MultiNyx: A Multi-Level Abstraction Framework for Systematic Analysis of Hypervisors

Pedro Fonseca
University of Washington
pfonseca@cs.washington.edu

Xi Wang
University of Washington
xi@cs.washington.edu

Arvind Krishnamurthy
University of Washington
arvind@cs.washington.edu

ABSTRACT

MultiNyx is a new framework designed to systematically analyze modern virtual machine monitors (VMMs), which rely on complex processor extensions to enhance their efficiency. To achieve better scalability, MultiNyx introduces selective, multi-level symbolic execution: it analyzes most instructions at a high semantic level, and leverages an executable specification (e.g., the Bochs CPU emulator) to analyze complex instructions at a low semantic level. MultiNyx seamlessly transitions between these different semantic levels of analysis by converting their state.

Our experiments demonstrate that MultiNyx is practical and effective at analyzing VMMs. By applying MultiNyx to KVM, we automatically generated 206,628 test cases. We found that many of these test cases revealed inconsistent results that could have security implications. In particular, 98 test cases revealed different results across KVM configurations running on the Intel architecture, and 641 produced different results across architectures (Intel and AMD). We reported some of these inconsistencies to the KVM developers, one of which already has been patched.

ACM Reference Format:
Pedro Fonseca, Xi Wang, and Arvind Krishnamurthy. 2018. MultiNyx: A Multi-Level Abstraction Framework for Systematic Analysis of Hypervisors . In *EuroSys '18: Thirteenth EuroSys Conference 2018, April 23–26, 2018, Porto, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3190508.3190529

1 INTRODUCTION

As the popularity and reliance on virtualization increased, both Intel and AMD introduced the VMX [26] and SVM [2] x86 ISA extensions[1], respectively, which are now nearly ubiquitous in recent processor models. These extensions dramatically reduce the performance overheads of virtualization, explaining their fast adoption.

The complexity of virtualization extensions poses additional challenges to VMM developers for several reasons [5, 11, 21]. First, extension instructions have particularly complex semantics, and their documentation can be ambiguous and often is difficult to understand [5, 36]. Second, the virtualization extensions change the semantics of many (possibly all) other instructions by introducing additional execution modes (e.g., VMX root and non-root modes), which affects even basic mechanisms such as memory accesses and control flow. Finally, each extension has several variants; for instance, only a subset of Intel CPUs with VMX support advanced virtualization features, such as "unrestricted guest" mode, EPT tables, and shadowing VMCS. This requires VMM developers to consider many different combinations of available features and to implement fallback mechanisms if a given feature is not supported by the host CPU. In practice, these extensions introduce many complex changes to an already complex architecture.

VMMs must provide essential security guarantees, such as isolation between virtual machines; failing to do so can compromise user security, specially in a multi-tenant data center. Unfortunately, due to the complexity of virtualization extensions and the inherent challenges in virtualizing a complex architecture, significant numbers of bugs are regularly found in VMMs, many of which have critical security implications. For instance, based on our survey of the CVE repository [20], 17 security advisories were issued between 2015 and 2017 alone regarding KVM, a mature and widely used VMM. Despite the prevalence of virtualization and its present-day importance, the testing of modern virtualization systems has received insufficient attention.

Today's main approach to testing VMMs relies on manually written test cases, which is burdensome for developers and generally of limited scope [16]. As an alternative, several fuzzing techniques have been proposed [33, 34]. Although both types of approaches are useful, having more systematic analysis techniques is particularly important for testing hypervisors because of the myriad of corner cases, arising from both the intricate hardware specification and the implementation complexity itself.

We propose a framework, MultiNyx, that lets VMM developers systematically test and analyze virtual machine monitors. The key contribution of MultiNyx is a novel symbolic execution approach that models automatically the semantics of complex instructions. MultiNyx achieves this by selectively using an executable specification to augment its knowledge of the x86 semantics. While other work has focused on testing virtual devices, this work focuses on the components that virtualize the CPU and memory – perhaps the most critical and complex components to virtualize in architectures such as x86.

We demonstrate how our approach can be used in a scalable manner to automatically infer the specification of *virtualization extensions* and thus create test cases that explore corner cases. This multi-level testing approach, combined with other techniques that MultiNyx implements, allows it to leverage generic, off-the-shelf, symbolic execution engines typically meant for user-level code. MultiNyx relies on such symbolic engines to analyze instead

[1]VMX and SVM are also known by their commercial names, VT-x and AMD-V, respectively.

EuroSys '18, April 23–26, 2018, Porto, Portugal

ACM ISBN 978-1-4503-5584-1/18/04.
https://doi.org/10.1145/3190508.3190529

system-level code that has significantly more complex semantics. Unlike other techniques, the MultiNyx approach systematically creates test cases that consider both the *specification* of the hardware extensions and the actual *implementation* of a virtual machine monitor.

Using MultiNyx to test KVM, we found that many of the 206,628 generated test cases exposed inconsistencies in the KVM output, when executed across different execution environments – 98 test cases revealed different results across KVM configurations on Intel and 641 produced different results across architectures. Several test cases showed defects in input validation, segmentation virtualization, and protection mechanisms that are serious and can impact the user security. We reported some of the inconsistencies to the KVM developers, who have already patched one of the bugs [29].

In our current implementation, MultiNyx relies on Bochs to approximate the actual hardware specification. Our experience shows that Bochs is sufficiently accurate to ensure that testing is effective. Nevertheless, MultiNyx is modular and can easily accept other executable specifications since it implements a generic instrumentation framework that required only 30 lines of Bochs-specific source code. In §8, we discuss different strategies to further improve the effectiveness of MultiNyx that depend on the availability of even more accurate specifications.

This work makes the following contributions:

- Multi-level analysis: a technique that leverages executable specifications to (1) automatically model the semantics of complex processor instructions and (2) apply generic symbolic execution engines to system-level code
- A methodology to create scaled down VMM unit tests that are particularly amenable to systematic analysis
- An algorithm to generate test cases that explore and analyze corner cases of VMM implementations
- The design of MultiNyx, a framework for analyzing and testing hardware-accelerated VMMs

2 MOTIVATION

The KVM hypervisor is widely used by cloud providers [14, 25, 35, 41]. It virtualizes the CPU and MMU using VMX and SVM extensions, and works in tandem with applications, such as QEMU, that drive it and virtualize devices (e.g., disks and network cards). KVM consists of three kernel modules: `kvm.ko`, `kvm_intel.ko` and `kvm_amd.ko`, with approximately 60 KLOCs. The first module contains architecture-independent functions (e.g., communication with applications through `ioctl`); the other two contain architecture-specific code that interacts with the processor extensions. While this paper focuses on KVM and VMX, the MultiNyx design is applicable to similar VMMs and processor extensions.

This section discusses the challenge to build correct VMMs by presenting concrete examples of virtualization bugs and discusses related work.

2.1 Virtualization Bugs

`CVE-2017-2583` [19] describes a security bug previously found in KVM and eventually fixed in Linux version 4.9.5 that had serious implications for users. Triggered by a seemingly inoffensive `MOV` instruction, this bug could either crash the VM or cause a VM privilege escalation (i.e., an application could escalate from ring 3 to ring 0). Interestingly, the bug impaired both the Intel and AMD architectures but in different ways: KVM could no longer reenter the VM on Intel, while it could reenter the VM with wrong register values (namely the `CPL` priority level) on AMD.

To trigger this bug, a series of conditions need to be satisfied. First, the `MOV` instruction has to be emulated by KVM (as opposed to executed directly by the CPU), which happens only in certain circumstances (§3). Second, the `MOV` instruction specifically has to attempt to load the NULL segment into the stack segment register. Third, the CPU has to execute the instruction in long mode and in ring 3 (`CS.CPL=3`). Fourth, our tests found that other privilege-related registers and segment descriptor fields have to be set to specific values (`SS.RPL=3` and `SS.DPL=3`). The striking aspect about this bug, which also applies to many other VMM bugs, is the complexity of the conditions that simultaneously need to be satisfied to trigger it. As a result, given the vast input space, such bugs are unlikely to be discovered by random testing techniques.

This bug was caused by KVM developers misunderstanding the conditions under which a *VM entry* (§3) would fail. VM entries can fail because processor extensions conduct an extensive series of checks on the VM state when the VMM tries to switch to VM code. As such, the processor semantics caused an execution path unknown to the KVM developers. In the Intel case, the VM entry checks are described throughout a dense chapter of 26-pages in the architecture manual. Furthermore, according to KVM developers, the Intel documentation at that time described these checks incorrectly [17]. However, even had the documentation been correct, it is easy to understand how developers could have misunderstood it or simply failed to reason correctly about such conditions given the complex checks.

Another example of a serious bug previously found in KVM was caused by the incorrect handling of a specially crafted undefined instruction (`CVE-2016-8630` [18]). The problem was caused by incorrect logic in the KVM instruction decoding mechanism that *emulates* the subset of instructions that the processor cannot virtualize: a defect in the `x86_decode_insn()` function let code running in the VM trigger an illegal access by KVM, crashing the VMM. To trigger this bug, the illegal instruction had to additionally contain a specific ModR/M byte value, and, importantly, KVM had to be configured to use shadow paging (i.e., EPT disabled).

These and other examples demonstrate the complexity of the execution paths of modern VMMs. In fact, while trying to understand a recent KVM patch that implemented a new feature in two execution paths, we asked the developer who *wrote the patch* about the conditions under which one of two paths was executed; surprisingly the developer himself did not know how or even whether that code path was executable: *"I don't know if any of [the emulation functions] can call em_cpuid in practice, but since the code was already there it was easy to add the* `CPUID` *faulting logic there as well"* [4].

The root cause of this challenge is that inspection of the KVM source code is not by itself sufficient to understand the execution paths. The complex semantics (and implications) of the hardware extensions must be fully understood to reason about the paths of the VMM implementation. Unfortunately, the extension semantics,

described in over 200 pages of dense and possibly incomplete documentation, are far from accessible even to experienced developers.

2.2 Related Work

We now describe related research on VMM testing, generic symbolic execution techniques, and VMM verification.

VMM Testing. Manual tests [16], albeit laborious, are in practice the standard approach to test VMMs. Thus, developers commonly overlook important test cases that would otherwise reveal critical bugs [3, 18, 19, 33, 34].

Martignoni et al. proposed automated techniques that rely on fuzzing [33, 34] and differential testing, an approach that still requires some developer effort to generate domain-specific templates. These encouraging proposals demonstrate that fuzzing can find bugs in VMMs. However, they are non-systematic because fuzzing fundamentally relies on randomly generated test cases. Given the massive input space for tests in this domain – which includes hundreds of general, control and model-specific registers as well as the RAM state – and the complex inter-relationships between different parts of the input, it is unlikely that non-systematic techniques would be "lucky" enough to generate tests that trigger hard-to-find corner cases.

In general, systematic testing can be achieved using symbolic execution techniques. These techniques enable systematic testing through formal analysis of program constraints and generation of test cases based on these constraints. The applications and benefits of symbolic execution have been extensively demonstrated both by academics [9, 13, 28] and industry [23]; however, there is limited work on applying symbolic execution to test virtualization infrastructures.

PokeEMU [32] is one exception. It applies symbolic execution to an executable specification, *systematically* generating CPU test cases. Thus, PokeEMU generates tests for the different instructions of the CPU, and runs (lifts) those tests cases on a VMM (i.e., on a virtual CPU). Given our problem domain, this approach has two important limitations. First, their testing target is a JIT-based VMM, not a hardware-accelerated VMM, and having generated only 32-bit tests (guests), it even precludes testing emulated virtualization instructions. Second, this approach generates test cases by exclusively analyzing the specification, without analyzing the implementation under test (i.e., black-box testing). As a consequence, bugs that arise from corner-cases of the VMM *implementation* will not be uncovered by this approach. In contrast, MultiNyx generates test cases that explore corner-cases that arise from the *complex interactions* between the VMM implementation and the virtualization extensions (i.e., white-box testing).

To control and analyze the machine state, PokeEMU authors also ran their concrete tests on KVM (with VMX). However, ironically, KVM served as an oracle to identify the *ground truth* for their tests because this environment was considered to be "the closest approximation of the real hardware" [32]. In contrast, the testing target for our work is precisely this *approximation* – the hardware-accelerated VMM, which has been found to have serious bugs [3, 8, 24].

Amit et al. [3] recently identified several VMM bugs by applying to KVM the closely guarded test cases that Intel uses to test their physical CPUs. However, these test cases suffer from a limitation similar to that of PokeEMU's: they were not created by analyzing the actual VMM implementation under test or its complex interactions with the processor extension.

In practice, none of the previous VMM testing methodologies *automatically* generate test cases by simultaneously analyzing both the specification and the implementation. Furthermore, to our best knowledge, MultiNyx is the first automated and systematic tool that generates tests specifically for testing *hardware-accelerated* VMMs.[2]

Generic Symbolic Execution. Since the idea of symbolic execution was initially proposed decades ago [28], a myriad of generic symbolic execution engines have been developed to address the real-world challenges of applying this powerful idea. Initial symbolic execution engines (e.g., KLEE [12], FuzzBALL [32]) analyzed code at the level of intermediate representations, which have relatively simple semantics, to mitigate scalability issues. However, several systems have recently leveraged symbolic execution techniques to directly analyze generic x86 code [13, 23, 38]. Despite these advances, few systems have been proposed to symbolically analyze system-level x86 code. Moreover, even fewer have found bugs directly involving complex semantics of system-level instructions.

S2E [13] consists of a VM-based symbolic execution framework implemented on QEMU that can analyze arbitrary x86 code. Unfortunately, despite having some support for SVM, QEMU is a low-fidelity emulator [32] designed to run VMs fast, not necessarily realistically. Thus, it is unlikely that conducting KVM tests in S2E would exhibit the same behavior as in real hardware. Instead, our work leverages a generic symbolic execution engine for application-level x86 code, Triton [38], and proposes a methodology to automatically augment its semantics using a high-fidelity executable specification (Bochs) [32]. Furthermore, our work addresses other real-world challenges in applying symbolic execution to test hardware-assisted VMMs.

VMM Verification. Previous work has applied verification techniques to hypervisors in an attempt to provide formal guarantees of correctness [30, 39, 40]. This is a promising direction, although its success has been limited to the verification of a subset of a hypervisor due to a lack of formal hardware specifications and the hardware complexity [1, 15].

3 INTEL VMX VIRTUALIZATION EXTENSION

Intel VMX extension conceptually duplicates the x86 processor state by proposing a new operating mode, the root mode. VMM code runs under root mode, whereas VM code runs under the non-root mode. The root and non-root modes are specifically intended for virtualization and are orthogonal to traditional execution modes (long, protected and real modes) and to privilege levels (i.e., rings). In particular, VM code can run, in non-root mode, in any execution mode and with any privilege regardless of the VMM.

In root mode, 11 new instructions are made available for VMMs to interact with the processor extension [26]. In addition, one instruction (`VMCALL`) in non-root mode lets the VM perform hypercalls

[2] Google reportedly has used a fuzzer to test KVM [24], but we were unable to find documentation regarding its design details.

(i.e., invoke the VMM). In addition to adding instructions, VMX introduced the concept of Virtual Machine Control Structure (VMCS), a key virtualization structure in RAM that is accessed by the VMM through the new instructions.

The VMCS consists of an extensive array of memory-mapped registers (fields) that serve many purposes (Table 1). For instance, the VMCS stores the VM register state when the VM is suspended (i.e., the VMM code is executing in root mode). It is also the place holder for the VMM register state when the VM is running. Further, the VMCS stores control information about the VMX settings; this includes options to control the VM exit conditions and options to enable and disable different virtualization features. Despite being mapped to memory, according to the Intel specification, the VMCS is mostly an opaque structure that, except for its first eight bytes, must be read and written by executing the `VMREAD` and `VMWRITE` instructions.

Under VMX, a VMM generally performs the following: (1) detects the CPU capabilities based on the `CPUID` instruction and the model specific (`MSR`) registers, (2) activates VMX (`VMXON`), (3) loads a VMCS, (4) initializes the VMCS through a series of invocations of `VMWRITE`/`VMREAD`, depending on the model/capabilities of the host CPU and the desired virtualization settings, (5) sets up the MMU related-components using shadow paging or EPT tables, (6) attempts to perform a VM entry with `VMLAUNCH`, (7) identifies the `VMLAUNCH` return reason, typically a VM exit (the alternative is a failed VM entry, generally a sign of a software bug), (8) if `VMLAUNCH` returned because of a VM exit, based on the specific exit reason, the VMM independently addresses the condition (e.g., emulates an instruction or addresses page faults) or otherwise returns to user-mode (e.g., to serve an IO request), and (9) repeats the virtualization cycle by going to step 5 to reenter the VM using instead the `VMRESUME` instruction.

The current version of the architecture manual specifies 64 different codes for "basic VM exit reasons". Most exit codes refer to VM attempts to execute instructions that the VMM must intercept and emulate because VMX does not support its virtualization. For instance, VMX has only very limited support for nested virtualization; as a result, the VMM has to emulate the VMX instructions executed by the VM to allow nested virtualization [6]. The `CPUID` instruction and accesses to the `CRx`, `GDTR`, `LDTR` and `MSR` registers are other system instructions often emulated.

Another important reason for emulation is the need to support IO. In x86, IO can be performed using the `IN`/`OUT` port instructions or memory-mapped IO (MMIO). Regardless of the mechanisms used, the VMM must handle IO operations because VMX virtualizes the CPU and memory but not devices; this task is often left for applications like QEMU unless there are performance considerations (e.g., KVM virtualizes a few interrupt controllers).

When a VM exit occurs, because VMX provides limited assistance in decoding instructions (e.g., exit reason or faulting addresses), the VMM must fetch the instruction, decode it and fetch its operands, in addition to emulating it. This is another challenging task for VMMs given the complex instruction encodings of x86, characteristic of CISC architectures. Furthermore, to emulate some instructions and events (e.g., interrupts and exceptions), the VMM must perform VM introspection, analyzing the VM state and understanding the x86 semantics in detail, especially, page table structures, interrupt descriptors, segment descriptors, and the many execution modes (long 64-bit, long compatibility, protected, real and vm86 mode). In practice, several VM exits can occur while virtualizing a single VM instruction because of conditions the VMM needs to address (e.g., multiple memory accesses that require intervention of the VMM).

#Fields	Description
63	Guest-state fields
23	Host-state fields
54	Control fields
15	VM-exit information fields

Table 1: Main classes of fields in the Virtual Machine Control Structure (VMCS) of the Intel VMX extension [26].

4 OVERVIEW OF MULTINYX

MultiNyx consists of two main components. The first component automatically generates new test cases for a given VMM implementation. It relies on running an initial (manually written) test case on an executable specification and, in the process, collecting an execution trace. Subsequently, this trace is symbolically analyzed to produce new test cases that will explore different paths of the VMM implementation or executable specification. The generated test cases are then themselves executed on the executable specification, and their respective traces symbolically analyzed. This cycle is repeated until a sufficient number of test cases has been generated.

The second MultiNyx component runs the test cases in different execution environments, cross-checking their results. This process lets users analyze the results in real-world execution settings to determine whether the test results match the expected outcome or, otherwise, suggest a defect in the VMM implementation.

5 MULTINYX DESIGN

The following sections describe our approach to making VMMs amenable to systematic analysis, as well as the two MultiNyx components for test generation and cross-checking.

5.1 Deconstructing VMMs for Testing

An important observation that both components of MultiNyx rely on is that it is possible to reduce the VMM exploration and analysis to small units of execution. In fact, virtualizing a single instruction for a given (virtual) CPU state is feasible and sufficient for good coverage of the VMM behavior because the state it operates on can be set externally.

The typical approach used both by manual tests and fuzzing techniques is to boot the VM and have the machine execute a long series of instructions, from within the VM, to initialize itself and reach a particular *target* VM state. From then on, these approaches trigger the execution, of a set of VM *target* instructions. MultiNyx follows a different approach, it directly provides the target VM state when initializing the VMM, by setting it *externally* with VMM-specific functions, and then instructs the VMM to run a single target instruction.

Both VMX and KVM implementation allow a VM to start in an arbitrary state (within the architecture constraints). The VMX

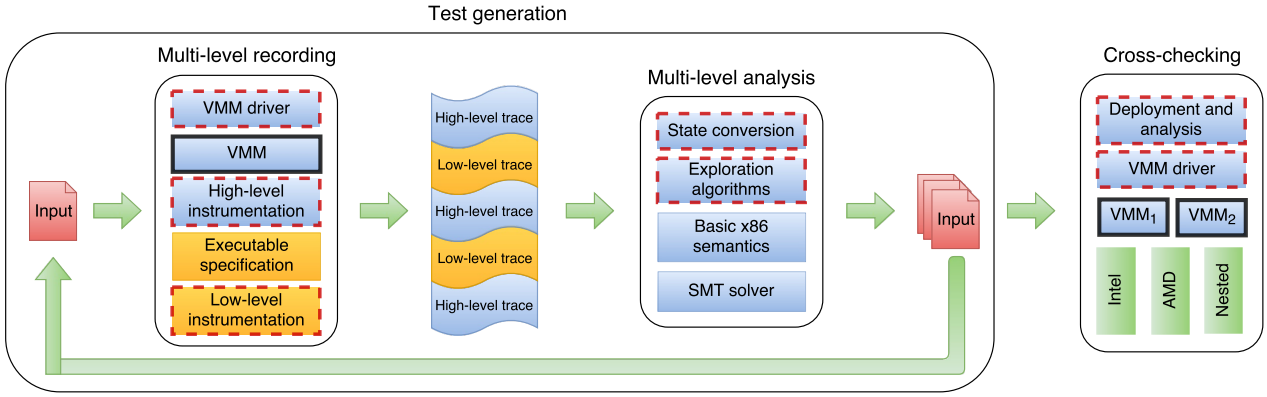

Figure 1: MultiNyx architecture. Generation of new test cases relies on the recording and analysis components. The multi-level recording component receives a concrete test case and produces an execution trace. The multi-level analysis component subsequently produces several concrete test cases by symbolically analyzing the trace from the recording component. The generated tests are fed into the analysis component, iterating the process thus generating more tests, or executed under different VMM configurations to check the test results.

extension enables VMMs to set the initial VM state when initializing the VMCS guest VM fields (Table 1) – in fact, this in an important feature when migrating VMs from another physical machine. KVM allows the user-mode application driving it, through the Linux `ioctl` system call, to externally set the initial VM register values; in KVM, this task is performed by invoking the API functions `KVM_SET_REGS` and `KVM_SET_SREGS`. Likewise, the KVM driver can set the initial VM RAM to contain the desired in-memory structures (e.g., instructions, memory operand values, page tables, interrupt and segment descriptors) using `KVM_SET_USER_MEMORY_REGION`. Single stepping through the instructions is configured by executing the KVM call `KVM_SET_GUEST_DEBUG`.

Both the exploration and analysis components rely on a custom MultiNyx VMM driver. This driver instructs the VMM to initialize the VM with a given starting state (VM registers and VM memory) in single-stepping mode. Furthermore, the VMM driver collects the final VM state after the target instruction is executed.

5.2 Test Generation Component

The generation component generates new test cases that trigger execution paths of interest for the cross-checking component. At a high-level, it achieves this by executing the VMM with a given concrete input (using the VMM driver) and recording a trace of the combined execution of the VMM and VM. Next, the combined trace is provided to an application-level symbolic execution engine, that generates and solves path constraints to produces new inputs based on the exploration strategy.

5.2.1 Multi-level Trace Recording. The mechanism to generate the trace is critical because the symbolic execution engine used by MultiNyx, like most, is not aware of the system-level code semantics. In particular, the engine knows nothing about interrupts, page tables, execution modes or processor extensions. Thus, it does not know how to symbolically execute a complex instruction like `VMLAUNCH`. In fact, even the full system semantics of a seemingly simple instruction like `MOV` or `RET` are very complex: it takes nearly 1 page of the Intel manual to describe the (incomplete) pseudo-code of `MOV`, and more than 7 pages to describe the `RET` instruction. The manual's current version makes no attempt to describe the pseudo-code of the VMX instructions.

MultiNyx collects a trace of the VMM implementation, *high-level trace*, containing the sequence of addresses of the executed VMM instructions. Additionally, the trace also includes the memory accesses (address, size, value and R/W type) to allow the inference of the initial memory state (and aid debugging). When complex system-level instructions are executed, namely `VMLAUNCH`/`VMRESUME` and the VM instructions that run in non-root mode, MultiNyx collects instead a trace of the executable specification, a *low-level trace*, as opposed to a direct trace of the VMM/VM instructions. Similarly, the low-level trace also includes the memory accesses to allow the inference of initial memory state. This tracing process produces a combined trace with several segments of different semantic levels of abstraction (high-level and low-level).

Collecting an executable specification trace is equivalent to collecting a trace of the executed pseudo-code associated with each instruction, as described in the architecture manual. However, the executable specification is more amenable to automatic analysis and is expected to be significantly more exhaustive. §8.4 discusses the limitations associated with an incomplete or incorrect executable specification, and §6.1 explains the mechanism our implementation uses to collect both types of traces.

5.2.2 Multi-level Trace Analysis. The two types of traces collected in a single execution operate at different semantic levels and thus on *different representations* of the VMM/VM state. Furthermore, the executable specification contains state that does not have a representation at the implementation level. For instance, the pseudo-code in Intel's manual has the concept of the machine's `EIP` register but it also has its own "`EIP` register"; the latter consists of the pseudo-code line number. An *executable* specification instead has an actual `EIP` register that is distinct from the higher-level `EIP` register value (represented in the executable specification memory).

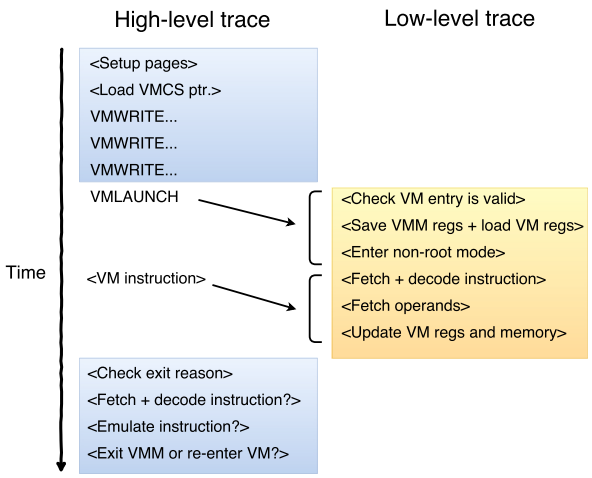

Figure 2: Trace example with three segments.

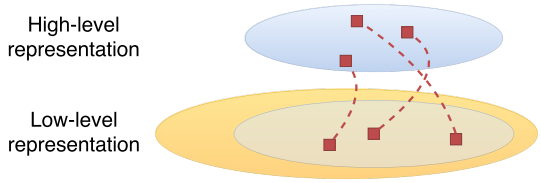

Figure 3: State mapping between high-level and low-level representation. The low-level representation includes all the state of the high-level representation and additional state that is specific to the low-level representation.

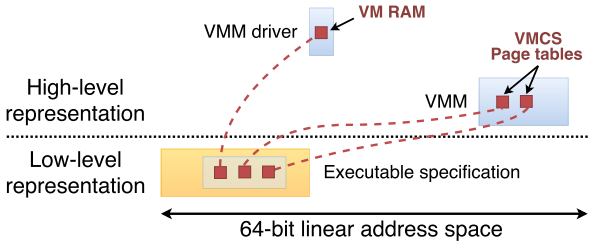

Figure 4: Memory layout of the systems analyzed. MultiNyx ensures that the VMM, the VMM driver, and the executable specification use non-overlapping areas in the 64-bit address space.

For both levels, two state components are particularly important: (1) the memory contents and (2) the processor registers values. For each trace segment and regardless of the type of trace, MultiNyx ensures that the memory state is collected by simply recording the memory accesses, as discussed earlier. When analyzing the trace, MultiNyx first makes a pass on the trace to find the values first read for all memory locations that segment accesses, thereby creating a minimal representation of the initial memory snapshot. To preserve the register values, MultiNyx records the register values observed at the beginning of the segment, for each segment of the trace.

The concrete data recorded is sufficient to concretely analyze the execution. However, MultiNyx requires additional information because it analyzes the execution symbolically to determine the conditions (inputs) that can lead to other execution paths. In particular, MultiNyx requires the annotation of the VMM variables that correspond to the execution test input; these consist of the initial VM register values and the initial VM RAM contents (§5.1). Throughout the analysis of the trace instructions, MultiNyx constructs the symbolic expressions, thus creating a symbolic representation of the VMM state.

MultiNyx takes special care during trace segment transitions to preserve the symbolic state. In particular, it transfers the high-level symbolic state into symbolic state that matches the low-level representation or vice-versa, depending on the trace transition (Figure 3). To transfer the symbolic state associated with memory, MultiNyx records the *mapping* of state between the two levels during the recording phase and then *applies* the mapping to transfer the symbolic state during the symbolic analysis phase.

All the memory of the high-level trace is represented in a buffer of the executable specification. Because this buffer represents the physical (RAM) memory of the machine it is indexed by physical addresses, as opposed to linear addresses used in the high-level trace. Therefore, the mapping function needs to perform two tasks: (1) convert the linear memory addresses of the high-level trace into physical addresses and (2) convert the physical addresses into the corresponding addresses of the executable specification, typically by adding the starting address of the RAM buffer. In practice, the VMCS, the VMM page tables (or EPT), and the VM RAM are the most important memory structures to transfer between the two analysis levels because they depend on the input and can affect the VMX mode operation. We note that the exact memory state mapping function is specific to the particular executable specification used, but our experience shows that it was easy and required writing only a few lines of code (§6).

The current implementation of MultiNyx leverages the fact that memory addresses of kernel-mode code (VMM) and user-mode code (executable specification) do not overlap (Figure 4). Our implementation also ensures that the VMM driver allocates the VM RAM at addresses that differ from those used by the executable specification so that they also do not overlap. The fact that the addresses of both levels do not overlap lets MultiNyx symbolically execute the two types of traces without interfering with each others' address space and without having to modify the internals of the symbolic engine. This non-overlapping property allows MultiNyx to transfer the symbolic state by simply copying, on each transition, the associated symbolic expressions between addresses in accordance with the mapping.

To transfer the register values, MultiNyx exploits the semantics of the processor extensions to avoid unnecessarily transferring the symbolic values of registers. According to the semantics of `VMLAUNCH` and `VMRESUME`, except for the general purpose registers, the VMM register values are discarded when performing a VM entry (and replaced with the guest register values contained in the VMCS) and are reloaded after a VM exit (either from the VMCS host area or from the VMM stack). Therefore, MultiNyx only transfers symbolically the values of those registers that persist across VM

entries and VM exits because the others are implicitly transferred through memory.

In addition to RAM contents and processor register values, a few other types of state exist in a machine, such as IO devices state. Based on our analysis of KVM, the MSR registers were the only other type of state relevant. KVM uses these registers to detect the virtualization-related capabilities of the host machine. MultiNyx handles these registers by recording them and simply replaying them during analysis because their values do not depend on the symbolic inputs and can thus remain concrete.

5.2.3 Exploration Algorithm. The analysis of the multi-level trace enables MultiNyx to generate new inputs that correspond to execution paths or data paths that differ from those of the given trace. The extensible MultiNyx design permits the deployment of different exploration algorithm plugins. We wrote two plugins, implementing a coverage-based and a data-path-based exploration algorithm, to demonstrate how the MultiNyx framework can systematically analyze VMMs to generate new test cases.

The *coverage-based algorithm* relies on prior work approaches [23] that construct path constraints as the trace is analyzed. Throughout the trace analysis, the constraint of each conditional branch is determined and appended to the constraint of the path prefix already analyzed. In addition, for each conditional branch the conjunction of the negated branch condition (i.e., the branch not taken) with the path prefix constraint is sent to the solver, generating a new input that forks on this branch.

The *data-path-based algorithm* relies on the observation that the values of certain variables are particularly important in the VMM/VM problem domain. Notably, the CPL (current privilege level) field, a sub-field of the code segment (CS) register, is particularly important. The CPL determines the ring privilege level of the running VM code so a correct VMM has the critical task of ensuring that the VM code can only modify the level, specially reduce it, in accordance with the architecture semantics to prevent privilege escalation attacks. For each trace, the data-path-based algorithm creates a formula that checks whether the same execution path could lead to a final CPL value that is different from both the observed initial and final trace CPL. Solving this formula allows MultiNyx to find VMM inputs, e.g., VM RAM contents, that cause the VMM to set the CPL value without appropriate checks.

5.3 Cross-checking Component

The cross-checking component analyzes a given VM test by comparing its results when running under different scenarios. This is important because MultiNyx does not assume that the executable specification used by the test generation component is correct. In fact, hardware itself significantly varies due to differences in features and specific virtualization extension supported. The cross-checking component therefore conducts a differential analysis on a given test by running the VMM with the same input but under different real-world environments, thereby aiding the user in analyzing the MultiNyx results. This approach lets users conduct tests by exploring different types of dimensions for the differential analysis: (1) different target VMM versions, (2) different configurations of a given VMM, (3) different hardware architectures, and (4) bare-metal virtualization vs. nested virtualization.

Comparing different target VMM versions lets users assess whether changes to a given VMM introduced bugs or otherwise correctly changed the VMM semantic, for instance, by adding a new feature. Similarly, comparing different configurations or hardware versions enables users to understand whether the behaviors across different configurations or hardware are consistent. Finally, the nested/bare-metal dimension enables MultiNyx to test two systems simultaneously – it analyzes the behavior of the VMM when itself runs inside a VM (possibly virtualized by a different VMM implementation).

6 IMPLEMENTATION

The MultiNyx implementation includes components that we wrote (Table 2) in addition to different off-the-shelf components, as detailed in the following sections.

Component	Language	LOCs
KVM driver	C	2,400
KVM annotations	C	1,400
Low-level trace recording	C++	600
High-level trace recording	C++	1,300
Multi-level analysis	C++/Python	3,100
Cross-checking and diagnosis	Bash/Python	4,400

Table 2: Components of the MultiNyx implementation, their language and the approximate number of lines of source code.

6.1 Multi-level Traces

MultiNyx currently uses Bochs [37] as the executable specification and records the high-level trace using a Bochs instrumentation library that we wrote. To implement MultiNyx, we made only a few minor Bochs-specific modifications (approximately 30 lines of source code). Thus, our design makes it easy for MultiNyx to accept other executable specifications. In particular, we modified the Bochs instrumentation API to provide additional information, such as the values read on logical memory accesses, the VMREAD/VMWRITE operations, and the VM entry and VM exit. We also added the ability to suppress interrupts on the machine when running the VMM to ensure that we could record minimal traces. Further, we implemented the multi-level state mapping function using existing Bochs functions, namely, the function to convert machine logical to physical addresses and the function that maps machine physical addresses to Bochs (linear) addresses (i.e., the buffer used by Bochs to emulate the machine RAM).

The low-level trace is recorded by leveraging the Pin binary instrumentation framework [31] and is completely independent of the executable specification. We wrote a pintool (i.e., a Pin instrumentation module) that records the instruction and memory trace of Bochs itself while it executes VMM/VM instructions: the VMLAUNCH/VMRESUME, the VM instruction, and the corresponding VM exit. Our Bochs instrumentation library spawns the pintool when the first VM entry is detected. After launch, the MultiNyx pintool adds instrumentation that is triggered when the Bochs VM entry and VM exit functions are invoked, and enables the instrumentation of all instructions executed by Bochs. During runtime,

the pintool instrumentation detects when the VM exit function is invoked and removes the instrumentation of Bochs instructions to reduce the instrumentation overhead; however, it keeps the VM entry/exit instrumentation to reactivate the instruction instrumentation when recording the next trace.

We implemented a hypercall mechanism in the Bochs instrumentation library that allows VMM code to communicate with Bochs and append annotations to the trace. The hypercall is triggered when a `NOP` instruction is executed with specific register values. MultiNyx uses this hypercall to annotate symbolic inputs and as a flexible debugging aid.

6.2 KVM Driver and KVM Annotations

The KVM driver we built for MultiNyx receives a RAM image file and a file with initial register values. When an execution starts, the driver pins the VM RAM buffer to memory to avoid page faults. In addition, MultiNyx lets users generate the initial VM RAM by automatically creating basic page tables and segment descriptors and by assembling instructions from given assembly code. This enables users to provide the first input to MultiNyx so that it can generate and analyze additional inputs.

Our MultiNyx implementation annotates symbolic inputs in the KVM code. In particular, we annotate the VM register values and the VM RAM. Most of the code added to KVM consists of an optional mechanism, implemented with `debugfs`, that extracts the host and guest register values for every VM entry and VM exit. This mechanism makes possible the differential analysis of intermediate register values providing additional diagnosis information to users.

The KVM driver and the KVM annotations are the only KVM-specific code that is required to employ MultiNyx. The use of a test driver is a requirement for any dynamic testing technique and only tens of lines of annotations are required to annotate the KVM input. As such, we expect that employing MultiNyx on other VMMs will involve minimal developer effort.

6.3 Multi-level Analysis

MultiNyx performs the multi-level analysis by leveraging: (1) Triton [38], a symbolic execution framework aware of only the basic x86 instruction semantics, and (2) Z3 [22], a state-of-the-art SMT solver supported by the symbolic execution framework. While parsing the multi-level trace, MultiNyx uses Triton to create the symbolic expressions and send them to Z3 according to the exploration algorithm we implemented. We changed the framework to cache and efficiently measure the size of constraints, and use the Z3 stack function. We implemented part of the exploration algorithms in C++ to reduce overheads associated with Python bindings; we implemented the less performance-critical parts in Python.

The `VMWRITE` and `VMREAD` instructions (VMX instructions) are analyzed by MultiNyx without resorting to a low-level trace because they have relatively simple semantics. To execute these instructions symbolically during the trace analysis, MultiNyx converts them into equivalent `MOV` instructions that write/read into and from the appropriate locations in the VMCS. Because the internal structure of VMCS is opaque (i.e., not documented by Intel), we inspected the Bochs source code to adopt the same structure that the executable specification implements for the VMCS; furthermore, our Bochs instrumentation library stores the values read from the VMCS and the offsets/size of the accesses. Taken together, this approach simplifies instruction simulation and ensures that `VMWRITE` and `VMREAD` simulations are consistent with the low-level trace used for complex instructions like `VMLAUNCH`. In addition to the `MOV` instruction, simulating these instructions also requires setting the return value and, in some cases, padding registers. This approach represents another possibility for augmenting the semantics of symbolic execution frameworks that is simple for low-complexity instructions and, in such cases, has the advantage of adding fewer instructions to the trace than resorting to a low-level trace of the executable specification.

Our implementation incorporates two mechanisms to improve scalability, which is a challenge specially for the symbolic analysis of x86 instructions. First, we selectively add branch constraints to the path constraint if their size falls below a certain threshold. This could affect soundness by generating inputs that do not generate new execution paths, a relatively benign effect that still ensures the approach is significantly more systematic than fuzzing. However, in general, the constraints that are not included are related to internal VMM memory allocations and are not expected to affect the analysis. Second, our algorithm assigns higher priority to branches that fork to addresses previously unexplored by MultiNyx. This heuristic lets MultiNyx focus on generating inputs that explore undiscovered paths in the VM/VMM.

6.4 Cross-checking

We implemented the cross-checking component in Python and Bash. MultiNyx relies on this component to send inputs to different execution environments (bare-metal, nested virtualization), different VMM configurations or even different implementations. It then collects the intermediate and final output of each test. Subsequently, MultiNyx analyzes the results, comparing the output across the different executions, and presents them to the user. We also implemented a tool for the user to query the results, searching for specific mismatches in the different configurations (§7.2).

6.5 Optimizations and Parallelization

MultiNyx includes several important optimizations to make it scale. First, since Bochs is already slow compared to a bare-metal machine, we implemented optimizations in the MultiNyx pintool that reduce the instrumentation overhead by leveraging the "trace version" feature [27] and by switching between two instrumented versions of Bochs during runtime – one that records traces, for the few instructions that require it and another that executes in the common case at near-native (Bochs) speed, without producing traces. This optimization improved the recording throughput by up to 40x compared to our previous implementation.

Second, we implemented several Triton modifications that improve the efficiency when handling large constraints and others that encode instructions with fewer constraints. For instance, compilers generally emit optimized instructions – such as `XOR EAX,EAX` – that simply clear the register (and update flags). MultiNyx optimizes these cases by clearing the value of the register and updating the flags, as opposed to duplicating the size of the register constraint and expecting the SMT solver to subsequently optimize it.

In addition to these optimizations, we parallelize the two components of test generation, i.e., recording and symbolic analysis. Each component spawns parallel tasks that simultaneously process different test cases. In addition, our experiments also distribute the symbolic analysis tasks across several machines (§7.1).

7 EVALUATION

To understand the effectiveness of MultiNyx, we automatically generated more than 206,628 tests and applied them to a mature and stable version of KVM (Linux 4.12.5). This section reports on our experience and discusses the results attained.

7.1 Test Generation

We initialized MultiNyx with a single manually written test case for a previously patched bug (the `MOV` bug discussed in §2.1). There is no strict requirement for the initial test case, but using an initial test case that causes a deep execution of the VMM, such as a valid `MOV`, helps to find with fewer iterations other test cases that cause deep executions. From this test, MultiNyx produced a trace that was subsequently analyzed symbolically. In turn, the symbolic analysis generated more test cases. By continuously repeating this process, MultiNyx produced a total of 41,162 traces, of which 26,487 were symbolically analyzed, resulting in the generation of 206,628 test cases. The traces and generated test cases consisted of 2.3 TB of uncompressed data.

Our test cases specified the starting VM register values and the starting VM RAM contents. For our experiments we executed as symbolic inputs most of the general (e.g., `RAX`), control (e.g., `CR0`, `EFLAGS`), and segment (e.g., `CS`) registers and all of the VM RAM memory (64KB). In practice, however, most traces accessed only a small subset of the VM RAM because each test only executes a single VM instruction, since our driver configures KVM to single step. The observed memory accesses typically consist of the instruction fetch, segment descriptors load, page table entries accesses and possibly instruction-dependent accesses. Thus, as an optimization, MultiNyx only marks as symbolic input the RAM memory locations accessed in a given trace.

By solving the constraints (§5), MultiNyx finds new sets of values for the symbolic inputs that will cause the execution of alternative VMM execution paths. Each set of input values constitute a new test case. Because we annotate the VM RAM and the VM control registers as symbolic inputs, the generated test cases automatically explore, namely, different VM execution modes (e.g., real mode, protected mode), instructions and in-memory CPU structures (e.g., page tables and segment descriptors).

In our experiments, the concrete execution of a test case, under Bochs toke on average 8 seconds, while the symbolic execution of a trace toke 12 minutes. Because of the different computational demand of the two MultiNyx components, we configured our testing infrastructure to run 4 parallel instances of Bochs on a single machine and to distribute the symbolic analysis across 5 machines, running up to 12 parallel tasks on each.

The concrete execution component ran on a desktop system with a 12-core Intel Xeon E5-1650 @ 3.50GHz CPU and 16GB of RAM. The symbolic execution component ran on 5 servers, each with two 12-core Intel Xeon E5-2680 @ 2.50GHz CPUs and 64 GB RAM. In this setting, generating all tests toke less than 3 days.

7.2 Test Results

After generating the test cases, we applied them to KVM on bare-metal machines under different configurations. We ran the experiments on two machines that had different virtualization extensions: (1) an AMD machine with an AMD Ryzen 7 1700 CPU (AMD), and (2) an Intel machine with an Intel Core i7-7700 CPU. Furthermore, we ran tests on the Intel machine with two different KVM configurations: (2a) the default configuration, which has EPT support enabled and nested virtualization disabled (Intel) and (2b) with EPT disabled and nested virtualization enabled (Intel w/o EPT).

Test results surprisingly showed a wide disparity in the behavior of KVM. Table 3 shows the number of tests that produced each of the different *KVM exit values*[3] when executing them on the three configurations. KVM returns to user-mode when it stops executing a VM because user-mode intervention is required – the exit code describes the reason for the KVM exit. For instance, KVM needs to exit because of IO operations performed by the VM (IO), triple faults or other VM operations that cause the VM to shutdown (shutdown), illegal state of the VM that prevents a VM entry (entry fail) or completed VM instruction execution when VMX is configured to single step (debug).

All the inconsistencies revealed in Table 3 are problems that should not happen in a robust VMM. Even hanging situations, for instance, can expose systems to denial of service attacks potentially seriously affecting users. In addition, the fact that the VMM behaves inconsistently across configurations could mask problems to developers that rely on KVM to build not just QEMU-like hypervisors but also other forms of sandbox mechanisms.

Config	Hang	Entry fail	Shutdown	IO	Debug
AMD	13,050	6,899	125,836	88,763	48,803
Intel w/ EPT	30	69,199	90,908	50,880	38,622
Intel w/o EPT	5,786	20,635	90,388	90,044	67,740

Table 3: Total number of tests based on their observed KVM exit value for each execution configuration. Tests can either hang or cause KVM to exit with a return reason (VM entry failure, VM shutdown, IO handling required, or debug). Some tests cause more than one KVM exit (e.g. several IO exits), despite executing a single VM instruction.

Even more concerning, we observed that many test cases produced the same KVM exit value but distinct final VM states (register values or memory contents) depending on the configuration. One reason for this is that the KVM interface does not guarantee that setting certain initial values for registers will be accepted; thus, part of the input may be ignored and inconsistently so across configurations. Accordingly, our test driver additionally read the register values (effective VMM input) that were accepted by KVM before initiating the execution of the VM. This information allowed us to identify the test cases that executed with the same *effective* VMM

[3] Note that a KVM exit is distinct from a VM/VMX exit. The former refers to a transition between kernel-mode to user-mode while the latter refers to a transition between VMX non-root mode (VM) to VMX root mode (VMM).

input but did not produce the same final VM state, i.e., cases likely to constitute serious bugs because they lead to silent failures (i.e., hard-to-detect, semantic bugs).

Table 4 and Table 5 shows the count of test cases in which KVM reported the successful execution of the tested instruction and matching effective VMM input but *mismatching* final VM states. These are the test cases that cause silent KVM failures and are particularly hard to detect without resorting to differential testing.

Across the Intel configurations (with vs. without EPT enabled), MultiNyx identified 98 mismatching test cases and, even worse, 641 across architectures (Intel vs. AMD). Additionally, Table 4 and Table 5 present the count of mismatching tests based on the VM state subset that did not match. For instance, 48 and 69 tests cause the instruction pointer (`EIP` register) to differ across the two cross-checking dimensions. Many of these test cases likely have the same or related underlying causes. However, a more detailed analysis and the fact that different test cases cause different parts of the state to mismatch suggests that they reveal several distinct bugs.

To simplify the analysis of these results we built a tool that allows users to query the results, searching for different aspects of the tests, such as specific mismatching registers. Our tool automatically identifies the differences and displays them side-by-side.

Component	Architecture	EPT
`RFLAGS`	68	66
`RIP`	69	48
General purpose reg.	81	37
Segment reg.	628	19
Total	641	98

Table 4: Count of tests generated by MultiNyx that reported a mismatch in the different components of the final VM state. A single test can reveal several mismatching components of the state.

Segment reg.	Architecture				EPT			
	Sel	Base	Limit	Attr	Sel	Base	Limit	Attr
`CS`	57	57	0	0	0	1	0	3
`DS`	0	0	16	356	1	1	9	6
`ES`	0	0	0	0	1	1	3	3
`SS`	5	5	0	621	0	0	2	0

Table 5: Count of tests generated by MultiNyx that reported a mismatch in the final VM segment register state. The segment register state includes the segment selector, base address, limit, and attributes. A single test can reveal several mismatching components of the segment registers state.

A closer inspection of the mismatching tests revealed that many of them (19 and 628) were related to segmentation and could affect the virtualized protection mechanisms. For instance, we reported to KVM developers the results of a test case that causes a `PUSH ES` instruction to change the stack pointer by 4 bytes under Intel but by 2 bytes under the Intel configuration without EPT support enabled. Accordingly, the test case causes the memory of the VM to be modified at different offsets. To trigger this problem, the VM had to be configured in real mode and specific bits of the segment registers had to be enabled, in particular the `CS.DB` bit, which specifies the operations size, had to be enabled. Another test case found that instructions, such as `OR AX,ESP`, ignored the top-most 16-bits of the `ESP` register depending on whether EPT was enabled. We reported the bug to the developers who confirmed the results and concluded that these two situations were caused by the same underlying problem – a misunderstanding of the VM mode of operation whereby KVM ignores part of the segment registers. Since then, KVM developers have applied a combined patch to fix these problems.

Another test case generated by MultiNyx affects the segment protection mechanism of the AMD version of KVM. This test case revealed that the `CPL/DPL` fields, which contain the privilege of execution (i.e., the ring level) and are part of the `CS` and `SS` segment registers, when initialized by a test case to 3 (user-mode) would be cleared to 0 (kernel-mode) during the test. This situation suggests that the bug could enable privilege escalation by code running in the VM and is currently still under investigation.

All the problems we reported were successfully reproduced by KVM developers. Further, we expect developers to be able to reproduce all the test case results from Table 3, Table 4 and Table 5 because they were obtained by running bare-metal KVM (i.e., without Bochs), and the MultiNyx test driver is relatively small and has itself been well tested.

Our experience applying MultiNyx to KVM revealed disparate behaviors, especially when handling errors (e.g., entry failures) or unusual situations that are exposed to the application driving it. In large part, this results from the complexity of the virtualization extensions and by the lack of checks conducted by the KVM implementation that would otherwise provide a uniform and consistent interface across the various KVM configurations. Unfortunately, the current software and hardware design invites application developers to make assumptions about the VMM that hold only under certain configurations and thus can lead to bugs. Furthermore, the disparity of behaviors has the unfortunate side-effect of making the testing result analysis more challenging and hence costlier.

7.3 MultiNyx Coverage

To better assess the effectiveness of MultiNyx in generating tests that cover well both the KVM implementation and the VMX specification, we computed the instruction coverage based on a 25k sample of traces from our experiments (Table 6). To ensure that our traces are manageably sized, specially with respect to the low-level traces, our instrumentation and testing infrastructure is selective and collects traces of only a subset of all instructions executed in the test (§6.1). Given this, the full coverage of tests is expected to be higher than the reported values.

In our experiments, we disabled address space randomization of the host machine running Bochs to ensure that instruction addresses were constant across tests (e.g., when running parallel versions of Bochs). For similar reasons, we also stored the load address of the KVM kernel modules, which are dynamic and relocatable, to allow the identification of the trace instructions that were effectively executed. The kernel binaries and Bochs were compiled with the default optimizations, but we enabled debugging symbols to simplify the manual inspection of traces.

Component	Fuzzing	MultiNyx	Diff
High-level			
kvm.ko	10,299	13,750	+34%
kvm_intel.ko	2,109	2,288	+8%
Low-level			
Specification (bochs)	11,908	49,957	+319%

Table 6: Absolute and relative instruction coverage of the fuzzing and MultiNyx tests. The absolute numbers report the count of unique instructions executed under instrumentation in a random sample of 25k traces.

Table 6 compares the instruction coverage of MultiNyx to that of a fuzzing testing strategy that uses the same KVM test driver. We implemented a simple fuzzing algorithm that starts from a given test case and randomly flips bits of memory and register values with a given probability. The starting VMM input for the fuzzing strategy was the same as that used to bootstrap MultiNyx (§7.2). The fuzzing algorithm served as a baseline for the evaluation of our coverage, since there are no fuzzing tools publicly available for HW-accelerated hypervisors, and could be improved as others have done in different testing contexts. §8 discusses how to combine MultiNyx with fuzzing strategies to address its limitations and further improve the testing effectiveness.

As expected, the comparison of coverage results showed that tests produced by MultiNyx had a higher coverage than fuzzing tests, with regard to both the KVM implementation (`kvm.ko` and `kvm_intel.ko` modules) and the specification (Bochs). The difference in coverage was particularly prominent regarding the specification – our results reflect a 319% increase – which is expected given that randomly generated tests are very unlikely to pass the KVM-implemented checks on the VM state (despite having insufficient checks, as discussed in §7.2). Thus, fuzzing tests are unlikely to even reach the point where KVM tries to enter the VM (e.g., VMX non-root mode), when the testing infrastructure activates the low-level trace recording. It is worth noting that even small increases in coverage can be very meaningful because bugs often hide in the few lines of code that are rarely executed.

8 DISCUSSION

8.1 Multi-level vs. Single-level Analysis

MultiNyx adopts a multi-level analysis approach to test VMMs. Alternatively, we could have chosen a single semantic level analysis, either entirely low-level or entirely high-level. A single-level analysis would have serious drawbacks, depending on the level of abstraction considered, as described next.

An entirely low-level analysis would have limited scalability because it would dramatically increase trace sizes. This would make the recording and storing of traces significantly more expensive. More importantly, it would prevent symbolic analysis from scaling since it is memory and compute intensive, and, beyond a certain point, SMT solvers cannot solve constraint formulas. In our experiments, approximately half of the traces were high-level (VMM instructions), while the other half were low-level (specification). For deep paths, each level could have more than 100k instructions. This shows that recording just two instructions at the low-level (a VM-entry instruction and a single-stepped VM instruction) can expand trace size by up to 50,000 times, although in this case the instructions are particularly complex. If applied to the entire test case, this expansion would prevent standard analysis techniques from scaling.

On the other hand, an entirely high-level analysis would be impossible without somehow adding the semantics of the virtualization extensions. Manually encoding these complex semantics in a symbolic engine would be tedious and error prone. Our work shows that it is practical and effective to use the multi-level analysis approach to automatically model extension semantics.

8.2 Testing Single vs. Multiple Instructions

The current implementation of MultiNyx tests single VM instructions by configuring the VMM to single step once, for each test case. In practice, it would be trivial to test groups of instructions by single stepping several instructions sequentially but this approach would make traces larger. Thus, it could hinder the scalability of symbolic analysis. Conducting tests with groups of instructions, on the other hand, could have the benefit of enabling MultiNyx to test VM instructions on states that might not be possible to set externally using existing VMM-specific functions.

8.3 Concurrency

The test cases generated by MultiNyx configure the VM with a single virtual CPU. Hence these test cases do not test the correctness of the VMM with regard to concurrency, which is mainly exercised when virtual machines are configured with multiple virtual CPUs. In such situations, multiple CPUs can execute instructions that cause concurrent VM exits and consequently concurrent VMM code execution. Systematically testing VMMs for concurrency bugs is future work that may benefit from existing symbolic execution techniques for multi-threaded software [7, 10].

8.4 Specification Accuracy

The design of MultiNyx requires an executable specification, and our implementation leverages Bochs for this purpose. In practice, however, Bochs is not a completely accurate hardware specification. In fact, during our experiments recording traces from automatically generated tests cases, we found at least two test cases that caused Bochs to "panic" because of an assertion violation.

One approach to address the limitations of Bochs is to simply use more accurate executable specifications. Intel, for instance, reportedly has an exact architectural simulator that is used in internal testing procedures [3]. Replacing Bochs with such a simulator would further increase the effectiveness of MultiNyx by exploring additional corner cases of the VMM implementation.

If more accurate executable specifications are unavailable, another approach to mitigate the impact of Boch's inaccuracies is to complement MultiNyx with fuzzing strategies. This could be achieved using MultiNyx to produce an initial corpus of test cases that have high, but not full, coverage and subsequently fuzzing these test cases hoping to further increase their coverage.

8.5 Processor Extensions

As Baumann recently observed, hardware manufacturers have been introducing at a fast-pace unusually complex hardware features

with processor extensions [5]. Often, such features provide advanced software functions that were traditionally left for other software layers to implement. The virtualization extensions that we address in this work are notable examples of this trend. As future work, we consider exploring the applicability of the general approach we propose in this work to other advanced processor features (e.g., SGX and IO virtualization).

9 CONCLUSION

This work presents the design of MultiNyx and our experience applying it to KVM. MultiNyx records an execution trace of the VMM implementation and selectively switches to recording a low-level trace of the specification when executing complex processor instructions. We show that this approach lets MultiNyx model the semantics of virtualization extensions and, by applying symbolic execution, effectively produces test cases that explore corner cases of both the implementation and specification. We applied MultiNyx to KVM across different execution configurations and found numerous output discrepancies.

REFERENCES

[1] Eyad Alkassar, Mark A. Hillebrand, Wolfgang Paul, and Elena Petrova. 2010. Automated Verification of a Small Hypervisor. In *Verified Software: Theories, Tools, Experiments*. Springer Berlin Heidelberg, Berlin, Heidelberg, 40–54.

[2] AMD. 2017. AMD64 Architecture Programmer's Manual Volume 3: General Purpose and System Instructions. https://support.amd.com/TechDocs/24593.pdf. (December 2017). Accessed: 2018-03-06.

[3] Nadav Amit, Dan Tsafrir, Assaf Schuster, Ahmad Ayoub, and Eran Shlomo. 2015. Virtual CPU Validation. In *Proceedings of the 25th Symposium on Operating Systems Principles*. Monterey, CA, 311–327. https://doi.org/10.1145/2815400.2815420

[4] Anonymous. 2017. Personal Communication. (June 2017).

[5] Andrew Baumann. 2017. Hardware Is the New Software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS)*. Whistler, Canada.

[6] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. 2010. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*. Vancouver, Canada, 423–436.

[7] Tom Bergan, Dan Grossman, and Luis Ceze. 2014. Symbolic Execution of Multithreaded Programs from Arbitrary Program Contexts. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. Portland, OR, 491–506. https://doi.org/10.1145/2660193.2660200

[8] Paolo Bonzini. 2014. The Security State of KVM. https://lwn.net/Articles/619332/. (November 2014). Accessed: 2017-09-22.

[9] Stefan Bucur, Johannes Kinder, and George Candea. 2014. Prototyping Symbolic Execution Engines for Interpreted Languages. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Salt Lake City, UT, 239–254.

[10] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel Symbolic Execution for Automated Real-world Software Testing. In *Proceedings of the Sixth Conference on Computer Systems*. Salzburg, Austria, 183–198. https://doi.org/10.1145/1966445.1966463

[11] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. 2017. *Hardware and Software Support for Virtualization*. Synthesis Lectures on Computer Architecture, Vol. 12. Morgan & Claypool Publishers. 1–206 pages.

[12] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*. San Diego, CA, 209–224.

[13] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Newport Beach, CA, 265–278.

[14] CloudSigma. 2017. Simple Cloud Hosting. https://www.cloudsigma.com/features/. (2017). Accessed: 2017-09-22.

[15] Ernie Cohen, Wolfgang Paul, and Sabine Schmaltz. 2013. *Theory of Multi-core Hypervisor Verification*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–27.

[16] KVM contributors. 2017. KVM Unit Tests. https://www.linux-kvm.org/page/KVM-unit-tests. (2017). Accessed: 2017-07-26.

[17] KVM contributors. 2017. KVM: x86: Fix Emulation of "MOV SS, Null Selector". https://github.com/torvalds/linux/commit/33ab91103b3415e12457e3104f0e4517ce12d0f3. (January 2017). Accessed: 2017-07-26.

[18] The MITRE Corporation. 2016. CVE-2016-8630. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8630. (October 2016). Accessed: 2017-07-26.

[19] The MITRE Corporation. 2016. CVE-2017-2683. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-2583. (December 2016). Accessed: 2017-07-26.

[20] The MITRE Corporation. 2017. Common Vulnerabilities and Exposures. https://cve.mitre.org/. (2017). Accessed: 2017-07-26.

[21] Victor Costan and Srinivas Devadas. 2017. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086. (February 2017). http://eprint.iacr.org/2016/086.

[22] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Budapest, Hungary, 337–340.

[23] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*. San Diego, CA.

[24] Google. 2017. 7 Ways We Harden Our KVM Hypervisor at Google Cloud. https://cloudplatform.googleblog.com/2017/01/7-ways-we-harden-our-KVM-hypervisor-at-Google-Cloud-security-in-plaintext.html. (2017). Accessed: 2017-09-22.

[25] Google. 2017. Google Compute Engine FAQ. https://cloud.google.com/compute/docs/faq. (2017). Accessed: 2017-09-22.

[26] Intel. 2017. Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf. (December 2017). Accessed: 2018-03-06.

[27] Intel. 2017. Trace Version APIs. https://software.intel.com/sites/landingpage/pintool/docs/81205/Pin/html/group__TRACE__VERSION__API.html. (February 2017). Accessed: 2017-10-26.

[28] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

[29] Greg Kroah-Hartman. 2018. Linux Kernel 4.4.115 Changelog. https://cdn.kernel.org/pub/linux/kernel/v4.x/ChangeLog-4.4.115. (February 2018). Accessed: 2018-02-23.

[30] Dirk Leinenbach and Thomas Santen. 2009. *Verifying the Microsoft Hyper-V Hypervisor with VCC*. Springer Berlin Heidelberg, Berlin, Heidelberg, 806–809.

[31] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Chicago, IL, 190–200.

[32] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 2012. Path-Exploration Lifting: Hi-Fi Tests for Lo-Fi Emulators. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. London, UK. https://doi.org/10.1145/2150976.2151012

[33] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2010. Testing System Virtual Machines. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*. Trento, Italy, 171–182. https://doi.org/10.1145/1831708.1831730

[34] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. Testing CPU Emulators. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. Chicago, IL, 261–272. https://doi.org/10.1145/1572272.1572303

[35] MivoCloud. 2017. KVM Cloud Hosting in Europe. https://www.mivocloud.com/. (2017). Accessed: 2017-09-22.

[36] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. *A Better x86 Memory Model: x86-TSO (Extended Version)*. Technical Report UCAM-CL-TR-745. University of Cambridge, Computer Laboratory.

[37] The Bochs Project. 2017. bochs: Bochs IA-32 Emulator Project. http://bochs.sourceforge.net/. (2017). Accessed: 2017-07-30.

[38] Florent Saudel and Jonathan Salwan. 2015. Triton: A Dynamic Symbolic Execution Framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*. SSTIC, 31–54.

[39] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome, and Anupam Datta. 2013. Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*. San Francisco, CA.

[40] Amit Vasudevan, Sagar Chaki, Petros Maniatis, Limin Jia, and Anupam Datta. 2016. überSpark: Enforcing Verifiable Object Abstractions for Automated Compositional Security Analysis of a Hypervisor. In *Proceedings of the 25th Usenix Security Symposium*. Austin, TX, 87–104.

[41] vServer Center. 2017. KVM Hosting | KVM Cloud Servers. https://www.vservercenter.com/kvm-hosting.html. (2017). Accessed: 2017-09-22.