HyperMirage: Direct State Manipulation in Hybrid Virtual CPU Fuzzing

Manuel Andreas
Technical University of Munich
manuel.andreas@tum.de

Fabian Specht
Technical University of Munich
f.specht@tum.de

Marius Momeu
Technical University of Munich
marius.momeu@tum.de

***Abstract*—Hypervisors are crucial for the security and availability of modern cloud infrastructures, yet they must expose a large virtualization interface to guest VMs—an attack surface that adversaries can exploit. Among the most intricate and security-sensitive components of hypervisors is their virtual CPU implementation, typically implemented at the highest privilege level. Although previous fuzzing research made promising steps towards scrutinizing the virtual CPU component of HVs, existing techniques fail at covering it in depth, as its convoluted nature requires laborious manual setup for accessing individual interfaces, all the while employing sub-optimal techniques that lower fuzzing throughput.**

We address these shortcomings via HyperMirage, a novel hypervisor fuzzer that automatically and efficiently explores the large space of architectural states emulated by virtual CPU implementations. HyperMirage spares security analysts from manually crafting fuzzing seeds in the form of architecturally valid VM states by employing a novel Direct State Manipulation approach, which directly and automatically mutates the HV's view of a VM's state that is consumed during fuzzing. Additionally, we extend a state-of-the-art compiler-based symbolic execution engine, making it the first one available for bare-metal targets, and integrate it into an efficient coverage-guided HV fuzzer, enabling HyperMirage to drastically improve fuzzing throughput when compared to existing techniques.

We provide a case study of HyperMirage by fuzzing the production-grade Xen and KVM hypervisors on the Intel x86 architecture. Our evaluation shows that HyperMirage is capable of covering 200% more virtual CPU interfaces than prior work and achieves drastically more coverage on the entire virtual CPU space when compared to available HV fuzzers. Moreover, HyperMirage discovered 9 new bugs in Xen and 2 in KVM, all of which have been confirmed by the respective project maintainers.

I. Introduction

The adoption of hardware virtualization extensions—such as Intel VMX [1], AMD SVM [2], and ARM Virtualization Extension [3]—enabled several opportunities for the cloud computing landscape. However, it also introduced a whole new surface that attackers could abuse. Exploits against cloud infrastructures typically target *hypervisors (HVs)*—the software components responsible for ensuring isolation between cloud tenants—aiming to trigger either faults in their code, thus causing denial-of-service [4], [5], or security bugs, therefore breaking out of their sandbox (i.e., *Virtual Machine (VM)*) [6], [7], [8]. As such, securing hypervisors is of paramount importance for the sustainable functioning of the cloud.

Besides ensuring VM isolation, a hypervisor also has the major task of *virtualizing* the underlying hardware, which consists of emulating the behavior of the CPU and I/O devices (e.g., a network card). For that, CPUs implement a suite of events called *VM-exits* by Intel [1, Chapter 28] and *#VMEXITS* by AMD [2, Chapter 15.6] that are triggered on specific actions of its VMs (e.g., writing to an MSR) and redirect execution to the HV for handling. Due to the high amount of vulnerabilities in the VM-exit handling code of HVs [9], [10], [11], [12], [13], a significant amount of work has been put into scrutinizing them automatically via *fuzzing*. Notably, while most prior work focused on fuzzing the I/O emulation interfaces of HVs, i.e., PIO, MMIO, or Hypercalls [14], [15], [16], [17], [18], [19], [20], [21], [22], the interfaces that emulate CPU behavior, e.g., (reading/writing) MSRs, MMIO instruction emulation, APIC accesses, or hardware task switching have received significantly less scrutiny [23], [24], [25], [26], [27]. Surprisingly, although Intel defines 76 VM-exits [1] that could be triggered during VM execution, the interfaces fuzzed by prior work only account for 8 VM-exits (i.e., $\approx 10\%$), thus leaving a potentially large attack surface still unexplored—we tackle this issue in this work and focus on CPU emulation interfaces, as they represent the majority of the VM-exits.

This alarming gap is mainly due to the difficulty of generating valid VM states that can be used as fuzzing inputs to trigger specific VM-exits, as they must fulfill numerous requirements imposed by the architecture. Prior work [23] tackled this challenge by manually crafting initial architecturally valid VMs for a few hand-selected VM-exits. However, crafting these requires significant effort as their architectural requirements are scattered across several pages of the ISA specification and are often described ambiguously or inconsistently with the silicon implementation [27]. Further, most VM-exits may be exerted in several ways, implicitly affecting CPU-provided information for VM-exit handling (e.g., the `exit qualification` field), thereby exercising a different execution path in the HV. As such, we deem manually crafting valid VM states to cover the vast VM-exit space as infeasible, which was also admitted by the authors of prior work [23].

Network and Distributed System Security (NDSS) Symposium 2026
23 - 27 February 2026 , San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.231763
www.ndss-symposium.org

In our work, we tackle these issues with a novel hypervisor fuzzing approach—dubbed *HyperMirage*—which aims to craft fuzzing inputs automatically to reach unexplored HV code and find new bugs. Our key observation is that compiling architecturally valid VM states as fuzzing seeds is redundant for finding bugs in HVs. Instead, we introduce a novel technique called *Direct State Manipulation (DSM)* that generates input VMs automatically, and avoids running them during fuzzing, thus giving the HV the *illusion* that it did—hence the name *HyperMirage*. DSM's novelty lies in focusing its input mutations *directly* on the elements that are consumed by the HV while handling a VM-exit for a VM, which, according to the Intel manual, broadly consist of the VM's General Purpose Registers (GPRs), memory, and, most notably, the Virtual Machine Control Structure (VMCS).

Surprisingly, although the VMCS greatly influences execution in the HV, several of its crucial components were omitted during fuzzing in prior work, which may potentially miss security bugs in the HV. This is mainly because modifying a field in the VMCS typically requires executing (a sequence of) specific instructions in the VM, which may require significant manual labor. In this work, we shed light on the structure of the VMCS by systematically analyzing its specification in architecture manuals and identifying those fields that are relevant for finding bugs in HVs. We integrate this knowledge into DSM and configure it to redirect input mutations directly on these fields, as well as the guest's GPRs and memory, thus empowering HyperMirage to automatically explore the vast space of architectural VM states in depth. By doing this, DSM is able to reach architectural edge cases that reveal deep bugs in the HV. However, it also leads to impossible architectural states, which trigger false-positive crashes during fuzzing. We evaluate the occurrence of these cases and find that they can be trivially triaged.

Nevertheless, mutating the VM state randomly (in a *black-box* manner) or based on coverage feedback (in a *graybox* manner) is known to suffer from getting stuck at solving complex branch conditions (such as multi-byte equality checks). Prior work overcomes this issue via *hybrid fuzzing*, which combines concrete and symbolic execution (i.e., *concolic*) to systematically discover new execution paths in the analyzed target. However, while efficient hybrid fuzzing solutions have been employed on user space applications [28], [29], [30], [31] and OS kernels [32], [33], they have yet to be effectively applied to fuzz hypervisors. HyperFuzzer [23] proposed a custom hybrid fuzzing approach to fuzz Hyper-V, which, however, relies on a custom (closed-source) symbolic execution engine that employs a sub-optimal backend for solving symbolic expressions, affecting its fuzzing throughput. Our tool HyperMirage side-steps these limitations by providing the first hybrid fuzzing infrastructure that integrates SymCC [29], a state-of-the-art, highly optimized, symbolic execution engine, into a bare-metal coverage-guided hypervisor fuzzer. Out of the box, SymCC is built for user space programs, requiring a runtime-loaded dynamic library performing both symbolic expression generation and constraint solving with the help of an embedded SMT-solver, making it unsuitable for bare-metal targets. We tackle this issue by extending SymCC with a novel *symbolic record-and-replay* approach that enables instrumenting bare-metal targets and shifting the constraint solving to an environment outside of the target.

We equip DSM with symbolic record-and-replay and integrate them into Nyx [22], a state-of-the-art, high-throughput, snapshot-based, graybox fuzzer, enabling it to inject architectural events (i.e., VM-exits) into the target HV and perform fine-grained and efficient mutations on the fuzzing inputs. We then use the Xen [34] and KVM [35] hypervisors as case studies for HyperMirage, as they are both mature and open-source HVs that power several production cloud infrastructures. We find that HyperMirage identifies 9 new bugs in Xen and 2 in KVM, spread across four different components, even those already covered by Xen's in-tree fuzzing infrastructure as well as prior work on KVM. When compared to HyperPill [14], an open-source state-of-the-art HV fuzzer, we find that HyperMirage achieves 189% more coverage for Xen's as well as 79% more coverage for KVM's virtual CPU implementation. Even more, HyperMirage finds that previously untouched virtual CPU interfaces host comparable complexity to some of those hand-picked by prior work. Finally, HyperMirage achieves these results without manually specifying any expert-crafted fuzzing seeds.

To summarize, we make the following contributions:

- We present HyperMirage—a novel hypervisor fuzzer that relies on *Direct State Manipulation (DSM)* to generate fuzzing inputs by directly manipulating the VM state that is consumed by hypervisors, capable of exploring its sub-states in depth and breadth.
- A novel hybrid fuzzing infrastructure for testing hypervisors efficiently; we extend SymCC, a highly-optimized state-of-the-art symbolic execution engine, and we integrate it into Nyx, a state-of-the-art coverage-guided hypervisor fuzzer; to the best of our knowledge, we are the first to leverage a symbolic execution engine that can instrument hypervisor code and run natively (i.e., without emulation).
- A case study and evaluation for HyperMirage on the Xen and KVM hypervisors for Intel CPUs, demonstrating that it can fuzz 200% more VM-exit interfaces than prior work and achieves 189% higher coverage for Xen's and 79% for KVM's virtual CPU implementation when compared to an open-source state-of-the-art HV fuzzer. HyperMirage discovers 9 new bugs in Xen and 2 in KVM, all of which have been confirmed by the respective project maintainers.

II. CPU Virtualization Extensions

In this section, we give a primer on virtualization extensions as implemented by the Intel x86 architecture (the most widespread), aiming to outline the interactions between VMs and HVs, as well as the architectural structures populated by the former and consumed by the latter during VM-exits, which HyperMirage targets to find deep bugs in HVs.

Intel calls its virtualization extension Virtual Machine Extensions (VMX), AMD calls them Secure Virtual Machine (SVM), while ARM calls them Virtualization Extensions (VE). As the underlying fuzzing infrastructure (i.e., Nyx) leveraged by HyperMirage is optimized for Intel CPUs, we focus on Intel VMX in this paper. However, in Section VII, we provide a discussion on the challenges of porting HyperMirage to AMD SVM and ARM VE.

VMX introduces a new operation mode called *root* mode, which follows the traditional *ring*-based privilege separation, and is reserved for executing the HV and the privileged host OS (e.g., Xen's Dom0 and QEMU). Unprivileged VMs run isolated in *non-root* mode, where they can execute arbitrary kernel code in ring 0, switch to long-, protected-, or real-mode, or any other architecturally-valid runtime state, independently from the HV or the other VMs.

The VMCS is a central component of a HV's interaction with a VM under Intel VMX. It is responsible for transitioning to non-root mode (VM-entry) and back into root mode (VM-exit). It also enables the HV to define the isolation sandbox for guest VMs and configure the CPU behavior while executing them in non-root mode. Structurally, the VMCS is a 4KB array of integers of different widths, whose layout is opaque to the HV, as they can only be accessed via the special `VMREAD` and `VMWRITE` instructions. Broadly, the VMCS maintains four different types of fields:

1) **Guest-State (`VMCS[GS]`):** Configures the initial state of a VM and temporarily saves it during the handling of a VM-exit. Most notably, the guest VM can modify its own state or execute certain instructions that directly or indirectly manipulate `VMCS[GS]`, thus making it a crucial element for HyperMirage. It is subdivided into two categories: the *guest register state* (`VMCS[GrS]`) and the *guest non-register state* (`VMCS[GnrS]`). Except for general-purpose registers (`GPRs`), which have fall-through semantics, `VMCS[GrS]` saves all of the guest's CPU registers during VM-exit, i.e., RSP, RIP, RFLAGS, control registers (CRs), debug registers (DRs), segment registers (e.g., SS, CS, DS, etc.), as well as some MSRs. In contrast, `VMCS[GnrS]` holds fields that are partly populated by the CPU on specific VM actions (such as setting the *interruptibility state* on executing the `STI` instruction), and partly by the HV (such as *VMX-preemption timer value*).
2) **Host-State (`VMCS[HS]`):** Configures the HV state restored during a VM-exit, such as the instruction address of the VM-exit handling function. The `VMCS[HS]` is not interesting to HyperMirage as it cannot be manipulated by the VM to trigger bugs in the HV.
3) **VMX Controls (`VMCS[CTRL]`):** Governs CPU behavior for non-root mode execution, VM-exits, and VM-entries, and are configured by the HV. Alone, the non-root mode execution controls host 64 different execution control bits across one 64-bit and three 32-bit vectors, as well as several sub-structures that govern if certain actions in the VM should trigger a VM-exit. Specifically, `VMCS[CTRL]` maintains memory pointers to 8 bitmaps (e.g., the exception and the MSR bitmap), with each bit instructing the CPU to trigger a VM-exit if the corresponding event takes place in the guest VM. For example, if a bit in the exception bitmap is set, then any corresponding exception vectors issued in the guest VM will be intercepted by the HV. In total, `VMCS[CTRL]` consists of 31 different categories. Interestingly, some of these may not be implemented by the HV, or may not be configurable (e.g., due to a missing HW feature). Those that are, however, must be explicitly configured by the management interface of the HV (i.e., Dom0 for Xen, QEMU for KVM) and require a clear understanding of their semantics—all of which put an extra burden on a security analyst.
4) **VM-exit Information (`VMCS[VEI]`):** Provides additional information that the HV may use to handle the VM-exit, and is populated by the CPU. `VMCS[VEI]` is highly complex as it comprises of 5 different categories with rich semantics, all of which may be influenced by a VM. The first category holds basic information about the VM-exit, such as the *exit reason*, an *exit qualification*, and the guest linear or physical address that led to the VM-exit. Intel defines 76 exit reasons and provides 19 additional exit qualifications, each associated with one or more VM-exits. Interestingly, 7 exit qualifications are further subdivided into several fields of different bit lengths, providing a large permutation of values. The other categories of the `VMCS[VEI]` also have rich semantics. For example, the information for VM-exits due to instruction execution provides an *instruction length* field and an *instruction information* vector that can have 8 different formats depending on the instruction that triggered the VM-exit.

To enable efficient virtualization of VM memory (`MEM`), Intel introduced the concept of Second Level Address Translation (SLAT) in the form of Intel Extended Page Table (EPT). This enables a VM to independently manage its own page table without interference by the HV, as physical memory isolation is provided by the SLAT, which directly map guest physical addresses to host physical addresses.

Combining the aforementioned concepts, the lifecycle of a VM under Intel VMX is depicted in Figure 1. First, the HV allocates memory and configures it as physical memory for a VM via EPTs, as well as allocates and configures a VMCS ❶. Then, the HV launches the VM by executing the `VMLAUNCH` instruction ❷. Implicitly, the CPU initializes itself as per the `VMCS[GS]` fields. At this point, the CPU natively executes all code in the context of the VM in non-root mode ❸. As soon as the VM performs a privileged operation that the HV needs to intercept (e.g., a write to an MMIO region), a `VM-exit` is triggered, where parts of the current VM state are stored in the `VMCS[GS]` fields, and the pre-configured HV state (e.g., the instruction pointer) is loaded from `VMCS[HS]`. The VMs GPRs simply fall through and have to be preserved by the VM-

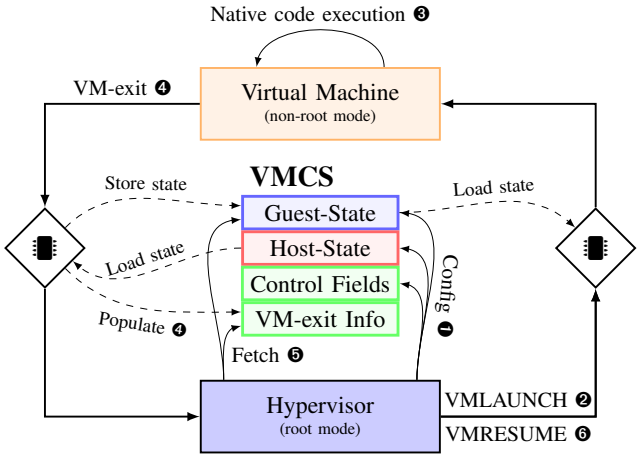


Fig. 1: Lifecycle under Intel VMX.

exit handler. Importantly, the CPU also implicitly populates the `VMCS[VEI]` fields ❹ with details on the exact action that resulted in the VM-exit. Finally, the HV will handle the VM-exit (e.g., validate and perform the write to the emulated MMIO device) by reading out fields of the VMCS ❺ and eventually resuming execution in the guest ❻.

III. Motivation

We motivate the design of HyperMirage and its DSM approach two-fold. First, the architectural complexity of Intel VMX, as introduced in the previous section and documented across several pages in the Intel manual, fundamentally hinders scalability in fuzzing HVs via manually generated seeds. Second, a real-world vulnerability discovered by HyperMirage is triggered by an unusual VM state that would be hard for a human analyst to generate.

A. Architectural Complexity of Intel VMX

Architecturally under Intel VMX, VM-induced HV code execution is solely performed during handling of VM-exit events. As such, to maximize fuzzing performance, a fuzzer shall be able to repeatedly trigger as diverse VM-exits as possible. Ideally, to maximize code coverage, a virtual CPU fuzzer shall be able to fuzz all existing VM-exits. However, achieving this in practice is challenging since Intel VMX defines in total 76 different VM-exit reasons, all triggered by drastically different VM states. Moreover, a majority of VM-exits host further `VMCS[VEI]` fields as part of the HVs VMCS, describing in detail the VM interaction resulting in the VM-exit. For instance, fully exerting all possible values of the `exit qualification` field for the `control-register access` VM-exit already requires 131 different VM states. This is without accounting for further VM-exit handling complexity depending on various other state, such as the VM's execution mode. Automatically generating VM states that, when executed, exert all possible values for the per VM-exit information fields remains unrealistic as their individual meaning is described across various pages in the Intel Software Developer Manual (SDM), which is not suited for automated analysis. Even worse, a specific HVs configuration of the `VMCS[CTRL]` fields may disable certain VM-exits entirely (e.g., control register accesses), requiring even more manual labor to bring the HV to a state suitable for fuzzing.

We make the observation that during the handling of VM-exits, the HV interprets the current state of the offending VM in four ways: the VM's `GPRs`, parts of the VMCS: `VMCS[GS]` and `VMCS[VEI]`, as well as the VM's memory (`MEM`). As such, a methodology is required that automatically exerts the HVs *view* on the aforementioned components without requiring the execution of a corresponding VM in hopes of bringing forth the desired values. This naturally encompasses all possible states of a VM, including complex setups such as different execution modes. Equipping a fuzzer with the capability of directly modifying hardware structures such as the VMCS brings forth the possibility of exerting the HV with architecturally unusual or even invalid states. In the following subsection, we motivate that this is in fact a desired property.

B. Real-World Vulnerability

Although unintuitive, we now highlight the fact that *architecturally unusual* fuzzing inputs are in fact a desired property, on account of a real-world vulnerability in the Xen hypervisor, discovered by HyperMirage: **CVE-2023-46842**. Initiating this vulnerability requires a malicious VM to prepare an *architecturally unusual* register state before performing a hypercall in 32-bit mode with a particularly long execution time. The long execution time forces Xen to create a *hypercall continuation*, storing the hypercall's arguments (i.e., the VM's GPRs) for later resumption. During the resumption of the continuation, Xen performs consistency checks on the stored register state, asserting that none of the upper 32 bits are set. If the validation fails, Xen panics due to the presumed inconsistent state.

The assumption that GPRs coming from a 32-bit hypercall must have its upper 32 bits cleared is, in principle, reasonable, as code executing in 32-bit mode is unable to access the upper 32 bits of its GPRs. However, a malicious VM is able to deliberately prepare its GPRs accordingly while executing in long mode (i.e., 64-bit mode) before transitioning back to 32-bit mode. Interestingly, the Intel SDM makes no statement on what happens to the higher halves of GPRs during such a transition. Behavior is *only* defined for GPRs that are available in long mode (i.e., R8-R15 as well as certain SIMD registers) [1, Chapter 10.8.5.4]. On all CPUs we tested, the higher halves were preserved, leading to a denial-of-service of the entire HV.

The fact that hypervisor developers make assumptions about architectural correctness in certain edge cases motivates the design of DSM. Specifically, DSM exercises the HVs VM-exit handler with VM states that may or may not be architecturally valid. This design makes sure that no artificial constraints on fuzzing inputs are imposed that may leave HV bugs, based on incorrect architectural assumptions, undiscovered. Verification work on architectural validity is imposed on human analysts

only when the fuzzer has already revealed that specific states would trigger unwanted behavior in the HV. This specific design led to the aforementioned vulnerability being discovered and reproduced during the early prototyping of HyperMirage.

IV. Design

In this section, we introduce the design of HyperMirage. We start out by describing the overall hybrid fuzzing setup leveraged by HyperMirage to test HVs efficiently. Then, we present DSM, a novel technique that enables fuzzing the HV's entire view of a VM in a holistic, fine-grained manner. Finally, we describe our novel symbolic record-and-replay approach that enables SymCC [29] to perform efficient symbolic execution on bare-metal software, including hypervisors.

A. Threat Model

We assume a standard multi-tenant cloud scenario, where an attacker is able to control the entire execution of its unprivileged guest VM running atop a potentially vulnerable HV. The attacker's goal is to exploit a vulnerability in the HV to (1) break out of their isolated VM in order to elevate their access to the HV or other isolated VMs or (2) disrupt the HV's execution, thereby causing a denial-of-service. To achieve that, the attacker may bring its VM into an arbitrary architectural state, and issue an arbitrary number of VM-exits, in an arbitrary order, to trigger a bug in a vulnerable VM-exit handler. To facilitate fine-grained and efficient input control during fuzzing, we grant HyperMirage direct access to the guest-controlled portions of the VMCS (i.e., `VMCS[GS]` and `VMCS[VEI]`), typically jointly managed by the HV and the CPU and only indirectly influenced by its VMs.

B. Fuzzing Overview

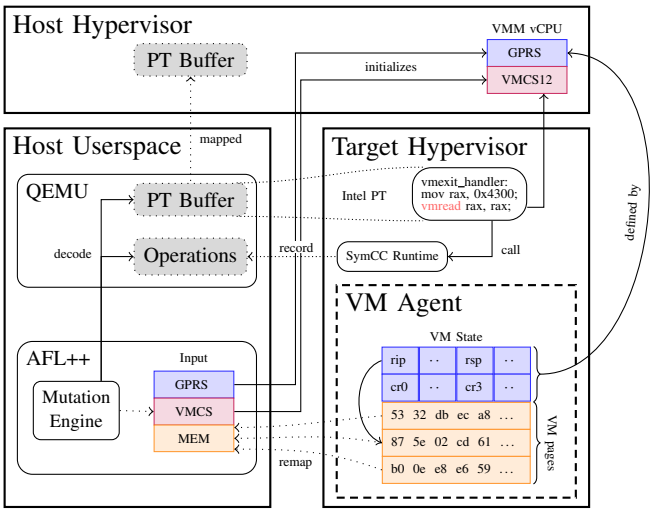


Fig. 2: Overview of HyperMirage's design.

Figure 2 provides an overview of the architecture of HyperMirage. HyperMirage models its fuzzing inputs in the form of a VM's four components: `GPRs`, `VMCS[GS]`, `VMCS[VEI]` and `MEM`. As discussed in Section III, this encompasses the HV's entire view of its VMs during VM-exit handling. To directly impose this view, without the execution of a matching VM, we propose a new HV fuzzing methodology: DSM. To enable DSM, HyperMirage leverages nested virtualization for the target HV, controlled by a *host hypervisor*. Moreover, to bring the HV into a state ready for VM-exit handling, HyperMirage requires an initialization phase, where an unprivileged VM agent is spawned, which performs a handshake with HyperMirage and creates a snapshot of the target HV. The requirements of the VM agent are detailed in Section V.

During the fuzzing loop, the fuzzing control (AFL++) generates a new input through random mutations or symbolic execution. Then, the snapshot is restored, ensuring that each fuzzing execution starts from a clean state, after which the VM Agent immediately triggers a special hypercall. This signal is recognized by the host HV, which has the capability to directly apply the fuzzing input to the target HV's *view* on the VM agent via DSM (see Section IV-C). Finally, the host HV resumes code execution at the target HV's VM-exit handler.

To perform mutations on the fuzzing inputs, HyperMirage deploys hybrid fuzzing. Coverage-guided random mutations (i.e., graybox fuzzing) are required for easy-to-reach paths, as well as the discovery of violations that are not directly represented in the program's source code (e.g., out-of-bounds memory accesses). Symbolic execution (i.e., whitebox fuzzing) is necessary to compute solutions for hard-to-reach paths that are hidden behind a chain of complex constraints, which are unlikely to be satisfied by random mutations.

To obtain coverage guidance for graybox fuzzing, we record and parse an execution trace of the target HV via Intel's processor trace (Intel PT) functionality and forward the coverage bitmap to AFL++. This approach has been proven effective in previous work [36], [22]. For whitebox fuzzing, we perform compilation-based symbolic execution by devising a new runtime for SymCC that can be deployed on bare-metal targets such as HVs. During execution of a fuzzing input, any VM state accessed by the target HV is symbolized, and any operations on symbolic values are recorded for later decoding.

Combining the aforementioned concepts, we obtain the hybrid fuzzer for virtual CPUs: HyperMirage.

C. Direct State Manipulation

As described in Section III, exhausting the VM-exit handler of a HV in depth by executing specially prepared VMs is infeasible. As such, in contrast to previous work [24], [27], [21], [22], [23], we introduce DSM to directly exert the HVs *view* on its VM's four crucial components.

Accordingly, we design the format of HyperMirage's fuzzing inputs as depicted in Figure 3. Every fuzzing input incorporates all `GPRs`, `VMCS[GS]`, `VMCS[VEI]`, and a region for `MEM`. Most notably, we merely reserve a small, pre-defined number of bytes (512) for the memory region. When testing an input, we fill up the memory page that hosts the fuzzing input by repeatedly overwriting the page with the `MEM` portion until the entire page is covered. All memory belonging to the VM agent is then remapped to this singular page. This lets

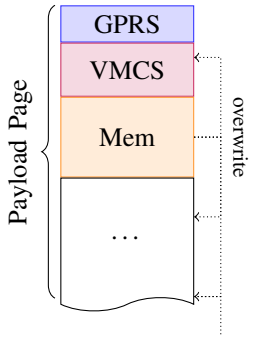


Fig. 3: Fuzzing input format of HyperMirage.

us keep our fuzzing input size relatively low (< 1024 bytes), while making sure that every VM memory read initiated by the HV is retrieving bytes that are defined by the fuzzing input. We specifically choose this repeating memory pattern to allow unique byte combinations *across* page boundaries.

The fuzzing input is applied to the VM-exit handler of the target HV as follows: Under Intel VMX, the host HV controls the VMCS of all nested HVs (i.e., the VMCS of the target HV for the VM agent, depicted as `VMCS12` in Figure 2). As such, the VMCS-related fields can be directly applied to the `VMCS12`. Similarly, the host HV defines the virtual CPU for the target HV and can, therefore, freely overwrite its `GPRs` before resuming code execution at its VM-exit handler. Finally, to redirect memory fetches, all physical memory belonging to the VM agent is remapped in the SLAT governed by the host HV. After remapping, all pages designated for the VM agent will point to the page hosting the current fuzzing input, which has been expanded as explained previously.

In the end, DSM has three distinct advantages: First, every single tested input will trigger an interaction with the HV (as a VM-exit is forcibly injected). Secondly, DSM has the capability to explore every single virtual CPU interface of the HV, and third, bypassing architectural requirements implies that no hand-crafted fuzzing seeds must be specified.

D. Symbolic Record-and-Replay

To integrate high-performance symbolic execution into HyperMirage, we turn to SymCC, a state-of-the-art compiler-based symbolic execution framework. SymCC instruments the analyzed target during compilation to report any constraints imposed on symbolic inputs via calls to an injected runtime library. In vanilla SymCC, this runtime library takes the form of a dynamic library, implicitly loaded on process startup, that packages an SMT solver (e.g., Z3 [37]) to solve symbolic constraints. As HyperMirage deals with bare-metal software (i.e., hypervisors), this design is not directly applicable. First, bare-metal environments, in general, do not support dynamic loading of libraries. Second, an SMT solver such as Z3, designed for user space applications, will itself expect a managed environment supporting dynamic memory allocations, file system support, etc. While the first is trivial to solve by linking the runtime statically, the latter is a fundamental limitation unless the SMT solver is completely re-tailored to the environment of the bare-metal software.

We solve the second issue in a more generic manner via a *symbolic record-and-replay* design that is specifically tailored to analysis targets, which are unable to directly incorporate an SMT solver. As such, the analysis target is only instrumented to record symbolic operations occurring at runtime, such that they can be retrieved at a later point in time and replayed to an SMT solver hosted in a less restricted environment. To this end, we split the runtime library into two components: the frontend that is statically compiled into the target HV and the backend that lives in the host HV.

The frontend merely stores the type of symbolic operation and its parameters encountered at runtime in a designated memory region. As the frontend is unable to access the SMT solver, it utilizes place-holder values for all symbolic expressions, following a Static Single-Assignment (SSA) form. The place-holder is simply an increasing counter, starting at 1. We specially reserve the *null* place-holder to represent concrete values. This enables SymCC to perform its concreteness checks directly in the instrumented code, drastically improving performance. Further, we need only mark those values as symbolic that are directly retrieved from the VM's state. We identify those values as follows. `GPRs` follow fall-through semantics, where the HVs VM-exit handler typically directly pushes them to the stack, and passes a pointer to the respective stack region directly to the C function of the VM-exit handler. All `VMREAD` instructions reading from `VMCS[GS]` and `VMCS[VEI]`, naturally receive values from the relevant VMCS fields. Finally, HVs implement a set of functions to fetch VM memory. Instrumenting them to symbolize their destination addresses after every function call serves to symbolize all fetched VM `MEM`.

As soon as an input has finished executing, the backend starts decoding and replaying the recorded operations. This, in principle, follows the original SymCC approach. Special handling is required for the introduced place-holder values. During replay, we create a hashmap that maps place-holders to their now-known symbolic expressions. Whenever a constraint is solved for, HyperMirage forwards the required modification of the fuzzing input to AFL++ to perform the guided mutation.

Component	Language	LoC
Direct State Manipulation	C	2600
SymCC Compiler Pass	C++	47
SymCC Runtime	C & C++	1977
AFL++ Symbolic Mutator	C++	318

TABLE I: LoC of HyperMirage's implementation.

V. Implementation

In this section, we discuss the implementation details of HyperMirage. In particular, we want to highlight the relative simplicity of our implementation when compared to state-of-the-art. The lines of code for each of our components are depicted in Table I.

A. Direct State Manipulation

We implement our novel DSM approach into the Nyx [22] framework, a snapshotting-based, high-throughput graybox fuzzer for HVs. We choose Nyx over more recent HV fuzzers, such as HyperFuzzer [23] or HyperPill [14], since they are closed-source or use emulation instead of native execution, respectively. As such, the utilized host HV, as well as QEMU in Figure 2, are modified versions of KVM-PT and QEMU-PT as introduced in [22]. VMCS modification capabilities and capabilities to catch VMRESUME events (thereby indicating the finished VM-exit handling of the target HV) are implemented directly inside of KVM-PT. The VM agent follows the requirements of Nyx, i.e., it performs necessary hypercalls to take the snapshot that is restored upon every fuzzing run.

The memory remapping, as described in Section IV-C, is performed on the EPT level of the host HV (L0), which is responsible for making physical memory available to the target HV (L1). In turn, the target HV (L1) hosts its own set of EPTs to allocate physical memory for the VM agent (L2). The VM agent itself once again hosts its own page table (PT), mapping L2 virtual addresses to L2 physical addresses. In essence, to appropriately remap memory in the EPTs of L0, we must first translate accessible L2 virtual addresses to their corresponding L2 physical addresses. L2 physical addresses are then mapped to L1 physical addresses via L1's EPTs. Finally, with the acquired L1 physical addresses, it is possible to adjust L0 EPTs to remap all those to the same page that hosts the fuzzing input. The parsing of the L1 EPTs, as well as the L2 PTs, is implemented in QEMU-PT. L0 EPT remapping is performed via `ioctls` to KVM-PT.

In the end, this memory remapping avoids hooking or modifying the memory accessing functionality of the target HV. Instead, all memory reads and writes are forced to directly interface with the fuzzing input without making the target HV aware of this fact. For efficiency reasons, the memory remapping is performed once during the snapshotting phase.

B. Hybrid Fuzzing Integration

We integrate our whitebox fuzzing approach based on SymCC into AFL++ [38] in the form of a custom mutator plugin. Where applicable, we follow the implementation of the SymQEMU custom mutator [39]. This is, by their own rights, preferred to the SymCC custom mutator, also part of the AFL++ repository [40]. Custom mutators are alerted whenever an input from the fuzzers' input queue is picked for mutation by AFL++. At this point, the SymQEMU heuristic is applied, such that all inputs are only symbolically analyzed exactly once. During symbolic analysis of an input, a number of new diverging inputs are generated by applying solved-for modifications, and the new inputs are retrieved by AFL++. In turn, these newly generated inputs may trigger new coverage and be placed into the queue of interesting inputs. Utilizing this approach requires no code modifications to AFL++.

To perform symbolic testing of an input, the mutator spawns its own parallel copy of the entire fuzzing setup, with the distinct difference of booting an instrumented version of the target HV. In essence, the SymCC-related blocks depicted in Figure 2 are only present in this context. This ensures that graybox fuzzing is not needlessly slowed down when no symbolic execution is performed.

```
1  SymID _sym_build_add(SymID a, SymID b) {
2    if (unlikely(!trace_enabled))
3      return;
4
5    submit_operation(OP_ADD);
6    submit_value(a);
7    submit_value(b);
8    /* Represents symbolic return value */
9    submit_value(placeholder_id++);
10 }
```

Fig. 4: HyperMirage's SymCC frontend handler for additions.

C. Record-and-Replay SymCC Runtime

As introduced in Section IV, we devise a new runtime design for SymCC, split between its frontend and backend. Every frontend handler first checks whether symbolic tracing is enabled. Symbolic tracing is enabled by the host HV overwriting a global variable in the target HV when a fuzzing input is injected. Then, all values relevant to the operation associated with the handler are pushed onto the shared memory region. These values are operation identifiers, symbolic expressions, or concrete values. Importantly, symbolic expressions are, at this point, substituted with place-holder IDs following a SSA form. An example implementation for symbolic additions is shown in Figure 4. *SymID* represents the type for place-holder IDs (a typedef for 16-bit unsigned integers), *trace_enabled* controls whether or not symbolic execution shall be performed, i.e., it is disabled during the snapshotting process. The *submit_** invocations simply store the value in a designated memory region. Importantly, adding two symbolic values creates a new symbolic value, for which a new place-holder ID is returned.

When the backend gets to decoding and replaying the previously stored `ADD` operation, it first translates *a* and *b* into their actual symbolic values and forwards them to the symbolic add handler of the vanilla SymCC backend, in our case, the QSYM backend, which hosts a plethora of optimizations to minimize time spent in constraint solving. The place-holder ID of the return value is then associated with the newly retrieved symbolic expression from QSYM. In our prototype implementation, the backend is part of the host QEMU process. Here, the designated memory region with all recorded operations is mapped into QEMU's own virtual address space to avoid needless copying.

D. Bare-metal SymCC Challenges

Interrupts. One key issue with applying our record-and-replay SymCC runtime to bare-metal software is the inherent preemptibility of modern HVs. As such, it is possible that at any time during the execution of a frontend handler, an interrupt fires and code execution resumes at the interrupt handler. As no selective instrumentation is performed on the target HV, the

interrupt handler itself now records symbolic operations to the designated memory region, resulting in incoherent recorded traces. Instead of disabling interrupts entirely during fuzzing, which may alter regular execution of the target HV and possibly introduce deadlocks, HyperMirage deals with this issue by temporarily disabling trace generation (by resetting the `trace_enabled` global variable) as soon as an interrupt reaches the target HV.

Inline Assembly. Bare-metal software frequently deploys a variety of special instruction accesses (e.g., `rdmsr`), optimizations, or security mitigations (e.g., Spectre retpolines [41]) through invocations of inline assembly blocks. As inline assembly blocks are not represented in LLVM IR, SymCC fails to instrument their semantic meaning, thereby losing soundness and failing to accurately recover constraints. To combat this, we manually identify inline assembly constructs that are relevant to VM-exit handling (e.g., Xen's `alternative_vcall` mechanism, which is utilized to replace indirect calls to VMX/SVM specific handlers with direct calls after boot) and analyze every encountered inline assembly block during instrumentation. If a known construct is recognized, we emit appropriate invocations to our record-and-replay frontend that correctly symbolize the respective inline assembly operations. Note that prior work utilizing KLEE [42] (an LLVM-based symbolic execution engine) on the Linux kernel has addressed this challenge in a similar manner [43].

VI. Evaluation

We evaluate HyperMirage by answering the following research questions:

- **RQ1** How does HyperMirage compare to state-of-the-art hypervisor fuzzers?
- **RQ2** How effective is DSM in exploring previously untouched virtual CPU interfaces?
- **RQ3** Is HyperMirage able to discover novel vulnerabilities in battle-tested hypervisors?
- **RQ4** Does DSM result in false positive crashing inputs, and do they hinder triaging?

A. Experiment Setup

All experiments and fuzzing campaigns are performed on a desktop machine with an Intel Core i7-6700 and 16GB of DDR4 RAM. To evaluate HyperMirage, we implement prototypes for the two popular open source hypervisors Xen and KVM, targeting their latest major release build at the time of writing: Xen v4.20.0 and KVM v6.15.0.

We choose to evaluate HyperMirage against the hypervisor fuzzer HyperPill [14], as (1) it presents the most recent advances in fuzzing hypervisors, (2) is openly available, and (3) is capable of fuzzing a subset of virtual CPU interfaces (i.e., MMIO, MSR writes[1], port I/O, and hypercalls). Since HyperPill's focus lies primarily on fuzzing virtual devices, a better fit would have been HyperFuzzer [23], a hybrid fuzzer for virtual CPUs, which is unfortunately neither open source nor does it evaluate any open source HVs, making a fair comparison to HyperMirage impossible. Nonetheless, we provide a best effort performance evaluation against HyperFuzzer in the Appendix.

We execute HyperMirage and HyperPill for 24 hours as recommended in [44] and collect average execution times per fuzzing input as well as the amount of covered edges. For simplicity, we restrict both of their execution to a single core[2]. We provide HyperMirage with a single fuzzing seed, initialized with random data, to emphasize the fact that HyperMirage requires no manually-crafted fuzzing seeds. Further, for HyperMirage we perform the experiment once in pure graybox mode, i.e., without the symbolic execution component, and once in hybrid mode. To force HyperPill to focus its fuzzing efforts on the virtual CPU implementation of the HV, we restrict its coverage range to the `.text` sections of Xen and the KVM modules, respectively. To obtain accurate coverage and performance numbers, we disable any supported sanitizers and memory leak detectors for these experiments.

B. Coverage

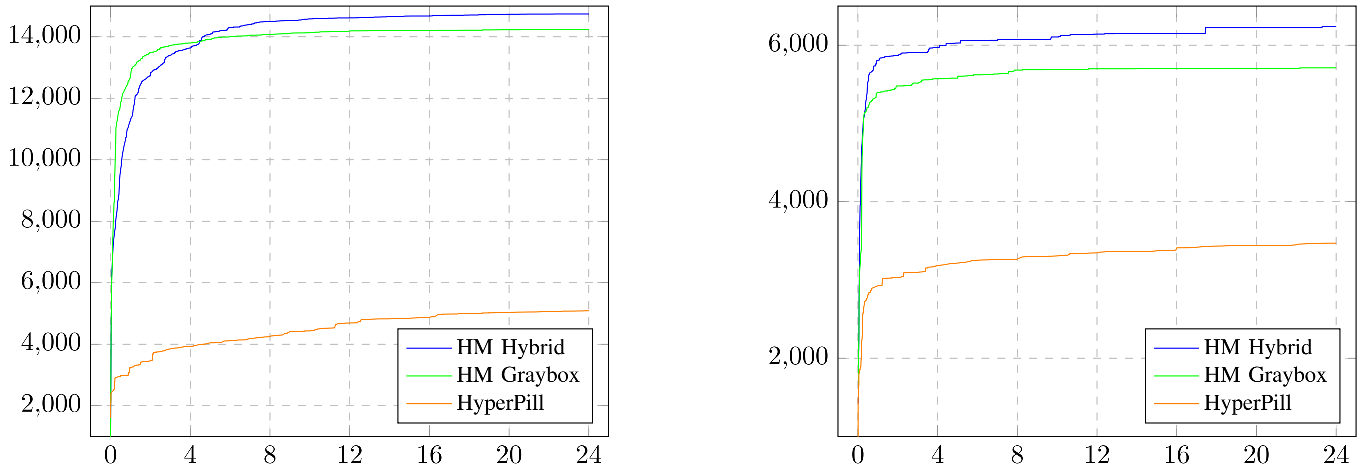
In this section, we explore HyperMirage's edge discovery capabilities. Specifically, we first evaluate HyperMirage's efficacy in achieving HV coverage when compared to HyperPill. Then, we turn towards the evaluation of HyperMirage's coverage across all VM-exits implemented by Xen and KVM, i.e., those that have not yet previously been fuzzed.

We plot the total coverage of both HyperMirage and HyperPill in Figure 5. Further, we obtain per-VM-exit coverage for HyperMirage during the hybrid fuzzing runs by counting occurrences of the current value of the `exit reason` field of the fuzzing inputs' `VMCS[VEI]` whenever a new edge is discovered. To minimize over-counting of edges that are not VM-exit specific (i.e., code present at the beginning of the VM-exit handler before distinguishing on the exit reason), we make sure the randomized seed for HyperMirage hosts an undefined exit reason. All edges discovered by an input hosting an undefined exit reason in its `VMCS[VEI]` are simply ignored during counting. These results are depicted in Table II. Note that some building blocks can, in principle, be reached from multiple VM-exits (e.g., the instruction emulator is exerted by both APIC Access and EPT Violation VM-exits), yet we are only counting their first occurrence (which might lead the coverage statistics for those VM-exits to appear low). Furthermore, we analyzed the code base of both Xen and KVM to classify whether specific VM-exit reasons have no handling code at all (●), contain only trivial code that is not expected to exhibit coverage (❍), or are unreachable due to missing CPU feature support (◗).

1) Total Coverage Comparison: Figure 5 shows that HyperMirage achieves drastically more coverage than HyperPill for both Xen and KVM. HyperMirage achieves 189% more coverage for Xen and 79% for KVM. Analyzing Table II to

[1]While HyperPill hosts code for MSR reads, it is currently unused.

[2]In reality, both projects are parallelizable up to the number of cores available on the system.



(a) Xen (b) KVM

Fig. 5: Discovered edges of HyperMirage (HM) and HyperPill. The y-axis depicts discovered edges. The x-axis depicts the time in hours.

	EPT VIOLATION	VMCALL	MSR WRITE	MSR READ	ACCESS LDTR OR TR	IO INSTRUCTION	VMLAUNCH	APIC ACCESS	RDTSCP	CPUID	VMXOFF	TASK SWITCH	CR ACCESS	INVPCID	GETSEC	MCE DURING VMENTER	TRIPLE FAULT	VMXON	PAUSE INSTRUCTION	HLT	XSETBV	VMPTRLD
Xen	8086	2971	458	323	235	231	178	136	124	119	112	111	78	●	❍	72	43	42	40	26	23	21
KVM	2365	222	1531	483	0	62	1	67	❍	64	17	173	182	117	●	❍	❍	6	2	60	17	21

	VMREAD	EPT MISCONFIG	EXTERNAL INTERRUPT	INVEPT	INVLPG	VMWRITE	VMCLEAR	INVVPID	VMRESUME	VMPTRST	DR ACCESS	XRSTORS	NOTIFY	INVD	MSR LOADING	PENDING VIRT INTR	RDTSC	VMX TIMER EXPIRED	VMFUNC	INIT	IO SMI	PENDING VIRT NMI
Xen	20	20	14	13	12	12	11	8	7	5	5	❍	5	4	4	❍	3	●	❍	❍	●	❍
KVM	13	44	❍	47	7	12	24	39	12	22	30	❍	12	❍	●	13	❍	4	0	●	●	2

	RSM	INVALID GUEST STATE	MWAIT INSTRUCTION	MONITOR TRAP FLAG	MONITOR INSTRUCTION	TPR BELOW THRESHOLD	ACCESS GDTR OR IDTR	WBINVD	XSAVES	BUS LOCK	EXCEPTION NMI	SIPI	OTHER SMI	RDPMC	EOI INDUCED	APIC WRITE	RDRAND	ENCLS	RDSEED	PML FULL	UMWAIT	TPAUSE
Xen	●	❍	❍	2	❍	❍	1	1	❍	❍	0	❍	●	●	◗	◗	●	●	●	0	●	●
KVM	●	●	2	❍	2	❍	0	1	❍	❍	33	●	●	2	◗	◗	❍	◗	❍	12	❍	❍

● VM-exit not implemented by the HV.
❍ VM-exit handler that does not exhibit coverage.
◗ Missing hardware support on the experiment machine.

TABLE II: Per-VM-exit edge discovery of HyperMirage.

figure out where HyperMirage acquires the majority of its coverage, we find that the vast majority of edges for Xen stem from the EPT Violation (i.e., MMIO) as well as VMCALL (i.e., hypercall) VM-exits. For KVM, we find EPT Violation, as well as MSR write emulation, to be the largest coverage trigger. As these are all VM-exits supported by HyperPill, we derive that the advantages of HyperMirage do not only stem from the ability to fuzz a wider range of VM-exits, but also from fuzzing them more effectively. This is particularly interesting, as HyperPill contains manual specifications for its supported interfaces, whereas HyperMirage is not aware of any particular interfaces at all. In the following, we explore these three interfaces in more detail to provide insights into why we believe HyperMirage outperforms HyperPill.

EPT Violation (similarly EPT Misconfiguration and APIC Access) VM-exits are utilized to intercept accesses to MMIO regions. In order for a HV to perform the MMIO access, it needs to decode and emulate the offending memory accessing instruction, which brings forth a huge attack surface. For HyperPill, we find that every single MMIO interaction is built with eight variations of the same `MOV` instruction, varied only by direction of access (i.e., read/write) as well as operand size (i.e., 1, 2, 4, or 8-byte access). While this approach is suitable for fuzzing underlying emulated devices (typically implemented in user space), it falls short in achieving coverage in the x86 emulator. In contrast, HyperMirage gets direct feedback from the constraints imposed by the decoding component of the instruction emulator and can leverage DSM to replace the decoded instruction as seen by the HV, therefore effectively exploring the entire x86 instruction emulator.

For hypercalls, symbolic execution is expected to be much more effective in accurately deducing the utilized calling convention as well as available command and sub-command values. While HyperPill deploys an ad-hoc mechanism for a similar purpose, it relies on filling GPRs with random values and recognizing them in operands of `CMP` instructions during execution of a fuzzing input. While suitable for simple equality constraints, this approach falls short when non-equality comparisons are performed or register values are either masked or truncated before being compared to. This is the case e.g., for compatibility mode hypercalls, which mask out the upper halves of the GPRs before interacting with them.

For the MSR write emulation interface, we apply reasoning similar to that of hypercalls. An effective fuzzer needs to be able to enumerate all MSRs that are emulated by the target HV as well as any constraints imposed on the supplied MSR

values. Once more, the ad-hoc mechanism deployed by HyperPill falls short in recovering non-trivial constraints, whereas HyperMirage discovers solutions even for complex constraints, due to its integration of symbolic execution. Finally, the much higher throughput of HyperMirage (see Section VI-C) inevitably contributes to faster discovery of new edges.

2) Per VM-exit Coverage: As HyperMirage and DSM are specifically designed to be able to holistically target all VM-exits supported by the target HV, we now evaluate how well HyperMirage explores previously untouched virtual CPU interfaces. This is a key limitation in previous approaches [21], [23], [14], which hand-picked and restricted their evaluation to the interfaces hypercalls (VMCALL), task switch, port I/O (I/O Instruction), MMIO (EPT Violation / EPT Misconfiguration / APIC Access), and MSR emulation.

First and foremost, we see that MMIO-related VM-exits (i.e., EPT violation, but note that coverage may similarly be exerted by EPT Misconfiguration or APIC access VM-exits), MSR emulation, and VMCALL VM-exits make up 86% of all discovered edges for Xen and 77% for KVM, thereby justifying the focus of prior manually-driven approaches on these interfaces. Conversely, we find a plethora of virtualization interfaces that host similar complexity to ones manually chosen by prior work. For example, the task switch interface (hand-picked by HyperFuzzer [23]) merely amounts to 0.8% of edge coverage for Xen and 2.9% for KVM. However, we find that, dependent on the HV, interfaces such as nested virtualization, LDTR/TR accesses, RDTSCP, CPUID, and CR accesses account for equal or more coverage when compared to the task switch interface. Prior work argued that task switch is an interesting interface to target, as it was involved in previously discovered bugs [23]. We note that this similarly holds true for interfaces not covered by existing research, e.g., nested virtualization [45], [46], [47], [10], CPUID [48] or CR access [49]. We find that 11.7% for Xen and 18.9% for KVM of our total coverage stems from virtual CPU interfaces that were not targeted by previous work. As such, we find that DSM enables HyperMirage to meaningfully increase coverage on a HVs' virtual CPU implementation. VM-exits that have a low edge coverage either share a lot of code with other VM-exits (e.g., ACCESS LDTR OR TR for KVM), contain little handling code (e.g., MWAIT, WBINVD), or are dependent on HV state (e.g., VMLAUNCH, VMFUNC) and therefore difficult to exert (see Section VII).

The higher coverage achieved by the hybrid configuration over the graybox configuration shows that the integration of symbolic execution effectively improves virtual CPU coverage. Nonetheless, we see that pure graybox fuzzing, facilitated by DSM, itself already effectively explores the virtual CPU implementation of both Xen and KVM. This can largely be attributed to the huge portion of coverage stemming from the HV's x86 memory-accessing instruction emulator. Intuitively, hybrid fuzzing should fare better, as a large variety of different instruction opcodes must be generated by the fuzzer. An analysis of Xen's and KVM's source code reveals that opcode decoding is performed on a byte-by-byte basis, which enables random mutations to quickly progress by guessing individual bytes of an opcode and therefore triggering new coverage. This incrementally leads the graybox approach to exploring large parts of the instruction emulator. Interestingly, such a pattern is a known optimization deployed by previous fuzzing approaches [50], where, for example, 4-byte integer comparisons are split up into four individual byte-wise comparisons.

The difference in edge-count between Xen and KVM directly reflects the complexity of their kernel-level component. For instance, Xen's hypercall interface supports 21 hypercalls, some of which have alternative compatibility implementations for 32-bit guests, whereas KVM merely supports 5 rather simplistic hypercalls.

C. Performance

To gauge performance metrics of HyperMirage, we compare its throughput to HyperPill and discuss its memory footprint.

	HyperMirage		**HyperPill**
	Hybrid	**Graybox**	
Xen	3357	3720	23
KVM	2023	2277	93

TABLE III: Average executions per second by HyperMirage and HyperPill.

1) Throughput: We depict average executions per second for HyperMirage and HyperPill in Table III. We find that HyperMirage drastically outperforms HyperPill with $144\times$ more executions for Xen and $17.69\times$ executions for KVM. This is unsurprising as HyperPill is built upon Bochs, an x86 CPU emulator, while HyperMirage is executing entirely native and built with high throughput in mind. We attribute lower throughput for KVM when compared to Xen to the higher system noise, as Linux is more eager to schedule L1 user space processes. The low drop-off in throughput when going from HyperMirage's graybox to hybrid fuzzing can be explained by the fact that HyperMirage tailors an existing state-of-the-art concolic executor, QSYM, to HVs, which implements various fuzzing optimizations, such as optimistically solving constraints and uninteresting basic block pruning [28].

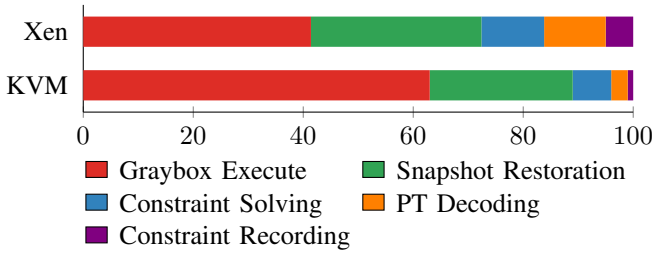


Fig. 6: Percentage of time spent by HyperMirage per component.

To emphasize this, we evaluate where HyperMirage spends most of its time during a long-running fuzzing campaign in Figure 6 and find that the majority of time is spent during

regular graybox execution of the target. Merely $\approx 16\%$ of time for Xen and $\approx 8\%$ for KVM is spent recording and solving constraints. This confirms that the integration of state-of-the-art symbolic executors is worthwhile, as overall coverage improves, yet the throughput of randomized mutations (which are required to discover bugs that are not directly modeled in the source code) is hardly impacted.

2) Memory Consumption: To set up a fuzzing instance, HyperMirage requires as much memory as the target HV and its nested VM require. With symbolic execution support, HyperMirage spawns this setup twice, essentially doubling the memory footprint. The memory consumed by AFL++, as well as the SymCC QSYM backend, is negligible compared to the overhead of spawning two nested VMs. For both Xen and KVM, we managed to set up fuzzing VMs with memory consumption as low as 128MB. Together with a duplicated symbolic executor VM as well as the remaining overhead of the infrastructure, memory usage per hybrid fuzzing instance can be as low as 512MB.

D. Bug Discovery

Ultimately, the goal of a fuzzer is to unveil new bugs in the target software. We depict the bugs discovered during HyperMirage's development and fuzzing campaigns in Table IV. In total, HyperMirage has discovered 11 bugs out of which 4 are confirmed as security critical. All bugs were reported to the respective project maintainers, and CVEs were assigned. We now describe the discovered vulnerabilities and appropriate real-world attack scenarios as well as the components of HyperMirage that led to their discovery.

CVE-2023-46842. The nature of this vulnerability has already been extensively described in Section III-B. In essence, any Xen HVM guest may set up an architecturally undefined, specially prepared GPR state before triggering a long-running hypercall, all the while executing in 32-bit protected mode in order to panic the HV. DSM enables the discovery of this vulnerability, as no artificial constraints are imposed on the architectural validity of inputs. Previous fuzzing methods, relying on manually specified interactions with the HV, would likely miss this architectural edge case.

CVE-2024-45818. Guest memory accesses to MMIO regions are trapped, and the offending instructions are emulated by Xen. If a memory access hits the VGA region (`0xa0000 - 0xc0000`), Xen intentionally acquires a lock until the emulation is finished. However, this lock is acquired on every single memory access. Therefore, any HVM guest can execute an instruction that performs multiple memory accesses (e.g., `REP MOVS`) to the VGA region and force the HV to deadlock. The constraints required to reach this condition greatly favor hybrid fuzzing. Nonetheless, we witnessed the discovery of this bug during a two-week-long graybox-only fuzzing campaign.

CVE-2025-38351. The Hyper-V hypercall emulation of KVM enables guests to perform paravirtualized Translation Lookaside Buffer (TLB) flushes by supplying a list of Guest Virtual Addresses (GVAs), which are in turn flushed by KVM via the `INVVPID` VMX instruction. Missing validation of these GVAs enables malicious guests to send up non-canonical GVAs, resulting in the `INVVPID` instruction failing, and as a result triggering a warning in the KVM host. Such warnings result in host panics in case the `panic_on_warn` kernel parameter is set, which is commonplace for host kernels in cloud environments [51]. The discovery of this vulnerability relies on the joint effort of constraint solving as well as random mutations. The symbolic executor quickly reaches the offending sub-command of the hypercall handler. Then, as the failure of `INVVPID` on non-canonical addresses is not modeled in source code, supplying it with an invalid GVA relies on random graybox mutations.

CVE-2025-38469. As part of the Xen hypercall emulation, KVM provides a hypercall that enables guests to interact with the scheduler. A specific code path in one of the sub-command handlers of this hypercall allocates a dynamic buffer for temporary storage, but does not release it again before returning from the function in a specific error-handling case. Reaching this error-handling case requires a chain of constraints across the VMCS (`exit reason` = 18), registers (`RAX` = 29 && `RDI` = 3) as well as memory values fetched from the GVA passed in via `RSI` (a 4-byte integer with value between 1 and 128 followed by an invalid 8-byte GVA). Since detecting memory leaks requires deploying Linux's memory leak detector `kmemleak`, which drastically decreases fuzzing performance, as a single memory leak scan takes between 100 and 1000 milliseconds (depending on the amount of physical RAM), this bug was only discovered during hybrid fuzzing. The lower throughput stemming from `kmemleak` makes random graybox mutations highly ineffective, whereas HyperMirage's symbolic record-and-replay leads HyperMirage to the offending error-handling case in only 25 minutes.

All above vulnerabilities enable a malicious VM to impact its host HV as long as the VM is able to control its own kernel, which is standard in cloud tenant scenarios. With that, we find that HyperMirage is effective in discovering bugs and vulnerabilities in real-world HVs in a variety of virtual CPU interfaces without requiring tedious manual labor of pre-defining *appropriate* VM states for any interface.

E. False Positives

A potential drawback of DSM is the occurrence of false positives, i.e., fuzzing inputs that crash the target HV but are infeasible to reproduce during regular execution of a VM. As such, we provide an analysis of the crashing inputs discovered by HyperMirage during the 24-hour hybrid fuzzing experiments. For brevity, we restrict this analysis to the Xen experiments, but note that the same analysis on KVM reveals comparable results. For Xen, HyperMirage discovers in total 157 unique crashing inputs. From these, we de-duplicate crashing inputs by their *signal*, e.g., the emulation of different instructions violating the same assert are treated as a singular crash. Among the 18 de-duplicated crashing inputs, we classify whether they represent architecturally valid VM states. In total, we found 13 (i.e., 72%) of the crashing inputs to be false positives. While at first glance this seems like a huge

CVE	Commit	Hypervisor	Description	VM-exit	Component	Signal
CVE-2023-46842	1166467e	Xen	HVM hypercalls may trigger Xen bug check.	VMCALL	Hypercall	Panic
N/A	d980886f	Xen	Out-of-bounds shift in memory exchange hypercall.	VMCALL	Hypercall	UBSan
CVE-2024-45818	c41c3d8c	Xen	Deadlock in HVM standard VGA handling.	APIC Emul.	MMIO	Panic
N/A	672894a1	Xen	MMIO cache emulation failure.	APIC Emul.	MMIO	Assert
N/A	N/A	Xen	MMIO cache emulation failure.[r]	APIC Emul.	MMIO	Assert
N/A	59e6ad65	Xen	Missing cleanup on HVM memory mappings.	EPT Violation	Instruction Emul.	Assert
N/A	a677964c	Xen	Incorrect offset in instruction emulation.	EPT Violation	Instruction Emul.	Assert
N/A	73570ceb	Xen	Incorrect IP rollback in instruction emulation.	EPT Violation	Instruction Emul.	Assert
N/A	a150ecce	Xen	Out-of-bounds shift in FPU emulation.	EPT Violation	Instruction Emul.	UBSan
CVE-2025-38351	fa787ac0	KVM	INVVPID failure during PV TLB flush.	VMCALL	Hypercall	Panic
CVE-2025-38469	5a53249d	KVM	Memory leak in schedop poll hypercall.	VMCALL	Hypercall	Kmemleak

[r] This bug was rediscovered under different circumstances after a fixing commit has been released.

TABLE IV: Bugs discovered by HyperMirage. Entries without CVEs are deemed non-security-critical. The Commit ID represents the patch that resolves the bug.

number, resulting in a lot of manual overhead, 4 out of the 13 false positives were trivially filtered out via textual matching against the panic messages. To be precise, these cases effectively contain panic messages of the form 'feature *FEAT* not enabled'. Further, the remaining 9 false positives can be grouped into the following three categories.

Segmentation Asserts (7 false positives). For interaction with segment descriptors under Intel VMX, the CPU caches the values of active segment descriptors in appropriate fields in the `VMCS[GS]` area. Whenever Xen interacts with these cached values, it performs a variety of architectural assertions on their attribute and base values (e.g., that the base holds a canonical address). As DSM enables direct mutation of these fields, the aforementioned asserts will inadvertently be violated. We manually verify these asserts and conclude that they are not reachable under regular execution, as either the CPU similarly rejects non-conforming values, or they are already caught by Xen's emulator for segment-loading instructions. Manual analysis required ≈ 1 hour of manual labor, after which we are able to categorically classify these asserts as false positives.

Fatal Page Fault (1 false positive). During the handling of the `VM-entry failure due to Model Specific Register (MSR) loading` VM-exit, Xen accesses its MSR loading area in order to print diagnostic information. In regular execution, this VM-exit only occurs if previously set up by the HV, at which point Xen would also allocate the MSR loading area. Without this allocation, accessing the MSR loading area results in a fatal page fault, trivially identifiable as a false positive. Triaging until the root cause took ≈ 10 minutes of manual labor.

General protection fault in emulation of `fxsave` / `fxrstor` (1 false positive). To emulate the `fxsave` and `fxrstor` instructions, Xen eventually executes these instructions natively in Xen's context. For this, the operand's memory address (resolved according to the appropriate segment descriptor) is first mapped into HV memory. A specially crafted input can force Xen to execute this instruction on an unaligned block of memory, triggering a general protection fault. During an attempt to reproduce this behavior outside of the fuzzer, we realized that Xen indeed performs an alignment check on the memory operand. However, the alignment check and the actual memory mapping utilize different methodologies to identify the VM's current execution mode. The alignment check assumes the VM is executing in long mode by bits set in CR0 as well as the EFER MSR, thereby not expanding the memory address according to the segment descriptor, as they are ignored in long mode. Later, when the address is mapped, Xen checks for the VM flag of the RFLAGS register and realizes it is executing in virtual 8086 mode, where the segment descriptor base and limit apply. This inconsistency results in Xen possibly mapping an unaligned address. Understanding this inconsistency required ≈ 2 hours of manual labor.

In the end, we find that only a small number of crashing inputs require manual analysis to distinguish them from true positives. We believe that the ratio compared to discovered real-world bugs and vulnerabilities is justifiable. We also highlight that while the analysis of the `fxsave`-related false positives did not reveal an actual bug, it nonetheless exposed an inconsistency in Xen's decision code on the VM's current execution mode. We believe these types of inconsistencies are still worth patching in order to avoid related bugs in the future.

VII. Discussion

In this section, we briefly highlight the current limitations of our prototype HyperMirage, and provide directions for future work we deem promising.

Improved Error Detection. HyperMirage detects errors by hooking error-handling mechanisms implemented by the HV itself, i.e., `panic` or `assert`, which catch CPU exceptions as well as developer-introduced sanity-checks. However, more subtle bugs, such as out-of-bounds memory accesses that do not directly trigger CPU exceptions, are currently not caught by Xen, as only KVM implements KASAN, an Address Sanitizer [52] for kernel space. Conversely, we find that primarily Xen introduces developer sanity-checks e.g., for certain incorrect emulation results, which is hardly the case for KVM. As such, it is desirable to obtain a generic ASAN implementation that is applicable to a wide range of HVs, as well as a generic approach to detecting bugs that result in incorrect emulation of a HV. While prior work described

approaches to tackle these issues, they are either not available for HVs (e.g., BoKASAN [53]) or not suitable for high-performance fuzzing [24], [27].

Automated Reproducer Generation. Currently, all crashing inputs produced by HyperMirage are manually analyzed, and relevant values (e.g., GPRs, memory, execution mode, etc.) have to be transplanted into a stand-alone reproduction environment. To further ease an analyst's job, the majority of this process should be automated by generating appropriate reproducers from the VM state description of the crashing input. This would reduce reproduction efforts to the manual translation of complex fields of the `VMCS[GS]` and `VMCS[VEI]`, whose semantics are difficult to infer automatically.

VM-Exit Dependencies. While we and prior work observe that the majority of VM-exit handling code is entirely dependent on the current state of the VM [23], there remains a small portion of HV-state dependent code. For instance, the nested virtualization interface is likely not fully explored by HyperMirage, as it contains HV-internal state management, which is influenced by individual VM-exits. For instance, exerting the `VMPTRLD` VM-exit is only meaningful when previously `VMXON` has successfully been executed. However, our current fuzzing infrastructure is, in general, not well-suited for this HV-internal state management as the entire HV-state is reset after handling an individual VM-exit. We leave modeling these inter-VM-exit dependencies for future work.

Porting to other Architectures. HyperMirage currently targets the virtual CPU implementation provided by Intel VMX, as it benefits from Nyx's integration with Intel's *Processor Tracing* feature for speeding up coverage tracing, which is not present on AMD or ARM CPUs. However, HyperMirage is compatible with other solutions, such as the instrumentation-based coverage tracing infrastructure provided by AFL++.

Since both AMD and Intel CPUs are x86-based, AMD SVM is conceptually very similar to Intel VMX. For example, while upon VM-exit VMX stores the VM's state in the *Guest-State Area* of the *VMCS*, AMD saves it in the *Save State Area* of the *Virtual Machine Control Block (VMCB)*. Additionally, instead of Intel EPTs, AMD implements second-level address translation as NPTs, which have a similar format to EPTs. Moreover, while GPRs must be HV-saved on VMX, the SVM hardware automatically stores them in the *Save State Area* of the VMCB. As such, adoption to AMD SVM requires minor changes in DSM and relevant parts in our symbolic record-and-replay approach to respect the layout of the VMCB.

While VMX and SVM serialize the guest's state automatically in the VMCS and VMCB, ARM VE does not employ a similar approach. In contrast, aiming to improve performance on HV-VM context switching, VE leaves it up to each HV implementation to fetch the VM state components that it needs to handle a VM-exit. Therefore, testing HVs that implement ARM VE via HyperMirage would require analyzing the target HV and identifying the state needed during each VM-exit. We leave this open for future work.

VIII. Related Work

We now discuss related work on fuzzing HV interfaces in commodity HVs (including Xen and KVM) as well as applications of hybrid fuzzing techniques.

Hybrid Fuzzing. Hybrid fuzzing has been mainly studied for improving bug discovery in user space [54], [55], [28], [56], [29], [30]. QSYM relies on *dynamic binary translation (DBT)* at the instruction level to transform machine instructions to IR and perform symbolic computations via an optimized symbolic backend. SymCC [29] avoids expensive DBT by instrumenting target programs at compile time and directly calling a symbolic execution backend linked into the target program, demonstrating a significant increase in performance over QSYM. SymQEMU [30] builds upon SymCC by performing its instrumentation on the IR of QEMU's Tiny Code Generator (TCG), making the SymCC approach available for binary-only targets. HFL [32] introduced hybrid fuzzing to OS kernels (e.g., Linux) by leveraging S2E [57], which relies on a custom DBT module in QEMU to generate symbolic instructions and on KLEE for delegating the computation of symbolic values. HyperFuzzer [23] was the first to leverage hybrid fuzzing to find bugs in HVs by employing a custom (closed-source) symbolic execution engine based on recording and decoding pure control-flow traces. As it provides the best performance, HyperMirage extends SymCC with the symbolic record-and-replay technique, making it available for fuzzing bare-metal targets with high throughput.

Testing CPU Emulation. 'Fuzzing the Xen Hypervisor' [58] deploys a black-box hypercall fuzzer, which is, however, limited by its restriction to a single virtual CPU interface as well as black-box mutations without feedback guidance. 'Virtual CPU Validation' [24] validates CPU emulation implementations by leveraging tests written for physical CPUs. PokeEmu [25] and FastPokeEmu [26] reveal issues in HVs by performing symbolic execution on a high-fidelity emulator (e.g., Bochs) to generate test cases for low-fidelity emulators (e.g., QEMU TCG) as well as native execution environments (e.g., QEMU+KVM). MultiNyx [27] performs multi-level symbolic execution on the HV under test and a reference CPU emulator, aiming to generate test cases that systematically analyze the implementation of the HV. While effective in generating complex test cases, they suffer from low testing throughput. HyperFuzzer [23] addresses these limitations by integrating fuzzing and symbolic execution into a hybrid fuzzer that tests Hyper-V's virtual CPU implementation; however, it relies on manually generated VM states used as fuzzing seeds to target specific VM-exits, which limits it to only testing 5 VM-exits. IRIS [59] proposed automatically inferring valid VM states to fuzz Xen's virtual CPU implementation by dynamically profiling the VM-exits generated by the Linux kernel under a suite of workloads. However, this only generates regular architectural VM states, which prevents IRIS from discovering bugs. Instead, HyperMirage generates VM states that fuzz CPU virtualization automatically by analyzing the HV codebase, and comprehensively by adopting the novel DSM

technique, which is able to explore the VM-exit interface in depth and reveal bugs that can only be triggered by unusual, yet valid, architectural states.

Fuzzing I/O Emulation. Hyper-Cube [21] is a black-box HV fuzzer that relies on random mutations on custom bytecode, whose opcodes represent interactions with the MMIO and port I/O interfaces of HVs. NYX [22] is a fast snapshot-based graybox fuzzer that enhances this concept by allowing grammar specifications for the bytecode interpreter. ViDezzo [15] and Truman [19] focus on better inferring the interactions between virtual devices and device drivers in order to perform smarter input mutations. As HyperMirage focuses on fuzzing the CPU emulation interfaces of HVs, where different challenges apply, we do not believe it is effective in scrutinizing virtual devices implemented in user space. Moreover, HyperMirage's novelty lies in applying state-of-the-art concolic execution to kernel space, and not user space, where concolic executors are readily available. HyperPill [14] is capable of fuzzing the MSR, MMIO, port I/O, and hypercall interfaces of binary-only HVs in an emulated environment. Similar to HyperMirage, HyperPill directly manipulates VMCS fields to improve fuzzing throughput, but overall throughput remains low because of emulation. While HyperPill leverages only coverage-guided fuzzing on a subset of available VMCS fields, HyperMirage explores the whole VMCS in depth via DSM and hybrid fuzzing, managing to cover the majority of VM-exits implemented by the HV.

IX. Conclusion

HyperMirage is an efficient hybrid fuzzer for virtual CPUs that achieves coverage in a wide variety of virtual CPU interfaces, without requiring expert knowledge in crafting fuzzing seeds. For this, we introduced Direct State Manipulation, a novel fuzzing methodology to directly exert a HV's *view* on its VMs without requiring their execution. Moreover, we are the first to make the state-of-the-art symbolic executor, SymCC, available for bare-metal software. Our evaluation shows that HyperMirage achieves drastically higher coverage over HyperPill. HyperMirage identifies 9 new bugs in Xen and 2 in KVM, all of which have been confirmed by the respective project maintainers.

Availability

To encourage open research, we make HyperMirage available at https://github.com/tum-itsec/hypermirage.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. This work was funded in part by the Bavarian Ministry of Science and Arts (STMWK), under the project "Security in everyday use of digital technologies (ForDaySec)". Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of STMWK.

References

[1] Intel Corporation. (2025) Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. Accessed: 2025-04-15.

[2] AMD. (2024) AMD64 Architecture Programmer's Manual Volume 2: System Programming. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf. Accessed: 2025-04-15.

[3] Arm Limited. (2024) Arm® architecture reference manual for a-profile architecture. https://developer.arm.com/documentation/ddi0487/latest. Accessed: 2025-04-24.

[4] MITRE. CVE-2023-4155. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-4155. Accessed: 2025-04-23.

[5] ——. CVE-2024-43521. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-43521. Accessed: 2025-04-23.

[6] ——. CVE-2017-15590. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-15590. Accessed: 2025-04-23.

[7] ——. CVE-2021-29657. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-29657. Accessed: 2025-04-23.

[8] ——. CVE-2024-38080. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-38080. Accessed: 2025-04-23.

[9] ——. CVE-2017-1000407. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1000407. Accessed: 2025-04-23.

[10] ——. CVE-2020-2732. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-2732. Accessed: 2025-04-23.

[11] ——. CVE-2021-4093. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-4093. Accessed: 2025-04-23.

[12] ——. CVE-2024-55881. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-55881. Accessed: 2025-04-23.

[13] ——. CVE-2025-21779. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2025-21779. Accessed: 2025-04-23.

[14] A. Bulekov, Q. Liu, M. Egele, and M. Payer, "Hyperpill: Fuzzing for hypervisor-bugs by leveraging the hardware virtualization interface," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[15] Q. Liu, F. Toffalini, Y. Zhou, and M. Payer, "ViDeZZo: Dependency-aware Virtual Device Fuzzing," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[16] G. Pan, X. Lin, X. Zhang, Y. Jia, S. Ji, C. Wu, X. Ying, J. Wang, and Y. Wu, "V-Shuttle: Scalable and Semantics-Aware Hypervisor Virtual Device Fuzzing," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[17] A. Bulekov, B. Das, S. Hajnoczi, and M. Egele, "Morphuzz: Bending (input) space to fuzz virtual devices," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[18] C. Myung, G. Lee, and B. Lee, "MundoFuzz: Hypervisor fuzzing with statistical coverage testing and grammar inference," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[19] Z. Ma, Q. Liu, Z. Li, T. Yin, W. Tan, C. Zhang, and M. Payer, "Truman: Constructing device behavior models from os drivers to fuzz virtual devices," in *Network and Distributed System Security Symposium (NDSS)*, 2025.

[20] Y. Chen, S. Zhang, X. Jia, Q. Zhou, H. Huang, S. Xu, and H. Du, "SEDSpec: Securing Emulated Devices by Enforcing Execution Specification," in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2024.

[21] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "Hyper-cube: High-dimensional hypervisor fuzzing." in *Network and Distributed System Security Symposium (NDSS)*, 2020.

[22] ——, "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[23] X. Ge, B. Niu, R. Brotzman, Y. Chen, H. Han, P. Godefroid, and W. Cui, "HyperFuzzer: An Efficient Hybrid Fuzzer for Virtual CPUs," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[24] N. Amit, D. Tsafrir, A. Schuster, A. Ayoub, and E. Shlomo, "Virtual CPU validation," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[25] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: Hi-fi tests for lo-fi emulators," *ACM SIGARCH Computer Architecture News*, 2012.

[26] Q. Yan and S. McCamant, "Fast PokeEMU: Scaling Generated Instruction Tests Using Aggregation and State Chaining," in *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2018.
[27] P. Fonseca, X. Wang, and A. Krishnamurthy, "MultiNyx: a multi-level abstraction framework for systematic analysis of hypervisors," in *Proceedings of the Thirteenth EuroSys Conference*, 2018.
[28] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.
[29] S. Poeplau and A. Francillon, "Symbolic execution with SymCC: Don't interpret, compile!" in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.
[30] ——, "SymQEMU: Compilation-based symbolic execution for binaries," in *Network and Distributed System Security Symposium (NDSS)*, 2021.
[31] C. Cadar and M. Nowack, "KLEE symbolic execution engine in 2019," *International Journal on Software Tools for Technology Transfer*, 2021.
[32] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "HFL: Hybrid Fuzzing on the Linux Kernel." in *Network and Distributed System Security Symposium (NDSS)*, 2020.
[33] K. Kim, T. Kim, E. Warraich, B. Lee, K. R. Butler, A. Bianchi, and D. J. Tian, "Fuzzusb: Hybrid stateful fuzzing of usb gadget stacks," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022.
[34] The Xen Project, "The Xen Hypervisor Project," https://xenproject.org, Accessed: 2025-04-17.
[35] Linux Kernel Community, "Kernel Virtual Machine," https://linux-kvm.org, Accessed: 2025-07-28.
[36] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL:Hardware-Assisted feedback fuzzing for OS kernels," in *26th USENIX security symposium (USENIX Security 17)*, 2017.
[37] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.
[38] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.
[39] ——. (2025) SymQEMU custom mutator - AFL++. https://github.com/AFLplusplus/AFLplusplus/tree/stable/custom_mutators/symqemu. Accessed: 2025-04-13.
[40] ——. (2025) SymCC custom mutator - AFL++. https://github.com/AFLplusplus/AFLplusplus/tree/stable/custom_mutators/symcc. Accessed: 2025-04-13.
[41] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.
[42] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, 2008.
[43] Y. Hao, J. Pu, X. Li, Z. Qian, and A. A. Sani, "Syzspec: Specification generation for linux kernel fuzzing via under-constrained symbolic execution," in *Proceedings of the 2025 on ACM SIGSAC Conference on Computer and Communications Security*, 2025.
[44] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018.
[45] MITRE. CVE-2013-4551. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-4551. Accessed: 2025-07-30.
[46] ——. CVE-2018-16882. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-16882. Accessed: 2025-07-30.
[47] ——. CVE-2019-7221. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-7221. Accessed: 2025-07-30.
[48] ——. CVE-2011-1936. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1936. Accessed: 2025-07-30.
[49] ——. CVE-2014-3690. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3690. Accessed: 2025-07-30.
[50] lafintel. (2016) Circumventing fuzzing roadblocks with compiler transformations. https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/. Accessed: 2025-04-17.
[51] Jonathan Corbet. Warning about WARN_ON(). https://lwn.net/Articles/969923/. Accessed: 2025-07-21.
[52] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *2012 USENIX annual technical conference (USENIX ATC 12)*, 2012.
[53] M. Cho, D. An, H. Jin, and T. Kwon, "BoKASAN: Binary-only kernel address sanitizer for effective kernel fuzzing," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.
[54] R. Majumdar and K. Sen, "Hybrid concolic testing," in *29th International Conference on Software Engineering (ICSE'07)*, 2007.
[55] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *Network and Distributed System Security Symposium (NDSS)*, 2016.
[56] L. Zhao, Y. Duan, and J. Xuan, "Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing," in *Network and Distributed System Security Symposium (NDSS)*, 2019.
[57] V. Chipounov, V. Kuznetsov, and G. Candea, "The s2e platform: Design, implementation, and applications," *ACM Transactions on Computer Systems (TOCS)*, 2012.
[58] S. Bleikertz. Fuzzing the Xen Hypervisor. https://openfoo.org/blog/xen-fuzz.html. Accessed: 2025-07-29.
[59] C. Cesarano, M. Cinque, D. Cotroneo, L. De Simone, and G. Farina, "IRIS: a Record and Replay Framework to Enable Hardware-assisted Virtualization Fuzzing," *arXiv preprint arXiv:2303.12817*, 2023.

APPENDIX

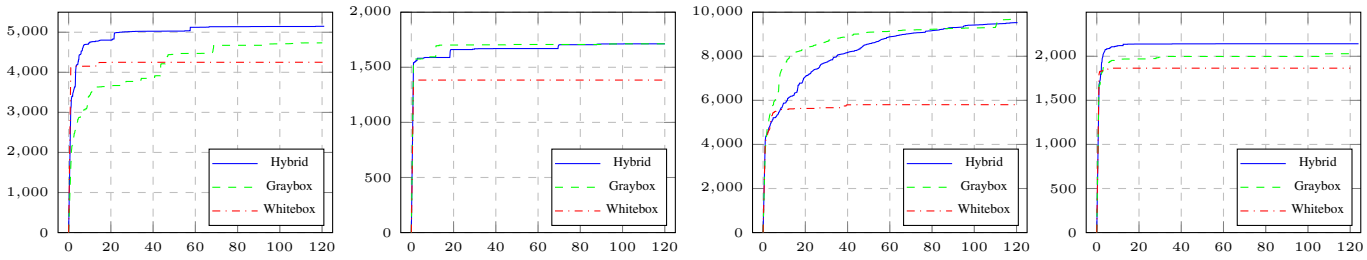
To perform an in-depth performance evaluation of HyperMirage, we turn towards the state-of-the-art hybrid fuzzer for virtual CPUs: HyperFuzzer [23]. Unfortunately, no completely fair evaluation is possible, as HyperFuzzer's experiments were performed on Hyper-V, for which source code is unavailable, and HyperFuzzer itself is closed source, meaning that we cannot port it to Xen or KVM. Nonetheless, HyperFuzzer represents the research project that is most similar to HyperMirage as its focus is the virtual CPU implementation of HVs, and it also deploys a hybrid fuzzing methodology. Therefore, we believe a comparative evaluation is of utmost importance, and we aim to provide as fair an evaluation as possible by minimizing the amount of differentiating factors.

A. Experiment Setup

We reproduce the methodology of HyperFuzzer's performance evaluation on the example of Xen. As HyperFuzzer evaluated their performance metrics separately for the four interfaces *Hypercalls*, *Task Switch*, *APIC emulation*, and *MSR emulation*, we artificially restrict HyperMirage to exactly these interfaces as well by exempting the `exit reason` field of `VMCS[VEI]` from any mutations. Then, we execute HyperMirage for 120 minutes on each interface to generate a corpus of interesting inputs dubbed the "expanded fuzzing" set by the HyperFuzzer authors. The obtained per-interface coverage is depicted in Figure 7. Note that, for completeness's sake, we also include the coverage achieved by the graybox and whitebox-only experiments performed in Appendix C

To rule out hardware performance differences, the experiment is carried out on an equivalent CPU[3] (see Section VI-A) as the one utilized by the HyperFuzzer authors.

[3]HyperFuzzer utilized a K-line CPU, which, if not overclocked, is identical to our model.



(a) Hypercall (b) Task Switch (c) APIC emulation (d) MSR Emulation

Fig. 7: Discovered edges of HyperMirage per interface during the 120-minute fuzzing campaign. The y-axis depicts discovered edges. The x-axis depicts the time in minutes.

B. Efficiency

We evaluate the efficiency of HyperMirage by (1) testing all inputs in the expanded fuzzing set for new coverage (i.e., without performing any mutations) and (2) running them through our symbolic record-and-replay component to solve constraints and measure raw run time taken per execution. In a single symbolic execution run, we include both the recording and replay phases (see Section IV-D) in the execution time measurement.

	HyperMirage		**HyperFuzzer**	
	Testing	**Symbolic**	**Testing**	**Symbolic**
Hypercalls	**0.21**	**320.83**	0.48	781.89
Task Switch	**0.28**	**40.86**	0.33	457.83
APIC Emu.	**0.21**	438.83	0.36	**212.66**
MSR Emu.	**0.17**	**226.86**	0.36	387.51
Average	**0.21**	**256.84**	0.38	459.97

TABLE V: Efficiency comparison between HyperMirage and HyperFuzzer per interface during the 120-minute fuzzing campaign. Numbers represent average execution time in milliseconds.

The mean runtime of a single execution per interface is depicted in Table V. In all cases, excluding one, HyperMirage outperforms HyperFuzzer in single-execution efficiency for both graybox testing as well as symbolic analysis. Graybox testing sees a speed-up of $1.7\times$ averaged across all interfaces, as HyperMirage merely restores the taken snapshot and resumes code execution at the HVs VM-exit handler, opposed to HyperFuzzer, which spawns and initializes a new VM for every tested input. For symbolic execution, HyperMirage achieves up to $2.4\times$ performance improvement for hypercalls and an up to $11\times$ increase for simpler interfaces such as task switching.

A notable outlier is the symbolic execution speed for the APIC emulation interface, where HyperMirage incurs a 2x *slowdown*. This is likely due to HyperMirage's significantly higher edge count on Xen (up to 10000, see Figure 7) when compared to HyperFuzzer's on Hyper-V (up to 4000[4] [23]). For this interface, Xen deploys a sophisticated x86 instruction emulator, which will result in a complex chain of constraints for HyperMirage to solve. While Hyper-V will deploy a similar emulator, the coverage numbers indicate a much lower complexity. As such, to even out complexity discrepancies between implementations as measured by edge count, a more realistic comparison would be against HyperFuzzer's efficiency numbers for the hypercall interface, which achieved coverage of approximately 8000 edges (and therefore, edge discovery is still higher for HyperMirage). With this comparison, HyperMirage depicts an improvement over HyperFuzzer by $\approx 1.8\times$ for complex interfaces.

We find that HyperMirage also drastically outperforms HyperFuzzer in symbolic execution efficiency. Where HyperFuzzer requires parsing and disassembling of a full execution trace of the HV, HyperMirage simply records symbolic operations during execution via optimized compiler-based techniques, and forwards constraints during a lightweight decoding phase to its solving backend.

[4]Edge coverage shot up to just below 4500 towards the end, however this is unlikely to meaningfully affect average execution time.

C. Throughput

To evaluate the throughput of HyperMirage, we calculate the average number of executions per second in three different setups: hybrid fuzzing, graybox-only fuzzing, and whitebox-only fuzzing. For whitebox-only fuzzing, we disable an optimization in the custom mutator plugin (see Section V-B), namely that every interesting input is only symbolically executed once, as otherwise, at some point, no further mutations would be performed. The results, as well as comparable numbers from HyperFuzzer (targeting Hyper-V), are displayed in Table VI. The results show that HyperMirage outperforms HyperFuzzer in hybrid fuzzing speeds on every single interface, with an average speed-up of factor $2.5\times$. Interestingly, this remains true even for the APIC emulation interface, where raw efficiency numbers show that the symbolic execution component on the expanded fuzzing set has a longer runtime than the equivalent for HyperFuzzer. This can be attributed to HyperMirage's integration of QSYM [28], which hosts a plethora of optimization strategies specially tailored to fuzzing. For the task switch interface, hybrid and graybox fuzzing throughput is effectively equal. This is to be expected when considering the raw symbolic execution run time, as seen in

	HyperMirage			HyperFuzzer		
	Hybrid	**Graybox**	**Whitebox**	**Hybrid**	**Graybox**	**Whitebox**
Hypercalls	**3721.20**	**4375.23**	**115.13**	979.40	2945.03	35.63
Task Switch	**4442.17**	**4422.11**	**159.32**	1369.95	4378.22	106.88
APIC Emu.	**2422.04**	**4352.77**	20.53	1774.05	3650.68	**210.85**
MSR Emu.	**2658.75**	**5024.86**	**129.83**	1171.24	3967.35	76.67
Average	**3311.04**	**4543.74**	106.20	1323.65	3735.32	**107.50**

TABLE VI: Throughput comparison between HyperMirage and HyperFuzzer per interface during the 120-minute fuzzing campaign. Numbers represent average executions per second.

Table V. In essence, this interface is too simplistic for the symbolic execution component to spend much time solving constraints.