unique control-flow edges covered by each fuzzing configuration. To measure the coverage for each virtualization interface, we exclude the inputs and their coverage if they do not trigger the targeted interface. Note that the expanded fuzzing set described in Table 3 is constructed based on "interesting" fuzzing inputs (those that trigger new code coverage) generated from this experiment.

Our experimental results on edge coverage are shown in Figure 9. We can see that hybrid fuzzing outperforms graybox-only and whitebox-only fuzzing. The coverage difference between graybox and hybrid fuzzing is small for Task Switch because the code for Task Switch is relatively simpler than the other three virtualization interfaces. The coverage jumps in hybrid fuzzing for hypercalls and APIC emulation are likely because random mutation just triggered some new code paths.

	MemSafe	IntOvf	Logical	MemLeak
Hypercalls	3	1	0	1
Task Switch	0	0	1	0
APIC Emu.	0	0	3	0
MSR Emu.	0	0	2	0

Table 7: Summary of real-world hypervisor bugs found by HyperFuzzer.

Whitebox-only fuzzing has the worst coverage in our experiments. We analyze the code paths covered by graybox fuzzing but not by whitebox fuzzing and find that there are two main reasons for whitebox fuzzing to miss them. First, NSE does not support symbolic pointers. For example, some guest VM state can be used as an index to a function pointer table. Without symbolic pointers, NSE will not be able to generate a new fuzzing input with a different index to the function pointer table. However, it is possible for random mutation to modify the index and lead to a different function. Second, the hardware checks are invisible to whitebox fuzzing. When whitebox fuzzing tries to flip a branch, the generated new input may violate a hidden hardware check. When this happens, whitebox fuzzing will result in a hardware rejection and make the targeted path unreachable. We evaluate the prevalence of hardware rejections in §6.3.2. On the other hand, it is possible for random mutation to generate a new input that happens to flip the branch. Therefore, we believe hybrid fuzzing is the right approach for hypervisors.

6.5 Bug Analysis

HyperFuzzer has found 11 previously unknown virtual CPU bugs in the Hyper-V hypervisor as summarized in Table 7. Out of the 11 bugs, 6 are *security critical* because a malicious guest VM can exploit them to compromise the hypervisor. The other 5 bugs are less critical because they only affect the testing VM itself. HyperFuzzer detects the 11 bugs based on signals like assertion violations (e.g., logical bugs) and crashes (e.g., memory safety). Most of the bugs reported here were found within hours of fuzzing.

We manually analyze the 11 bugs found by HyperFuzzer and find that all the 6 logical bugs require symbolic-execution-based input generation.

- Task Switch (1 bug): This bug requires a special task segment to be triggered.
- APIC Emulation (3 bugs): Two bugs require a specific guest mode (e.g., 16-bit protected mode) to be triggered. The third bug requires a special opcode (RDTSC) to be triggered.
- MSR Emulation (2 bugs): Both bugs require a special MSR index and value to be triggered.

The final VM states triggering these bugs were found thanks to NSE combined with coverage-guided random mutation. The latter helps explore trivial branches, while the former is effective for unlocking those hard-to-reach branches. For instance, NSE is able to go through the switch cases in the hypervisor's instruction emulation code to generate the RDTSC opcode required for one of the APIC emulation related bugs.

7 LIMITATIONS

HyperFuzzer is currently limited to hypervisors running on the Intel VMX platform due to its dependency on Intel PT. This makes it unable to reach a hypervisor's platform-dependent code. For example, certain MSRs are only available on AMD processors, and bugs in their emulation code will not be captured by HyperFuzzer.

HyperFuzzer does not support multiple virtual CPUs for whitebox fuzzing, which limits its ability to find race condition bugs in the hypervisor. Such support requires new techniques for both control-flow tracing and dynamic symbolic execution. We leave it for future work.

We have demonstrated NSE's high effectiveness in performing dynamic symbolic execution over a control-flow trace (§6.3) of Hyper-V hypervisor's virtual CPU. However, it may not be universally applicable to all targets. We believe that NSE works well for targets whose execution is driven by the inputs, such as an image parser or the virtual CPUs focused by this work. But NSE may not work well for programs whose input can be tangled with the internal state, which is unknown when only a control-flow trace is captured. For example, the behavior of a stateful web server depends on both the incoming input and its current state. A potential research direction is to systematically identify the critical internal data to log for dynamic symbolic execution.

8 RELATED WORK

HyperFuzzer is the *first* efficient hybrid fuzzer for virtual CPUs. Its main difference from previous work is that it does not rely on instrumentation or emulation to record the full program execution for symbolic execution. Instead, it only records the program's control flow by using commodity hardware tracing and is able to perform precise dynamic symbolic execution on top of the control flow trace. In this section, we discuss previous work on hybrid fuzzing, hypervisor testing, and hardware tracing.

8.1 Hybrid Fuzzing

Fuzzing means automatic test generation and execution with the goal of finding security vulnerabilities [28, 49].

Blackbox random fuzzing is the simplest form of fuzzing: it either mutates well-formed application inputs or directly generates inputs, and then tests the application with these new inputs [25]. Blackbox random fuzzing provides a simple fuzzing baseline. It is effective for some simple targets [42], but its effectiveness is limited: the probability of generating new interesting inputs is low [49].

Whitebox fuzzing [30] combines fuzzing with dynamic test generation [29], which consists of *symbolically* executing the program under test *dynamically*, gathering constraints on inputs from conditional branches encountered along the execution, and then negating and solving these constraints with a *constraint solver*. Solutions of satisfiable constraints are mapped to new inputs that exercise different program execution paths. Whitebox fuzzing can typically generate inputs that exercise more code than other approaches because it is *more precise* [27], which explains why it has been implemented in several popular open-source tools [15, 19]. However, it is more complex to engineer and its throughput is lower than blackbox fuzzing.

Graybox fuzzing extends blackbox fuzzing with whitebox fuzzing techniques. It approximates whitebox fuzzing by eliminating some of its components to reduce engineering cost and complexity while retaining some of its intelligence. AFL [1] is a popular open-source fuzzer which extends random fuzzing with code-coverage-based search heuristics, but without any symbolic execution, constraint generation or solving. Despite of its simplicity, AFL was shown to find many bugs missed by pure blackbox random fuzzing.

Hybrid fuzzing [17, 35, 39, 48, 50, 53] combines graybox fuzzing techniques with whitebox fuzzing. The goal is to explore trade-offs to determine when and where simpler techniques are sufficient to obtain good code coverage, and use more complex techniques, like symbolic execution and constraint solving, only when the simpler techniques are stuck. HFL [35] brings hybrid fuzzing to the kernel space by performing dynamic symbolic execution based on hardware emulation [19] and handling kernel-specific challenges such as inferring system call dependencies.

As the first efficient hybrid fuzzer for virtual CPUs, HyperFuzzer has two main differences when compared with HFL. First, they have different fuzzing targets. To fuzz hypervisors, HyperFuzzer performs hypervisor-only analysis by focusing on the hypervisor execution over a single VM trap. Second, HyperFuzzer leverages hardware tracing to achieve precise and efficient symbolic execution by using NSE while HFL is based on hardware emulation.

8.2 Hypervisor Testing

Amit *et al.* [14] adapts Intel's tools for testing a physical CPU to virtual CPUs implemented by a hypervisor. The CPU testing tool generates a sequence of random instructions to execute on the virtual CPU and checks for architectural state divergences in comparison to a reference implementation such as a CPU simulator. The tool does not take any feedback from the hypervisor execution (i.e., blackbox testing), but relies on its intimate awareness of x86 architecture to generate comprehensive test cases.

MultiNyx [24] systematically generates test cases for a hypervisor by applying dynamic symbolic execution to the whole system. To do so, MultiNyx runs the hypervisor on an instrumented Bochs

emulator [3], which itself runs on top of the Pin binary instrumentation framework [38]. MultiNyx combines traces across multiple levels to realize the dynamic symbolic execution for test case generation. This leads to high performance overhead because of the emulation cost and the complexity in reasoning about multi-level traces for symbolic execution.

PokeEMU [40] performs symbolic execution on a high-fidelity emulator (e.g., Bochs) to generate test cases for other virtual CPU implementations (e.g., hypervisors). PokeEMU does not reason about the execution of the system under test. Therefore, a corner case that is only present in the hypervisor may not be uncovered based on the analysis of a different virtual CPU implementation.

Hyper-Cube [44] implements a blackbox hypervisor fuzzer based on a custom operating system running a custom bytecode interpreter. Hyper-Cube's blackbox nature makes it less likely to hit hypervisor bugs involving complex conditions. Furthermore, Hyper-Cube does not mutate the VM's architectural state in which the bytecode gets interpreted. This can limit its testing coverage as the hypervisor depends on the VM's architectural state when emulating an operation.

Nyx [45] is a coverage-guided graybox hypervisor fuzzer. It runs a target hypervisor in a guest VM via nested virtualization, and records its coverage using Intel PT. It relies on Hyper-Cube to drive the workload, so it shares the same limitation that the VM's architectural environment is not mutated. Furthermore, its lack of whitebox fuzzing for precise input generation limits its search space. In fact, Nyx has been reported to only find bugs in ring-3 I/O device emulation code in QEMU (not in KVM's virtual CPU code running in ring-0).

As discussed in §2, in order to catch tricky virtual CPU bugs, HyperFuzzer must be able to mutate a VM's entire state (e.g., modify the GDT) and generate precise inputs based on dynamic symbolic execution (e.g., generate the RDTSC opcode). Hypervisor fuzzers like Hyper-Cube and Nyx do not meet either requirement. Therefore, they would miss the 6 bugs described in §6.5 that require symbolic execution and/or VM state mutation.

8.3 Hardware Tracing

Intel PT [32, Chap. 35] is today's most practical hardware tracing technology. It can record complete control flow with low performance overhead and without modifying the tracing target. It can also record fine-grained timestamps. Intel PT has been used in the following scenarios.

Fuzzing. Several systems [18, 46, 52] use Intel PT to enable coverage-guided graybox fuzzing. kAFL [46] implements a coverage-guided fuzzer for arbitrary OS kernels running inside a VM by enabling hardware tracing, such as Intel PT, from the hypervisor. kAFL has only been applied to OS kernel components and is limited to coverage-guided fuzzing. PTRIX [18] combines AFL with Intel PT to fuzz commercial-off-the-shelf (COTS) program binaries in an efficient manner. It achieves high efficiency by mapping highly-compressed Intel PT traces to code coverage without reconstructing the exact control flow. PTFuzz [52] enables graybox binary-only fuzzing by taking the control flow recorded by Intel PT as the feedback. It overcomes the inaccurate coverage representation in AFL by using the actual transitions between basic blocks logged in the control-flow trace.

Pointer Analysis. SNORLAX [34] uses the control flow recorded by Intel PT to perform points-to analysis and use its timestamps to determine thread interleaving. Compared with traditional static pointer analysis, this hardware-assisted approach limits the analysis to the recorded execution path and achieves higher accuracy.

Reverse Debugging. REPT [20, 26] is a reverse debugging tool based on Intel PT. It can infer data values based on the control flow recorded by Intel PT and the final program state captured in a memory dump. Despite its ability in recovering an approximate execution history, REPT cannot be directly applied to hybrid fuzzing because taking a memory dump for each run is too expensive.

Failure Reproduction. This is an important and hard problem in software engineering. Existing failure reproduction techniques [33, 51] face the challenge of path explosion and high overhead due to the extra logging. Execution Reconstruction (ER) [54] is a new production failure reproduction technique which harnesses failure reoccurrences to iteratively perform hardware-assisted control/data tracing and symbolic execution which identifies what key data should be logged for a successful failure reproduction.

A common theme shared by previous work on pointer analysis, reverse debugging and failure reproduction is that the recorded execution path can achieve better accuracy than traditional *static* analysis that has to examine all possible execution paths. In contrast, HyperFuzzer leverages Intel PT for whitebox fuzzing: starting from an incomplete execution history (i.e., only the control flow), HyperFuzzer is able to reconstruct enough of the execution to perform dynamic symbolic execution, path constraint generation and solving that is nearly as precise as traditional approaches that rely on a complete execution history.

9 CONCLUSION

We have presented HyperFuzzer, the first efficient hybrid fuzzer for virtual CPUs. HyperFuzzer achieves both efficiency and precision by leveraging hardware tracing to record the control flow of the hypervisor efficiently, and by introducing a new fuzzing technique called Nimble Symbolic Execution to perform precise symbolic execution by using only the recorded control flow and the fuzzing input. We implemented a prototype of HyperFuzzer for Microsoft Hyper-V hypervisor. Our experiments show that HyperFuzzer achieves high fuzzing throughput, and can identify and flip most input-dependent branches. More importantly, HyperFuzzer has found 11 previously unknown virtual CPU bugs in the Hyper-V hypervisor, and all of them were confirmed and fixed.

ACKNOWLEDGMENTS

We thank our shepherd, Yajin Zhou, and other reviewers for their insightful feedback. We are very grateful for all the help from our colleagues at Microsoft. In particular, Aditya Bhandari, Alexander Grest, David Hepkin, Daniel King, Eric Lee, Sunil Muthuswamy, Sai Ganesh Ramachandran, Bruce Sherwin, David Zhang provided tremendous help and valuable perspectives for integrating HyperFuzzer with the Windows Hyper-V hypervisor and resolving the bugs found by HyperFuzzer. We also thank Hangchen Yu for his internship work on enabling Intel PT tracing of the hypervisor.

REFERENCES

[1] American Fuzzy Lop. https://github.com/google/AFL.

[2] AWS Nitro System. https://aws.amazon.com/ec2/nitro/.

[3] Bochs x86 PC emulator. http://bochs.sourceforge.net/.

[4] Fuzzing Para-virtualized Devices in Hyper-V. https://msrc-blog.microsoft.com/2019/01/28/fuzzing-para-virtualized-devices-in-hyper-v/.

[5] Hyperseed. https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_OffensiveCon/2019_02%20-%20OffensiveCon%20-%20Growing%20Hypervisor%200day%20with%20Hyperseed.pdf.

[6] KVM Unit Tests. https://www.linux-kvm.org/page/KVM-unit-tests.

[7] Microsoft Hyper-V Bounty Program. https://www.microsoft.com/en-us/msrc/bounty-hyper-v.

[8] Ventures into Hyper-V - Fuzzing hypercalls. https://labs.f-secure.com/blog/ventures-into-hyper-v-part-1-fuzzing-hypercalls.

[9] Viridian Fuzzer. https://github.com/FSecureLABS/ViridianFuzzer.

[10] XenFuzz. https://www.openfoo.org/blog/xen-fuzz.html.

[11] Attacking the VM Worker Process. https://msrc-blog.microsoft.com/2019/09/11/attacking-the-vm-worker-process/.

[12] https://github.com/MSRSSP/hyperfuzzer-seeds.

[13] Eyad Alkassar, Mark A Hillebrand, Wolfgang Paul, and Elena Petrova. 2010. Automated Verification of a Small Hypervisor. In *Proceedings of the Third International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*.

[14] Nadav Amit, Dan Tsafrir, Assaf Schuster, Ahmad Ayoub, and Eran Shlomo. 2015. Virtual CPU Validation. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*.

[15] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*.

[16] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*.

[17] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. SAVIOR: Towards Bug-Driven Hybrid Testing. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*.

[18] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. 2019. Ptrix: Efficient Hardware-Assisted Fuzzing for COTS Binary. In *Proceedings of the 14th ACM Asia Conference on Computer and Communications Security*.

[19] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[20] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. REPT: Reverse Debugging of Failures in Deployed Software. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[21] Mike Dahlin, Ryan Johnson, Robert Bellarmine Krug, Michael McCoyd, and William Young. 2011. Toward the Verification of a Simple Hypervisor. In *Proceedings of the 10th International Workshop on the ACL2 Theorem Prover and its Applications*.

[22] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 337–340.

[23] Bassem Elkarablieh, Patrice Godefroid, and Michael Y. Levin. 2009. Precise Pointer Reasoning for Dynamic Test Generation. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA)*.

[24] Pedro Fonseca, Xi Wang, and Arvind Krishnamurthy. 2018. MultiNyx: a Multi-Level Abstraction Framework for Systematic Analysis of Hypervisors. In *Proceedings of the Thirteenth EuroSys Conference*.

[25] J. E. Forrester and B. P. Miller. 2000. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proceedings of the 4th USENIX Windows System Symposium*. Seattle.

[26] Xinyang Ge, Ben Niu, and Weidong Cui. 2020. Reverse Debugging of Kernel Failures in Deployed Systems. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC'20)*. 281–292.

[27] Patrice Godefroid. 2011. Higher-Order Test Generation. In *Proceedings of ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation (PLDI)*.

[28] Patrice Godefroid. 2020. Fuzzing: Hack, Art, and Science. *Communications of the ACM* 63, 2 (February 2020), 70–76.

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

[30] Patrice Godefroid, Michael Y Levin, and David Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*.

[31] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[32] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.

[33] Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing Field Failures for In-House Debugging. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*.

[34] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. 2017. Lazy Diagnosis of In-Production Concurrency Bugs. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*.

[35] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel. In *Proceedings of the 28th Network and Distributed Systems Security Conference (NDSS)*.

[36] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*.

[37] Dirk Leinenbach and Thomas Santen. 2009. Verifying the Microsoft Hyper-V hypervisor with VCC. In *Proceedings of the International Symposium on Formal Methods*.

[38] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *ACM SIGPLAN Notices* 40, 6 (2005), 190–200.

[39] Rupak Majumdar and Koushik Sen. 2007. Hybrid Concolic Testing. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*. IEEE, 416–426.

[40] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 2012. Path-Exploration Lifting: Hi-Fi Tests for Lo-Fi Emulators. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[41] Microsoft. Hyper-V Architecture. https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/hyper-v-architecture.

[42] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (Dec 1990).

[43] Florent Saudel and Jonathan Salwan. 2015. Triton: A Dynamic Symbolic Execution Framework. In *Symposium sur la sécurité des technologies de l'information et des communications*. 31–54.

[44] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2020. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing. In *Proceedings of the 28th Network and Distributed Systems Security Conference (NDSS)*.

[45] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2021. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *Proceedings of the 30th USENIX Security Symposium*. Virtual Event.

[46] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *Proceedings of the 26th USENIX Security Symposium*. 167–182.

[47] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*.

[48] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 24th Network and Distributed System Security Symposium (NDSS)*.

[49] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley.

[50] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium*.

[51] Cristian Zamfir and George Candea. 2010. Execution Synthesis: A Technique for Automated Debugging. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*.

[52] Gen Zhang, Xu Zhou, Yingqi Luo, Xugang Wu, and Erxue Min. 2018. PTFuzz: Guided Fuzzing with Processor Trace Feedback. *IEEE Access* 6 (2018), 37302–37313.

[53] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In *Proceedings of the 27th Network and Distributed System Security Symposium (NDSS)*.

[54] Gefei Zuo, Jiacheng Ma, Andrew Quinn, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. 2021. Execution Reconstruction: Harnessing Failure Reoccurrences for Failure Reproduction. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'21)*. Virtual Event.