Virtual CPU Validation

Nadav Amit[†] Dan Tsafrir[†] Assaf Schuster[†] Ahmad Ayoub[◇]
Eran Shlomo[◇]

[†]Technion – Israel Institute of Technology
{namit, dan, assaf}@cs.technion.ac.il

[◇]Intel Corporation
{ahmad.ayoub, eran.shlomo}@intel.com

Abstract

Testing the hypervisor is important for ensuring the correct operation and security of systems, but it is a hard and challenging task. We observe, however, that the challenge is similar in many respects to that of testing real CPUs. We thus propose to apply the testing environment of CPU vendors to hypervisors. We demonstrate the advantages of our proposal by adapting Intel's testing facility to the Linux KVM hypervisor. We uncover and fix 117 bugs, six of which are security vulnerabilities. We further find four flaws in Intel virtualization technology, causing a disparity between the observable behavior of code running on virtual and bare-metal servers.

1. Introduction

Since hardware-assisted virtualization was introduced to commodity x86 servers ten years ago, it has become the common practice for server deployment [7]. Today, about 75% of x86 server workloads run in virtual machines (VMs) [13]. Virtualization enables the consolidation of multiple VMs on a single server, thereby reducing hardware and operation costs [14]. Virtualization promises to reduce these costs without sacrificing robustness and security. We contend, however, that this promise is not fulfilled in practice, because hypervisors—the software layers that run VMs—are bug-prone. Hypervisor bugs can cause an operating system (OS) that runs within a VM to act incorrectly, crash, or become vulnerable to security exploits [18].

Hypervisor bugs are software bugs, but the damage they cause is similar to that of hardware bugs. Since hypervisors virtualize the hardware of VMs, their bugs cause the VMs to experience that the underlying hardware violates its specification. Patching hypervisor bugs is much easier than fixing the hardware, yet doing so may induce VM downtime and deter cloud customers, as indeed experienced by leading cloud providers [24, 71].

Several studies have addressed hypervisor bugs, but the proposed solutions are still inadequate. Existing formal verification techniques of hypervisors [3] are not full-fledged,

SOSP'15, October 4–7, 2015, Monterey, CA.
Copyright © 2015 ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

limited by the lack of formal hardware specifications [20] and the inability to validate important virtualization features such as interrupts. Code fuzzing approaches have been limited to testing instructions, suffer from a high rate of false-positives, and in general have not been used to thoroughly test hypervisors [42–44]. Avoiding hypervisor bugs is possible in principle by exposing to VMs a paravirtual hardware interface that is simpler and more easily verifiable than actual hardware [37]. But such an interface seems inapplicable for virtual CPUs (VCPUs), as it requires intrusive modifications to VM OSes.

Our work is based on the insight that hypervisor bugs resemble real hardware bugs in many cases, as they are triggered similarly and have similar consequences. We thus hypothesize that hardware validation tools would efficiently detect hypervisor bugs. We aspire to validate the most complicated virtual hardware component—the virtual CPU (VCPU).

We focus on x86 CPU virtualization, which requires hosts to be able to emulate multiple VCPU subsystems, notably (and perhaps counterintuitively) most x86 instructions. This requirement—like our proposed approach—holds regardless of whether hosting is software based [39], hardware assisted [35], and/or includes a dynamic binary translation component [1, 8].

We adapt Intel's tools for validating physical x86 CPUs [60] to test the KVM hypervisor [36], which is integrated in Linux and is used by cloud providers such as Google [28]. We find that the adaptation effort is reasonably low and that the result preserves the appealing features of the original tools: high coverage, reproducibility, and ease of debugging.

We use our testing infrastructure to study the number, severity, and cause of hypervisor bugs. We uncover 117 confirmed bugs that make VCPUs violate the CPU specifications. We fix most of the bugs—those that can be fixed—and commit the corresponding patches to KVM. We find that the severity of the majority of the bugs is moderate, but that a few (5%) introduce serious security vulnerabilities to the VMs or negatively affect their stability; for example, one bug existed in KVM for nearly five years, reportedly causing sporadic VM freezing [62]. We further find that most bugs (85%) are caused by implementors failing to strictly follow available CPU specifications, but that a few of these can nevertheless be attributed to missing or wrong documentation. Finally, we find four cases in which the CPU architecture is missing features, causing a disparity between the observable behavior of CPUs and VCPUs that cannot be fixed by software.

2. Virtualization Bugs

The hypervisor, which runs VMs, has the task of providing them with a virtual environment that behaves like real hardware. Yet building a fully-functional and bug-free hypervisor remains an arduous task despite CPU hardware assistance. To date, bug reports are continually being filed for the most popular hypervisors, and many of the reported bugs have existed in the code for a long time. Such bugs often cause the VM to fail (e.g., [11, 50, 51, 72]), or introduce security vulnerabilities (e.g., [18, 55, 71]).

CPU virtualization is probably the most important and difficult feature to implement correctly in hypervisors, as CPU architectures tend to be highly complicated. RISC CPUs are hard to virtualize [22], and even harder in the x86 architecture, whose instruction set consists of over 800 instructions, and which supports a variety of debug and performance monitoring facilities. To virtualize the CPU, the hypervisor traps and emulates dozens of types of events, and saves, restores and manipulates a VCPU state of over 100 registers.

CPU virtualization bugs can do a lot more harm than virtualization bugs in other hardware devices. While OSes are built to be robust and handle I/O device failures, they often cannot recover if the CPU, or VCPU in the case of VMs, does not conform to the specifications. Techniques for avoiding virtual device bugs, such as using a verifiable simplified paravirtual device [46], or disabling emulated devices after boot [52], are mostly inapplicable for CPU virtualization, as they require intrusive modification of VM OSes. Unlike device virtualization, CPU virtualization requires the use of privileged instructions that can only be executed in the kernel space. As a result, CPU virtualization bugs are more likely than others to become security vulnerabilities.

Uncovering hypervisor bugs—of which there are many— is a tedious job. Indeed, some bugs can easily be detected when a new OS or device driver is deployed (e.g., [50, 51, 72]), yet others are hard to reproduce (e.g., [67, 68]) or might even be ignored due to reproducibility difficulties (e.g., [34, 56]). The difficulty in validating hypervisors prevents new features from being employed in production systems. The most blatant example is *nested virtualization*, in which a hypervisor runs within a VM [9]. To this day this feature is still considered experimental [73], is unsupported [45], and suffers from a large number of bugs [66], despite the fact that it was introduced several years ago.

One of the greatest threats of hypervisor bugs is that they jeopardize VM security and isolation, which are actually the primary advantages of hardware virtualization over operating system-level virtualization [57]. In the worst case scenario, hypervisor bugs may allow code which runs within a VM to launch a privilege escalation attack on the hypervisor, and thereby run kernel-level operations on the virtualization host ([25, 29]).

Other attacks are also dangerous, especially in cloud environments, where VMs of different organizations are co-located, and an attacker can instantiate a VM that would be co-resident with a certain target VM [58]. Consequently, a malicious cloud user may be able to exploit hypervisor bugs to steal data from another VM, or to launch a denial of service (DoS) attack on a certain VM by crashing the host. In fact, even DoS attacks not directed at a certain VM may have other extreme effects. To achieve high-availability, virtualization platforms are usually configured to restart crashing VMs on another physical machine in the same resource pool [19, 71]. As a result, a single malicious VM that deploys a DoS attack can exhaust significant physical resources.

Even when security vulnerabilities are found before they are exploited, patching them in a timely manner without incurring VM downtime is not an easy task. In cloud environments, patching can introduce non-trivial bandwidth requirements, as it often requires that the running VMs be migrated to another physical server before the patch is applied [63]. Using direct attached storage for the VMs increases the bandwidth requirement even further and can render massive live migration impossible [23].

The damage caused by virtualization bugs may be best exemplified by the recent Xen security advisory, XSA-108, which reported a bug in the emulation of x2APIC machine state registers (MSRs) [18]. While real x86 CPUs hold up to 256 x2APIC MSRs, Xen erroneously emulated 1024, and served excessive MSR read operations from memory beyond the memory page that was used for emulating these MSRs. As a result, a malicious VM could issue MSR read operations that would either crash the entire host or read data from other VMs or the hypervisor. Although there were no reports of actual exploits of this bug, patching it required cloud vendors to invest substantial IT resources. More importantly, since many of the vendors could not perform live migration, patching was performed over a week, presumably because the patches were applied when the VMs were shut down. Eventually, cloud vendors, including Amazon Web Services, Rackspace and IBM SoftLayer, still needed to reboot many of the servers whose VMs were not shut down voluntarily. Amazon reported that 10% of the VMs were rebooted, and a survey revealed that some cloud users experienced a non-negligible downtime: over 18% of SoftLayer users reported a downtime of over an hour. Consequently, 29% of surveyed SoftLayer customers claimed to now be considering other providers [69].

3. Intel Virtualization Technology

Intel VT presents an instruction set that enables a hypervisor to run VCPUs of a VM in a less privileged mode called "guest mode." Code that runs in this mode can run both in kernel mode and in user mode, making it possible to run an entire OS within a VM. The VCPUs are controlled by the hypervisor, which runs in "root mode," and can trap VCPU sensitive instructions—instructions which may affect the entire system (e.g., writes to control registers)—as well as other sensitive events (e.g., interrupts). When such an event occurs, the CPU performs a "VM-exit," switching the CPU to "root mode" and running the hypervisor code. The hypervisor can then inspect the cause for the VM-exit and handle it, for instance by emulating the trapped instruction. Once the VM-exit is handled, the hypervisor can resume the execution of the VM in guest-mode by performing "VM-entry." The hypervisor configures which of the sensitive events should be trapped in a VM control structure (VMCS).

In addition to trapping sensitive events, the hypervisor can control how VM code is executed, without triggering a VM-exit. For example, Intel VT supports "second level address translation" (SLAT), from guest physical memory to host physical memory. The hypervisor sets SLAT paging structures according to the physical memory it allocated for the VM. When a CPU that runs in guest-mode issues memory accesses, the address goes through two levels of translation, first from guest virtual to guest physical, and then from guest physical to host physical.

When the CPU performs VM-entry and exit, it loads/stores certain portions of the VCPU state from/to the VMCS. However, part of the VCPU state, such as general purpose and floating-point unit (FPU) registers, are not kept in the VMCS. The architecture leaves it to the hypervisor to save and restore this state. To reduce exit handling time, hypervisors may decide not to save and restore some registers on every exit, if these registers have not been changed.

Sometimes the hypervisor also needs to handle certain infrequent events that are not virtualized by hardware, such as hardware "task-switch" or configuration of the interrupt controller (I/O APIC). Hypervisors can also report that the VCPU supports "nested virtualization" and emulate the execution of VT instructions in a VM, making it possible to run a hypervisor within the VM.

4. Testing

Our system validates that the behavior of the virtual CPU conforms with the specification of the physical CPU. It is therefore based on the methodology and tools used for physical CPU validation [60]. We next describe the existing Intel testing infrastructure for physical CPUs that we use in this study (§4.1) and the manner by which we adapt it to apply to virtual CPUs (§4.2).

4.1 Testing Physical CPUs

The physical CPU validation system consists of test generation, execution, and analysis, as depicted in Figure 1.

Generation Intel employs several test schemes to achieve good coverage. Some tests are focused on validating specific behaviors, like the CPU reset sequence; such tests are typically

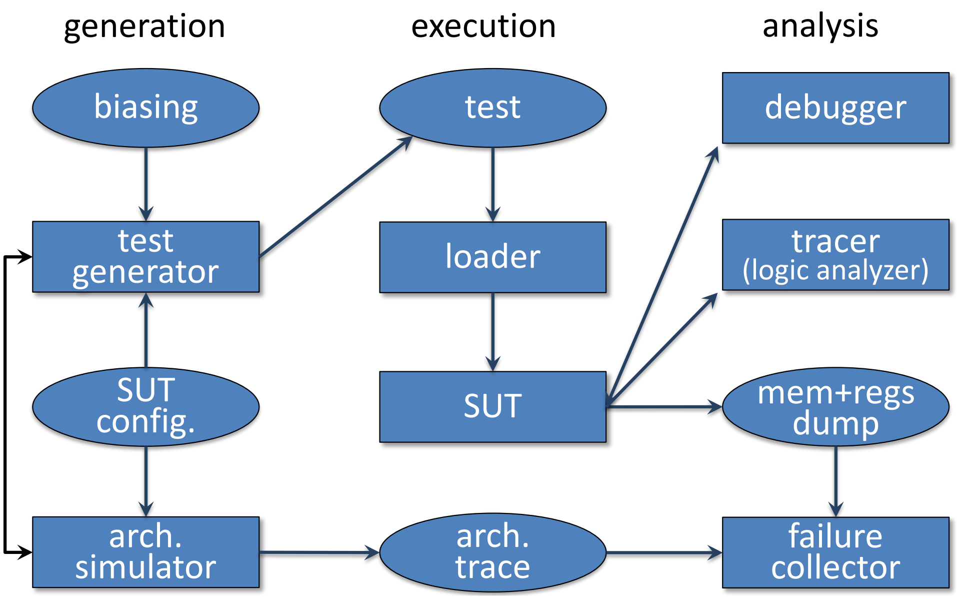

Figure 1. *Physical CPU validation system. Rectangles denote computing entities. Ellipses denote files.*

fixed or include a minor, carefully controlled random component. Most tests, however, are constructed via a random code generator, which utilizes code fuzzing and differential testing techniques. The generator creates comprehensive tests that exercise all the CPU subsystems. Unlike other code fuzzing tools, the generator is tightly coupled with an exact architectural simulator, used as a reference system.

Both generator and simulator use a system-under-test (SUT) configuration file that describes the SUT, specifying the physical memory map, the number of CPUs and their generation, the supported MSRs and their initial values, the supported instruction set extensions, and so on. The generator additionally uses a bias file that defines such parameters as the probability to generate individual instructions.

In Intel, there is no formal model that fully defines CPU behavior. Instead, an architectural simulator—reflecting the public and internal CPU specifications—is used as the reference system. The simulator is functional and unaware of the microarchitecture. As it is not cycle accurate, it operates at a reasonable speed and is thus usable for testing. By simulating execution of tests, the simulator provides their final, correct outcome for reference. The generator utilizes these outcomes to create *self-checking* tests. Each test is a memory image that is later loaded onto the (OS-less) SUT. The execution outcome then indicates whether the test passed or failed depending on whether it matched or mismatched the reference.

A test consists of three parts. The first is short initialization code that sets the basic environment, i.e., descriptor and page tables, model-specific registers (MSRs), and so forth; as this initial state is random, it helps to increase converge. The second part contains N random instructions that comply with the bias file. (Long random instruction sequences make debugging harder, whereas short sequences make initialization dominate the runtime; the test system typically uses $N = 4096$, striking a balance between the two.) The random sequences include every possible valid/invalid instruction, collectively exercising nearly all the architectural features. The third part of the test runs upon completion and reports the results.

Unlike typical fuzzing mechanisms, the generator is intimately aware of the semantics of the instruction set: (1) it employs instruction pre-handlers to, e.g., fulfill or purposely violate non-trivial preconditions (for instance, read MSR operations load the value of the ECX register, so an associated pre-handler can arrange things such that ECX would hold a valid MSR index if this is not the case already); (2) it employs instruction post-handlers to, e.g., eliminate nondeterminism (for instance, read timestamp counter instructions yield unknown results at generation time, so a handler can add subsequent instructions to overwrite these values); (3) it stresses bug-prone mechanisms such as memory aliasing and wraparound; (4) it ensures that random hardware breakpoints are meaningful by backpatching breakpoint-setting instructions to point to valid target instructions; (5) it prevents races between cores by tracking every byte used in the test (with the simulator's help); and (6) it avoids deadlocks and ensures completion by generating loops and branches in a carefully controlled way.

One feature that greatly enhances the generator's semantic awareness is that it works in tandem with the simulator on a per-instruction basis, simulating each instruction immediately after generation. It is thus capable of, for example, avoiding a host of undesirable situations, such as unintended exceptions that it was unable to foresee. The simulator allows the generator to roll back one instruction and try again.

Randomizing instructions individually as described above is highly effective. But some scenarios are too complex to be tested in this way. For example, inter-processor interrupts (IPIs) are hard to test, as they are inherently asynchronous and so the generator and simulator cannot tell which instructions will get hit by IPIs [10]. The generator therefore employs test "templates," which encode a recipe instructing it how to test. The IPIs template encoding includes: creating a fixed code chunk that synchronizes between the cores; generating a chunk of random code on the IPI target cores; and then emitting instructions that block the cores until the expected interrupts are received. Other templates are used, for example, when validating timer interrupts and cross-modifying code.

Execution Test generation is compute intensive and is significantly longer than test execution. Generation is therefore performed by multiple machines, outputting memory images that constitute self-checking tests. When a test is ready, it is immediately communicated via the network to the loader, whose role is to dispatch the test on the SUT, to retrieve indication whether the test has passed, and to forward the test to bug analysis if it has failed.

The loader and SUT are two distinct physical machines. The loader is connected to the SUT and controls its power switch. The SUT is equipped with a test device—denoted here as "Ldev"—which is likewise connected to the loader. Ldev gets the image from the loader, loads it into the SUT's memory, and generates an INIT event that starts the SUT working, running the newly arriving image. Ldev services the memory-mapped I/O (MMIO) and programmed I/O (PIO) issued by tests, such that MMIO/PIO read operations simply return the values previously written to the corresponding I/O addresses. Ldev is also used to generate external interrupts when their functionality is tested.

A running test indicates that it is finished by issuing an I/O write operation to Ldev via a predetermined I/O address. Ldev forwards the information to the loader, which in turn checks whether the test passed or failed. Sometimes, however, test failure causes the SUT to hang instead of exiting cleanly. The loader therefore sets a timer to bound the test execution time, and it proactively fails the test if the timer expires.

Analysis As noted, at the end of the execution, the SUT's state is compared against the reference outcome generated by the architectural simulator, which is encoded in the test. (The generator optimizes this procedure, using the simulator to identify the memory regions and registers that are affected by the test, such that only they will be compared.) Tests, however, may detect divergent executions sooner than their completion time, attempting to ease debugging and facilitate the root cause analysis. To this end, the generator incorporates within the test occasional partial comparisons of relevant memory locations and registers to their corresponding simulator-generated values. For example, tested exception handlers compare the SUT's registers and exception error code with the correct values output by the simulator.

Whether in the middle of the execution or at its end, a test fails when divergence is detected. The test then encodes in a suitable memory-resident structure information that accurately characterizes the divergence, to be retrieved later on by analysis tools seeking to identify the exact point where the divergence occurred. For instance, if a register value was found to be different than expected during an exception handler, the test records the register number, the actual and expected values, the bit-mask of the compared bits (not all bits are necessarily defined), and the faulting instruction pointer. Likewise, before synchronously waiting for interrupts, which might never arrive and thereby cause the test to hang, the test saves checkpoints in memory indicating the cause of the potential failure in case the test is timed out by the loader. The loader then dumps this memory structure into a file that is handed to the failure collector for later use.

When debugging a failing test, users utilize a remote x86 debugger that controls the SUT through an in-target probe (ITP) device [70] that the SUT houses. The ITP allows the debugger to, e.g., examine the internal CPU state, set breakpoints, and execute in single-step mode. Sometimes, however, a debugger affects the outcome of the test and interferes with the analysis. The system therefore supports non-intrusive tracing using logic analyzers that are connected to CPU pins, interconnects, or buses within the SUT. To further assist debugging, the architectural simulator can generate a detailed trace of the reference test execution, including, for example, all memory references and exception causes.

4.2 Testing Virtual CPUs

When adapting the physical CPU (PCPU) validation system to apply to VCPUs, we use the test generation subsystem unchanged. This approach is aligned with our insight that VCPUs should behave identically to PCPUs when subjected to the same tests. Thus, our adaptation efforts focus on enabling the execution and analysis of tests on VCPUs, as described next. (We report the analysis results in §5.)

System We test the Linux KVM/QEMU hypervisor (Linux versions 3.14–3.18 and QEMU version 2.1.0). QEMU is a regular user-level process that encapsulates a guest VM, such that each guest VCPU is in fact a thread within the QEMU process. KVM is a kernel module. KVM/QEMU employs hardware-assisted virtualization, which means that the VM code typically runs natively on the PCPU. Some VM operations, however, trap to, and are served by KVM/QEMU. For example, guest I/O requests directed at certain I/O devices are served by QEMU, which uses standard system calls to satisfy the requests, thereby emulating the functionality of these devices. In this setup, the SUT *is* the hypervisor.

We assign the role of the test loader to a Linux process that we implement, called "Vloader." Upon initialization, Vloader starts a QEMU instance that matches the SUT configuration file. The corresponding VM is diskless and has no OS. Vloader waits for the VM to boot (only BIOS), and then it starts getting requests from test generators as in the original PCPU validation system. Upon receiving a test memory image, Vloader communicates with the running QEMU instance via the QEMU Machine Protocol (QMP) [54], instructing it to: (1) suspend its preexisting OS-less VM, (2) load onto the VM the newly received memory image, and (3) send to the VM an INIT signal. Out of these three actions, QEMU only supports the first. We therefore implement the latter two.

Similarly to the original system, Vloader is also responsible for setting a timer and failing a test if it hangs, by communicating with QEMU. This action, however, can be accomplished using standard QMP operations.

Recall that the PCPU SUT is equipped with Ldev, a test device that handles I/O operations, such as the write operation that indicates that a test ended and whether it passed or failed. We add similar functionality in QEMU, by coupling our VM with a newly implemented emulated I/O device that has equivalent I/O semantics. We only implement PIO support, which is required for allowing tests to communicate their outcome.

Upon completion, the self-checking test compares the relevant memory regions to the reference produced by the architectural simulator. In the PCPU validation system, this comparison is performed (accelerated) using special hardware devices. But these devices are unaware of the additional memory indirection of virtualization in our VCPU validation system, so we modify QEMU to compare in software. If it discovers a difference, it dumps the memory-resident debug regions into a file. After the debug information is safely stored for later analysis, Vloader disposes of the running QEMU instance, spawning a new instance so as to prevent the failing test residues from causing additional failures.

Debug Support Being able to debug a failed test is important in both PCPU and VCPU validation systems—in the former, debugging uncovers PCPU bugs, whereas in the latter it uncovers hypervisor bugs. KVM allows for VM debugging with gdb. It supports single-stepping and hardware breakpoints by respectively manipulating the EFLAGS trap flag and the debug registers (DRs) of the VM. This approach is useful for debugging OSes of VMs. But it is unsuitable for our purposes, as it allows VMs to interfere with our debugging and our tests. For example, if a single-stepped VM invokes `PUSHF`, then the manipulated EFLAGS value becomes visible to the VM, which might affect the test's result. Likewise, when a VCPU encounters an exception, EFLAGS may change, thereby disabling single-stepping.

We overcome the problems associated with single-stepping by modifying KVM to utilize the Monitor Trap Flag (MTF) of Intel VT. MTF allows us to single-step VMs without changing their observable state or permitting them to occasionally disable single-stepping. We additionally use MTF to overcome the problems associated with breakpoints by modifying KVM to refrain from using the DRs of the VM altogether; instead, we opt for iteratively single-stepping the VM until reaching the instruction pointer associated with the breakpoint. This approach incurs a slowdown. But the slowdown is tolerable because tests consist of only a few thousands of instructions, and so the approach suffices.

Emulation Mode There are circumstances where hypervisors are required to trap and emulate VM instructions in software, even if some of these instructions are non-privileged and could have otherwise been executed directly on the PCPU (§5.1). Malicious guests can purposely create these circumstances and force emulation of arbitrary instructions, so it is important to get the emulation right. Our analysis, however, indicates that the relevant hypervisor subsystem is especially bug-prone due to the complexities involved in emulating x86 instructions while supporting multiple addressing modes, execution modes, operand sizes, etc. Consequently, to allow for thorough testing of this subsystem, we implement a

KVM mode that exercises this subsystem whenever possible, always preferring to emulate instructions whose emulation is supported instead of running them natively on the PCPU.

Under this mode, if emulation of an instruction is unsupported, we use MTF to resume the VM's native execution for one instruction only (single-stepping). Additionally, since KVM does not know how to emulate interrupt and exception events, we inject the interrupt/exception to the VM when it fires so as to make the VM aware, and then we likewise resume the VM for one instruction using MTF.

Nondeterminism The PCPU SUT is an OS-less, bare-metal machine dedicated to running tests. Conversely, the VCPU SUT operates within a general-purpose host environment, which runs services as kernel threads and allows uncontrolled interrupts. Such asynchronous activity might interfere with timing considerations of, e.g., IPI tests. We therefore curb asynchrony by following well-known practices to reduce OS jitter [47], including: off-lining and on-lining test cores to force workers away from them; setting interrupt affinity to core 0 and never using it for tests; utilizing a tick-less kernel; and pinning different VCPU threads to different PCPUs.

QEMU allocates memory for VMs on demand, and KVM populates extended page tables (EPT) on-demand [12, 31]. The hypervisor is thus a source of nondeterminism, as memory allocation in one test affects the execution of the subsequent test because the corresponding allocated/mapped pages persist between executions. We resolve this problem by instructing QEMU to always preallocate VM memory and by modifying KVM to always premap VM memory in the EPT.

Jitter and asynchrony did not affect our analysis, namely, *all* failing tests we found were reproducible.

Virtual I/O Paths Emulated I/O devices implement interfaces identical to those of physical devices, interacting with VMs in four ways: PIO, MMIO, DMA, and interrupts. The testing of I/O devices is out of scope. Still, CPU instructions and subsystems are in fact used to interact with I/O devices, and so execution correctness partially relies on certain hypervisor code paths. We contend that these paths are in scope, since they are generic and independent of specific devices.

Suppose, for example, that an instruction J traps since it involves MMIO. KVM decodes J, stages its operands, and learns that at least one of them resides in MMIO. If this operand should be obtained via input, KVM interacts with the device (possibly invoking QEMU), and it reissues J after the input becomes available. Among all of these activities, only the device interaction is specific; J's decoding, operand staging, and reissuing (among others) are generic.

The above corresponds to emulated I/O devices. Other generic hypervisor code paths are indirectly exercised with assigned I/O devices—instances of a physical devices that the hypervisor hands to VMs for their exclusive use, largely removing itself form the data path to boost performance. With this I/O model, relevant generic code paths include IOMMU programming and interrupt remapping and posting [32].

We did not test device assignment code paths, nor did we emulate device DMAs and external interrupts. For MMIO, we tested the initial part, namely, the activity before the device-specific interaction. The only I/O emulation mechanism we fully tested—notably, the reissuing of instructions that occurs after the device-specific interaction—is PIO.

Bootstrapping The effort to adapt the PCPU testing infrastructure along with finding bugs, fixing them, reporting them, and getting the associated patches committed took roughly a year. Running the first $N = 0$ test (where the number of random instructions comprising the

ID	*description*	*added*
K1	no #AC exceptions on emulated instructions	
K2	missing instructions emulation	
K3	atomic ops may be emulated as non-atomic	
K4	no emulation of port and data breakpoints	
K5	multiple nested virtualization bugs	
K6	missing emulation of machine state registers	✓
K7	no support for system management mode	
K8	no support for `MONITOR` and `MWAIT`	✓
K9	no support for performance monitor unit v3	✓
K10	different number of MTRRs than real CPUs	✓

Table 1. *Missing CPU virtualization support in KVM. We implemented K6,K8–K10 and disabled the use of the rest.*

middle part of the test is zero; see §4.1) took about two weeks. The test unsurprisingly failed, and making it pass took another month. Passing the first $N = 100$ test took approximately an additional month.

A main difficulty we faced while bootstrapping the system was that hypervisors may not virtualize certain CPU features, perhaps because they are viewed by the developers as unnecessary or as too hard to implement. In some cases, such missing support violates the CPU specification (entries K1–K7 in Table 1), whereas in other cases it conforms with the specifications since the (V)CPU is allowed to declare the missing support via capability registers (K8–K10). Regardless of whether the missing support is legitimate, our test generator relies on these features and fails to run if they are unsupported. We therefore implement the missing features in KVM if they are impactful, or entirely prevent their use with the bias file (§4.1) if they are not.

5. Results

We used the VCPU validation tools to test the KVM hypervisor. Our tests were executed on Intel's Skylake Client CPU. We ran over 100k tests, each containing 4096 random instructions per VCPU.

We now describe the bugs we encountered. We divide the bugs into different domains that correspond to the affected CPU features. For each domain we describe the architectural feature, the associated virtualization features, the bugs, and their potential impact. We then analyze the causes of the bugs and discuss the lessons we learned. Table 3 lists the bugs we found, a short description of each bug, and the patch number of its fix. We denote each bug as *Bx* as in Table 3.

5.1 Instruction Emulator

Ideally, the hypervisor would only need to emulate a small subset of the instruction set. However, on x86 architecture, the hypervisor may be required to emulate most instructions, for four reasons [6]:

Shadow Page Tables Prior to Nehalem micro-architecture, Intel CPUs did not support second level address translation. Hypervisors therefore employed page tables that held the translations of guest virtual memory addresses to *host* physical addresses. To efficiently synchronize them with the "shadow page tables" that the VCPU controlled, the hypervisor tracked changes of the shadow page tables, by trapping and emulating VM write accesses to them.

Real-Mode Prior to Westmere micro-architecture, Intel CPUs set restrictions on the guest-mode state, which prevented running real-mode code in guest-mode. Since CPUs boot in real-mode, hypervisors emulated the VCPU execution until it could run natively in guest-mode [16].

Port I/O (PIO) and Memory Mapped I/O (MMIO) Instructions that perform I/O operations using an emulated device are trapped by the hypervisor before they are executed. The hypervisor decodes and performs partial emulation of the instruction to recognize the accessed I/O space address. Using this information, the hypervisor then emulates the I/O device and completes the emulation of the trapped instruction. To avoid the overhead that VM-entries and exits incur, some hypervisors emulate entire code blocks that frequently perform I/O operations [1]. Note that the use of paravirtual devices does not obviate the need to emulate certain I/O devices, for example the programmable interval timer (PIT).

Vendor-Specific Instructions To allow migration of running VMs, hypervisors expose the "lowest common denominator" of physical server CPU features. In other words, hypervisors avoid exposing features not supported by all hosts that might run the VM. Hence, minor disparities between AMD and Intel CPUs may prevent the VM from using very useful instructions whose absence would degrade the VM's performance. To circumvent this limitation, the KVM hypervisor reports that the VCPU supports these instructions. KVM then traps illegal instruction exceptions that these instructions trigger and emulates them.

Since the x86 instruction set is relatively big, the instruction emulator is bug-prone. On modern CPUs, only a subset of the instructions would be emulated, yet as we show later (§6), the emulator can be tricked into emulating any instruction. We therefore use the emulator stress mode that tests the emulation of all instructions (§4.2). As we expected, the instruction emulator incorporates many bugs.

Some of these bugs are a serious threat as they pose security vulnerabilities. We discuss these vulnerabilities in §6. Other bugs may cause VM workloads to fail. For example, one of the bugs caused the decoder to miscalculate the instruction length when it was crossing a page boundary (B26). As a result, the emulator could mistakenly consider legal instructions as illegal, and deliver an exception. Although this bug was not previously reported, we believe it is likely to be experienced by KVM users.

During our tests, we encountered several bugs in the emulation of instructions commonly used for MMIO operations. The CPU flags could be updated incorrectly during the emulation of string scan and compare instructions (B37), compare-exchange instructions (B11), or when an instruction triggers an exception (B34). The emulation of some instructions used the wrong memory address (B7), and others the wrong operand size (B31, B32). Some bugs caused more subtle errors, for example the delivery of the wrong exception (B25) or the wrong error-code (B21, B52).

Some bugs are unlikely to occur on common VM workloads: assemblers might not generate machine code that triggers bugs (B2, B40, B67); OSes avoid invalid operations that lead to exceptions (B25); and the segmentation mechanism, whose emulation introduced many bugs, is not used by most OSes (B13, B14, B24, B25).

The causes for the bugs vary. Several are because of incorrect emulation of known x86 quirks [2] (B3, B4, B8, B36, B67, B75) or of lesser-known quirks (B46). Some are related to known architecture pitfalls such as wraparound (B54, B65). Some were introduced because of incorrect adaptation of the emulator to 64-bit (B7, B9, B22) or other new CPU features

(B35). Other bugs, for instance, NULL pointer dereferencing (B43, B45, B66), or wrong return value (B53), are caused by coding errors. One bug was caused due to a mistake in the CPU documentation (B63). Code redundancy caused some bugs to appear twice (e.g., B18 and B76).

Lessons Learned As we reviewed the bugs, we found that three of the instruction emulator bugs (B20, B26, B48) are software regressions, i.e., bugs caused by enhancements or other bug fixes. Two additional software regressions were found in other virtualization mechanisms (B89, B102). This result indicates that hypervisor testing is not a one-time effort, and should be performed on a regular basis.

Our analysis indicated that disabling the emulation of vendor-specific instructions could prevent one of the bugs (B20) and mitigate others (B27, B46, B47, B64). Since this emulation is necessary only in certain environments, we suggest it be disabled by default. Since we discovered other bugs caused by rarely used hypervisor features (§5.2), we generalize our suggestion: every optional hypervisor feature should be disabled unless it is explicitly required.

Since over half of the bugs we found are instruction emulator bugs, it appears that hypervisor developers would benefit from hardware enhancements that would simplify their software implementation. While eliminating the need for an instruction emulator is complicated due to the complex format of x86 instructions and the fact that they can access multiple memory locations, other partial solutions are possible.

For example, Intel VT includes a "decode assist" feature, which provides information about instructions that trigger VM-exits, thereby eliminating the need to decode these instructions in software. However, this feature does not provide decode information on most VM-exits, and therefore it cannot be used by the instruction emulator. We suggest enhancing the decode assist to provide information about *every* VM-exit triggered by a VM instruction and to provide more data about it, for instance, its operand values. Providing this information can eliminate hypervisor bugs as well as some of the security vulnerabilities we present in §6.

5.2 Debug Facilities

Each x86 CPU has four debug registers in which the OS can set linear addresses of desired breakpoints or watchpoints. A control debug register is used to activate the breakpoints and set their type. When the condition of a breakpoint is met, the processor invokes the debug exception handler and updates the debug status register to indicate the exception cause.

In our experiments we found eleven bugs in the way KVM virtualized the debug facilities: mishandling the architectural "Resume Flag" triggered multiple debug events on a single breakpoint (B77, B78) or caused breakpoints to be skipped (B76); transactional memory debug could not be enabled (B79); execution of the `ICEBP` instruction did not advance the instruction pointer (B75; it was fixed by others); and breakpoints did not correctly update the debug status and control registers (B80, B82, B83). These bugs can cause VM debuggers to fail, and since debug exceptions are handled in OS code, they may even cause the VM OS to panic.

Lessons Learned Discovering bugs in the virtualization of CPU debug facilities may be surprising, as Intel VT makes it possible for hypervisors to virtualize debug facilities correctly without trapping any debug exception and debug register access. KVM, however, trapped these events to allow a host level debugger to debug the VM. KVM performs this debugging by setting hardware breakpoints of the host debugger in debug registers that the VM does not use. To hide this manipulation, the hypervisor then traps VM access to the debug registers and emulates them. When a debug exception occurs, KVM traps it and determines whether it was triggered by a breakpoint of the host debugger, or whether it should be delivered back to the VM. Handling hardware breakpoints in this manner is obviously bug-prone.

One way to mitigate the impact of such bugs is not to trap debug register accesses and exceptions when a host debugger is not used. Indeed, recent KVM changes eliminated many of these traps. A better way is to enhance the virtualization architecture to allow the hypervisor to debug VMs without trapping the VM debug register accesses and exceptions.

Finally, we note that like debug registers, other CPU resources can be used by both the hypervisor and the VM. For instance, performance counters can be used by the hypervisor to monitor the VM [15] while the VM OS uses them for its own purposes. Supporting this use-case requires extensive software support, and intrusive intervention of the hypervisor in the VM run. As we experienced, such intervention can result in bugs as well (B111). Hypervisor robustness would therefore benefit from CPU features that would allow provisioning of all CPU resources without software intervention.

5.3 Local APIC

Each x86 core holds a "local advanced programmable interrupt controller" (LAPIC), which receives external and internal interrupts (e.g., timer-interrupts), and sends and receives inter-processor interrupts (IPIs). The LAPIC is a feature-rich component whose efficient emulation is difficult. New server CPUs can virtualize some of its behavior in hardware.

In our tests we encountered seven bugs in LAPIC emulation. Some of the bugs we found are rather disturbing as we believe they occur on common VM workloads.

The most disturbing bug that we found reportedly caused certain VMs to sporadically freeze. Due to this bug, an interrupt may not be delivered to a VCPU unless an additional interrupt is later sent to the same VCPU (B89). This bug occurs due to improper synchronization in a highly optimized lock-free code, which leads to a non-trivial race. Fixing this bug, which existed in KVM code for nearly five years, resolved a reported issue of occasional freeze of VMs [62]. Due to the complex nature of this bug, and since it occurred roughly once a month and only on certain systems, substantial efforts to debug it on real OSes were futile.

Two additional bugs that we found occur when a VM OS uses the APIC timer "time-stamp counter deadline" operation mode, similarly to the way Linux uses it. A timer set to this mode should deliver a single interrupt at a given absolute time. Yet we found that KVM often injects a spurious interrupt to the VM after the timer has elapsed and an interrupt has already been delivered to the VM (B87, B88). Apparently, OSes are robust enough not to crash despite this spurious interrupt. Other bugs that we found could lead to spurious or missing interrupts (B85, B86, B90) or render the APIC useless (B84).

Lessons Learned Debugging the LAPIC was more complicated than other mechanisms, as two of the bugs (B86, B89) were caused by non-trivial races. Nonetheless, by running fewer than 1k iterations of these tests, we were able to recreate the failures and find their root causes in a few hours.

5.4 Model-Specific Registers

The x86 architecture includes model-specific registers (MSRs) for controlling CPU functions and for state monitoring. MSRs can be read and written using the privileged `RDMSR` and `WRMSR` instructions. To virtualize MSRs, Intel VT uses MSR bitmaps that allow the hypervisor to configure which MSRs can be accessed directly by the VM, and which are handled by the hypervisor and therefore trigger a VM-exit when they are accessed.

Our tests—although they were not intended to stress MSR accesses—found eight bugs in MSR handling. First, writing invalid values to MSRs or accessing non-existent MSRs should cause an exception (#GP). We found that KVM did not emulate this behavior correctly (B91, B92, B96). In one case (B92), KVM emulated `WRMSR` by writing the MSR value to the real CPU without checking that the value is valid, and therefore could cause the host kernel to panic.

Second, we found cases where KVM erroneously emulated MSR writes without reflecting MSR values on the real CPU. As a result, certain features with observable implications were not enabled or disabled, resulting in wrong VCPU behavior (B93, B94, B97). Last, we found that invalid `WRMSR` instructions in real-mode caused VM-entry to fail, and the VM to crash (B95), since KVM mistakenly delivered an error-code for real-mode exceptions.

Lessons Learned Reviewing these bugs, we find they are all byproducts of the complexity of MSR architecture. Some MSRs affect the visible architectural behavior of the VM, whereas others do not. Deducing this information from the specification is not an easy task. It appears such bugs could easily have been avoided by documenting this information.

The VM BIOS might actually prevent the occurrence of some bugs, but it cannot be relied on to always prevent them. Bugs that occur upon MSR configuration that is only carried out by the BIOS might never be triggered; by default, hypervisors power-on VMs with a certain BIOS, which may not trigger them. However, some hypervisors make it possible to use a different BIOS or Unified Extensible Firmware Interface (UEFI) [26], which may trigger these bugs. Running a nested hypervisor that runs nested VMs with different BIOS may trigger these bugs as well.

But the VM BIOS can also introduce bugs of its own. In our research, as we addressed one of the disparities in the behavior of VCPUs and CPUs (K10), we unintentionally triggered a bug in the VM BIOS that caused the 32-bit version of Windows 7 to display the so-called blue screen of death [4]. The fact that we hit a VM BIOS bug suggests that more thorough VM BIOS testing is required, especially since such bugs may compromise system security [17].

5.5 Task-Switch

OSes often switch tasks, saving the current task state in memory and loading that of the next one. To facilitate this in software, Intel introduced the hardware "task-switch" mechanism 30 years ago. However, this rather complex mechanism never gained significant traction, reportedly because it was slow and not portable. Due to its infrequent use and complexity, Intel does not support the native execution of task-switch in guest-mode and AMD's support lacks important features [59]. As a result, hypervisors are required to emulate task-switch and cope with its complexity. Despite its unpopularity, task-switch remains in use in most 32-bit OSes, since it provides atomic context switching upon serious errors. 32-bit Linux, for instance, uses task-switch when it encounters a "double-fault," which is caused by unexpected exceptions.

In our tests we encountered five bugs in task-switch emulation. In two cases valid task-switch operations could fail due to incorrect privilege checks (B101, B102). The latter (B102) was introduced during our research by another KVM developer, and we therefore believe that it could indeed harm common workloads. In addition, we found that task-switch emulation mistakenly saved registers (B100), and erroneously masked hardware breakpoints (B99, B103).

Lessons Learned Bugs in task-switch emulation are expected due to its complexity and infrequent use. Nonetheless, these bugs are harmful since task-switch is used when the OS encounters an error. In such cases these bugs may prevent the VM OS from performing graceful shutdown.

5.6 Initialization

x86 CPUs support two initialization events: reset and INIT. The CPU responds to these events by initializing the CPU state to a fixed predefined state. The two events are similar but different as INIT leaves part of the CPU state unchanged. OSes commonly use INIT IPIs to enable the bootstrap processor to wake up the other processors [31].

Intel VT does not virtualize these initialization events and requires the hypervisor to emulate them. Although our tests were not intended to test CPU initialization, they revealed four bugs. As we described in §4.2, each test is invoked by injecting an INIT event to the VCPUs.

Two bugs were revealed directly by the tests. One caused pending exceptions and interrupts to be delivered after INIT (B105). This bug was discovered accidentally, as we initially ran the tests without restarting the VM after each test failure, and interrupts from one failing test were received on the following test, causing it to fail too. The second bug resulted in unexpected interrupts due to improper initialization of LAPIC during RESET (B109). This incorrect behavior was actually a workaround to circumvent an old bug in the VM BIOS used by QEMU. Although the BIOS bug was resolved long ago, the workaround was not removed.

Two additional bugs did not cause tests to fail, but were apparent when we examined execution traces during debugging, as they prevented the hypervisor from changing the bootstrap processor (B106, B104). The latter bug had additional implications as it cleared part of the CPU state that should remain intact during INIT. Motivated by these bugs, we created unit-tests to test the reset sequence and found that KVM does not initialize some registers during it (B107, B108).

Lessons Learned Again, we see that the VM BIOS and the OS initialization code may hide certain bugs. However, initialization bugs may become apparent when BIOS implementation or OS code change. The recent development of the OVMF project, which delivers UEFI support for VMs [26], revealed additional bugs in the initialization code.

Arguably, hypervisors should use non-buggy BIOSes instead of circumventing these bugs. However, in practice it is not always feasible, as the VM BIOS may be developed as a separate project. In KVM removing the code that circumvented BIOS bugs turned out to be complicated, as KVM merely provides an API for virtualizing VCPUs, and may therefore be used with old and buggy BIOS implementations. To fix KVM bugs without causing legacy software to fail, we extended KVM API so it would allow the turning off of quirks that were used to circumvent legacy BIOS bugs.

5.7 Bug Summary and Discussion

We are encouraged by the quantity and severity of the bugs exposed during the validation process. Running the tests provided several insights.

Debugging Time Debugging each bug and analyzing its root cause took between a few minutes to a day, and on average two hours. In general, instruction emulator bugs triggered only by a certain instruction were the easiest to debug. Bugs caused by races, missing documentation, or those that were affected by the debugging process were significantly harder to debug. The hardest bug to debug was certainly B89, which occurred due to complex race conditions.

Execution Time The generation of each test takes on average five seconds, and running it on the VM less than a second. To saturate the host, multiple generators can be used, and each test can be executed multiple times. The size of each test image is 1MB on average and copying the tests from the generator to the host can take negligible time.

Code Review Whenever we encountered a bug, we reviewed both the related code (e.g., the faulty function) and the code that deals with related architectural features. Soon after, we released code patches for fixing the bug, and these patches were then reviewed by the KVM community. Although eight of the bugs were found in internal or external reviews, the reviews often missed similar bugs that were later hit by the random test generator. For example, three of the bugs (B99, B103 and B83) occurred practically on the same line of code, yet the reviews missed the latter two bugs. Code review is therefore essential but insufficient for hypervisor validation.

Bug Causes Hypervisor bugs can be attributed to two main causes: not following the hardware specifications, and coding errors. While the vast majority of the bugs (85%) were caused by non-conformance to CPU specifications, they were less severe, as they only jeopardized VM security and stability. Some coding errors, however, caused the host to panic (§6) and others could degrade the VM performance (B111, B112). In the long run, we expect that most bugs would be caused by coding errors once hypervisors implement and fix CPU emulation features. Indeed, four out of the five software regression bugs we encountered were due to coding mistakes (B26, B48, B89, B101).

CPU Specification Intel x86 CPU specifications consist of over 3000 pages due to the high complexity of the architecture. We were not surprised to find that some bugs resulted from documentation errors. In one case the CPU behavior was undocumented (B69), in other cases it is undocumented but publicly known (B71, B75, B103), and in another the documentation was recently fixed (B70). In some cases the behavior is documented but unclear (B68, B102, B113).

False Positive During the initial stages of our validation effort we encountered several false indications of bugs. These indications were caused by the adaptation of the test environment to VCPUs and by missing KVM features (Table 1). Afterwards, we encountered a single false-positive failure in a test that exercised an internal and undocumented CPU feature. Excluding these false indications, we encountered a few test failures that were caused by incorrect emulation of undocumented CPU features. Some may question whether such bugs are real ones, yet since software tends to rely on undocumented yet consistent hardware behavior, we do not consider these cases as false positive indications.

False Negative To check whether the validation tool missed bugs, we reviewed KVM bug reports and patches that were sent to the stable 3.18 Linux branch. Our review found no bugs that the validation tool should have hit. Nonetheless, our testing was limited as we used a desktop CPU and have not completed the enabling of the test devices. As a result, we did not hit a bug in the way KVM handles machine-check exceptions. Using a coverage tool [27] we checked which KVM code is exercised by our tests. We found that the tests do not cover some cases, for example, 16-bit task-switch.

Remaining Bugs As we concluded our project, we hit no more failures in over 50k tests that ran in the regular KVM execution mode. We believe that running more tests in this mode on our testbed may find a few more bugs. We assume that enabling the test devices, using other CPUs, and exercising hypervisor control features (e.g., VM save/restore) would uncover many more bugs. In contrast to the regular execution mode, using the "emulation mode," which stresses the instruction emulator, continually reveals more bugs. Fixing some of these bugs requires significant changes in the way the instruction emulator operates.

5.8 CPU Architecture Flaws

We encountered in our project four test failures that were caused by discrepancies between CPU and VCPU behavior, but cannot be resolved by changing hypervisor software. These failures occurred since the CPU architecture violates in certain cases the VM properties as defined by Popek and Goldberg [53]: hypervisors can either make VCPU execution *equivalent* to real CPU execution[1] or run most instructions *efficiently* by executing them directly on the CPU, but it cannot do both. We attribute these limitations to CPU architecture flaws. Although these flaws are likely to have limited impact, they were previously unknown and we therefore describe them in detail. We categorize the failure causes into three groups.

Non-Virtualizable State Intel VT does not virtualize the *physical address-width*, which determines the size of the physical addresses produced by paging [31]. This width implicitly defines the number of reserved bits in PTEs, as address bits above the width are reserved. A page-walk that uses a PTE whose reserved bits are set triggers a page-fault. Software can obtain this width using the `CPUID` instruction.

We find that hypervisors are incapable of setting a suitable physical address-width for VCPUs [5]. Since the VM may be migrated between servers with different physical address width, hypervisors set a predefined fixed width that fits all servers. As we tried to generate tests that match the physical address-width that KVM reports, we encountered test failures, and could not solve them by changing the hypervisor: if the reported value is lower than the actual one, a page fault error-code can incorrectly indicate, from the VM point of view, that the reserved-bits are cleared; but if the reported value is higher, the VM may map device RAM to unsupported physical memory, thereby triggering an exit whenever the VCPU accesses this memory.

Missing State Save/Restore Facilities When the hypervisor performs VM-entry and exit, it needs to restore and save the VM state correspondingly. Intel VT saves and restores some of the VCPU registers atomically during VM-entry and exit, and the hypervisor saves and restores others in software. Our experiments indicated that the registers are restored incorrectly in two cases.

The first case occurs when a VM executes an `FSAVE` instruction, which stores the floating point unit (FPU) registers in memory. This state includes the last floating point instruction pointer (FIP). In real-mode this value is calculated using two internal registers:
$[FSAVE\ \ FIP] = [Internal\ \ FCS] \times 4 + [Internal\ \ FIP]$.
Hypervisors, however, cannot save the internal FCS register, as its saving was deprecated in new CPUs. As a result, after the hypervisor saves and restores the FPU registers of a VM, execution of an `FSAVE` instruction in the VM may store an invalid FIP [5]. A similar issue was reported before to cause the blue screen of death in certain Windows environments [11] and was resolved with the recent deprecation of FCS storing. Yet, our findings show that refraining from saving the FCS had undesired side-effects.

The second case occurs when the hypervisor uses the `XSAVES` instruction to save the VM extended-state registers. `XSAVES` does not save the *raw* value of a certain bit, (XINUSE[1]),

[1] Excluding increased latency and reduced physical resources.

ID	*MITRE CVE ID*	*requires privilege*	*potential attack*	*bug ID*
C1	2014-3610	✓	host DoS	B92
C2	2014-3647	×	guest DoS	B42
C3	2014-7842	×	guest DoS	B23
C4	2014-8480	✓	host DoS	B43
C5	2014-8481	×	host DoS	B45
C6	2015-0239	×	privilege escalation	B64

Table 2. *Security vulnerabilities.*

that serves to indicate whether a group of CPU registers (XMM) are zeroed. Our findings show that as a result the VCPU may save and restore these registers unnecessarily even when they are zeroed. The hypervisor can work around this issue using the `XSAVE` instruction instead.

Errata While errata are undesirable regardless of virtualization, they can also break the equivalence property of VMs. This may occur when a VM executes the `ENTER` instruction, which creates a stack frame for a procedure by copying stack frame pointers from an old stack into a new one. Due to a public erratum, copy operations may be carried out even if the instruction execution encounters a page-fault [33]. This behavior is observable when the source and destination memory overlap, thereby causing re-execution of the instruction to read the partial copy results from the source memory, instead of the original data. Our results indicate that page-faults in the second level address translation page tables trigger the erratum as well. As a result, VMs would experience it even when no page-fault—from the point of view of the VM—occurs.

It should be emphasized that in all of the cases we described, the physical CPU behavior follows public documentation. Nonetheless, it results in differences between VCPUs and CPUs that allow the VM to detect hypervisor presence (red-pilling), a technique which malware uses to avoid analysis.

5.9 Other Hypervisors

Using our methods for validating every hypervisor is possible, but requires some effort, as described in §4. Validation of proprietary hypervisors is more challenging, yet requires modifying the way the test is loaded and results are communicated to the tester, as well as only using assigned test devices. In addition, debugging and finding the root cause of bugs on proprietary hypervisors is a cumbersome task.

Although we have not run our VCPU validation system to test other hypervisors, we see no reason it cannot be done, and initial indications are that doing so would reveal bugs in these hypervisors as well. The Xen hypervisor developers tracked our bug reports and found several similar bugs in their system (B42, B44, B23), as well as a similar security exploit (XSA-110). To obtain some indication whether VMware suffers from similar bugs, we ran the KVM unit-test suite on VMware Workstation 10. This suite incorporates tests originally created to validate KVM bug fixes, as well as additional tests we introduced to validate several fixes of the KVM bugs we encountered. Running these tests revealed that VMware Workstation 10 suffers from one of the bugs (B94).

6. Security

We found that the bugs introduce six security vulnerabilities, listed in Table 2, along with their MITRE CVE ID, whether they can be exploited by an unprivileged userspace code, the potential attacks, and the bugs that caused them. We denote these vulnerabilities by *Cx*.

As shown, some bugs can cause the VM to crash since KVM shuts it down due to missing emulation support (C3), or because it corrupts the VCPU state and causes VM-entry to fail the CPU consistency tests (C2). These bugs can be used to launch a DoS attack on the VM. Other bugs cause the host to panic (C1, C4, C5), and can therefore be used to launch a DoS attack on the host. One of the bugs can corrupt the VCPU state in a way that allows unprivileged userspace code to gain VM kernel-space privileges. This bug can be used to launch a privilege escalation attack on the VM (C6).

Four of the vulnerabilities we found can be exploited by an unprivileged VM userspace code. This may be surprising, as hypervisors usually trap only a few events that occur in the VM userspace. However, unprivileged userspace applications can trigger bugs in the most bug-prone hypervisor component: the instruction emulator.

Instruction emulator bugs are exploitable by crafted instructions that access MMIO regions. Although most MMIO regions are accessible only to the OS, they are sometimes also accessible from userspace. In Linux, to speed up the execution of the `gettimeofday` system-call, the OS grants processes a read access to the high precision timer (HPET) MMIO region, which is emulated by the hypervisor. In addition, some systems assign devices to userspace processes [21]. As a result, a malicious VM process can cause the hypervisor to emulate most of the instructions, triggering most of the bugs, and thereby exploiting most of the instruction emulator vulnerabilities (C3, C4, C5).

Yet certain instruction emulator bugs occur during the emulation of instructions that do not have memory operands, or occur only when these operands have specific values. A malicious VM process can still exploit these bugs in VMs that consist of multiple VCPUs that run concurrently on multiple physical CPUs, by employing cross-modifying code: the VM triggers an exit on an instruction that accesses an MMIO region, and replaces it with another instruction before the hypervisor decodes it. Using this technique, a malicious process can trigger every instruction emulator bug.

The flow of such an attack (C6) is depicted in Figure 2. The malicious application creates two threads, thereby causing the VM OS to schedule each one on a different VCPU. The first thread, which runs on VCPU0, writes a benign instruction that accesses MMIO, `MOV` from the HPET in our example (1). This thread then executes the instruction (2), which causes VCPU0 to exit to the hypervisor (3). Then, VCPU1 overwrites the instruction that VCPU0 executed with the bug triggering instruction, `SYSENTER` in our example (4). Since it is hard for VCPU1 to hit the exact point in time in which the exit occurs, it alternately switches between the `MOV` and `SYSENTER` instructions. The hypervisor then tries to emulate the instruction that triggered the exit, but fetches the modified instruction instead (5). Once emulated (6), this instruction triggers the bug.

As long as hypervisor code can introduce bugs, hypervisor security may be compromised. Obviously, removing code in the hypervisor can improve security, but doing so can sacrifice hypervisor functionality and optimizations. To improve security without such sacrifice, others have suggested preserving the previous VM state after failure [40], reducing the trusted computing base of hypervisors [30, 64], and avoiding assertions that may crash the host. Our research leads us to suggest two additional security improvements.

One way to improve security is for hypervisors to avoid killing VMs, even if they cannot handle a VM trap or cannot reenter guest-mode, since such errors can be triggered by

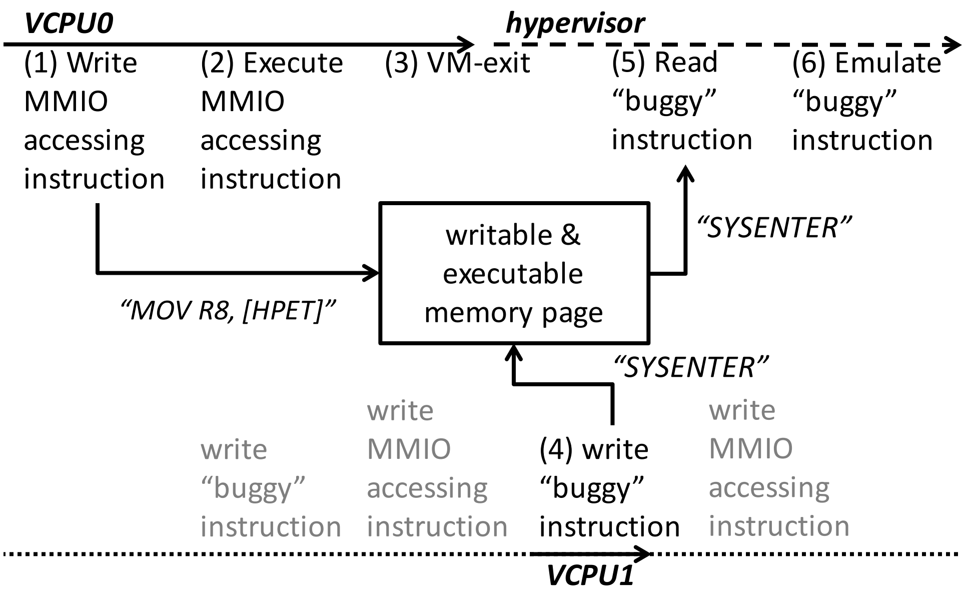
VCPU0
hypervisor
(1) Write MMIO accessing instruction
(2) Execute MMIO accessing instruction
(3) VM-exit
(5) Read "buggy" instruction
(6) Emulate "buggy" instruction
writable & executable memory page
"MOV R8, [HPET]"
"SYSENTER"
"SYSENTER"
write "buggy" instruction
write MMIO accessing instruction
(4) write "buggy" instruction
write MMIO accessing instruction
VCPU1

Figure 2. *Exploiting an instruction emulator bug.*

malicious VM processes. The hypervisor should gracefully recover, fix corruptions in the VCPU state and deliver an error indication to the VM. The architecture can define a new architectural event to indicate that the hypervisor encountered a recoverable error. OSes would be able to react to this event according to their policy, e.g., by killing the running process. Although this behavior allows a VM process to detect hypervisor presence, it is more secure since it prevents malicious processes from crashing the entire VM. We implemented a partial solution in KVM (B23).

7. Related Work

The challenge of developing a secure and correct hypervisor has been addressed using various approaches.

Formal Verification Arguably, the most compelling approach is using formal verification to prove hypervisor code correctness. The most ambitious attempt for such verification has been the Verisoft-XT project, which verified the Microsoft Hyper-V [41]. The effort of verifying this hypervisor required 60 person-years. Despite this effort, verification is incomplete, as fewer than 200 instructions were modeled, and the verification of several basic CPU features such as interrupts is partial at best [20]. During the project, "less than a handful (bugs) have been found" [41], which the project developers attributed to the low defect density of the hypervisor. However, despite the relatively small number of Hyper-V bug reports, some bugs, which one may expect verification would eliminate, postdate the project [48, 49, 71].

Micro-Hypervisors Micro-hypervisors can deliver better security because they have a smaller attack surface. For instance, the NOVA hypervisor [64] consists of fewer than 10k lines of code (LoC) in its trusted computing base (TCB). The reduction of the TCB indeed reduces the risk of comprising the *host* security. Yet micro-hypervisors still need to carry out the same virtualization tasks as other hypervisors. Although micro-hypervisors run these tasks in userspace, they are just as susceptible to software bugs that can compromise the *VM* security and robustness. To eliminate at least some of these bugs, there should be as little interaction as possible between the VM and the hypervisor. Unfortunately, this also results in reduced usability. For example, the NoHype [65] and Jailhouse [61] micro-hypervisors cannot perform dynamic provisioning of CPUs and memory.

Fuzzing Hypervisors can be validated by code fuzzing and differential testing—generating a random test, running it on both a VM and bare-metal system, and comparing the results. Recent work includes PokeEMU [42], which creates test cases based on high fidelity emulator implementation. PokeEMU uses symbolic execution to explore code-paths in a CPU emulator and thereby infers the CPU instruction set and the machine state that affects each instruction. Using this information, PokeEMU creates test cases, each of which exercises a single instruction under certain conditions. The test cases can then be executed on an instruction emulator and a real CPU to find discrepancies.

In contrast to PokeEMU, our system exercises multiple random instructions and test templates on each test, and consequently can reveal additional bugs. We found, for example, bugs that are only apparent when running multiple random instructions (B48, B111), bugs in interrupt delivery (B88, B90), and bugs that only occur on multi-core VMs (B85, B106).

To reduce testing time PokeEMU makes assumptions that allows it to test fewer cases, yet limit its coverage. First, it exercises only a single byte sequence of each instruction, under the assumption that the incremental benefit from more testing is relatively low. However, our results indicate that several bugs occur only when an instruction is emulated with a certain prefix, addressing mode or operands (B6, B73). Second, PokeEMU generates an initial machine state that exercises only exercises previously unexplored code-paths, and therefore may not find computation bugs (B54, B65).

Unlike vendor validation tools, PokeEMU delivers a high rate of false-positive indications of bugs. Roughly 10% of PokeEMU test-cases that the QEMU emulator ran failed, undefined CPU behavior was identified as the cause for a substantial number of failures. Our system suffered a negligible false-positive rate, since the architectural simulator indicates which instruction results are undefined, and the test generator prevents nondeterministic results due to interrupts or errata.

8. Future Work

One of the main challenges is the validation of host events that affect the VM execution, for example live migration of a VM from one physical host to another or paging of VM memory. Since such events often involve I/O operations, testing them is a prolonged process, and test failures are thus not likely to be reproducible. Although testing is possible using execution replay mechanisms [38], doing so can prevent existing bugs from being triggered, because it adds synchronization events.

Additional effort is also required for the validation of bug-prone subsystems of hypervisors. While CPU validation tools use test templates that stress bug-prone CPU subsystems, hypervisors may have additional weak spots. For instance, hypervisors often erroneously virtualize the time-stamp counter. Test templates for the validation of such features should therefore be incorporated into the CPU validation tools.

Finally, CPU validation usually focuses on changes in the and macro- or micro-architecture and assume the executed code is reasonable. This strategy is based on the assumption that bugs in legacy features or bugs that are triggered by senseless code do not occur in common OSes and therefore, if such a bug is uncovered, the CPU vendor can publish an erratum and guide software developers to avoid it. Nonetheless, virtualization raises the question whether this scheme is reasonable. Future research may evaluate the implications of existing CPU errata on virtualized environments and whether they pose a security threat.

9. Conclusions

Hardware-assisted virtualization is popular, arguably allowing users to run multiple workloads robustly and securely while incurring low performance overheads. But the robustness and security are not to be taken for granted, as it is challenging to virtualize the CPU correctly, notably in the face of newly added features and use cases. CPU vendors invest a lot of effort—hundreds of person years or more—to develop validation tools, and they exclusively enjoy the benefit of having an accurate reference system. We therefore speculate that effective hypervisor validation could truly be made possible only with their help. We further contend that it is in their interest to provide such help, as the majority of server workloads already run on virtual hardware, and this trend is expected to continue. We hope that open source hypervisors will be validated on a regular basis by Intel Open Source Technology Center.

Acknowledgments

We thank Paolo Bonzini from Redhat and our shepherd Andrew Baumann. The research leading to the results presented in this paper was partially supported by: the Israel Science Foundation (grant No. 605/12); the Ministry of Science and Technology, Israel (grant No. 3-9779); and the Google Patch Reward Program.

id	*description*	*patch*
	Instruction Emulator	
B1	Compatibility mode recognized incorrectly	42bf549f3c67
B2	Rep-IN uses memory operand	e6e39f0438bc
B3	NOP emulation clears RAX[63:32]	a825f5cc4a84
B4	CMOV DWORD does not clear [63:32]	140bad89fd25
B5	Outer-privilege level RET unsupported	9e8919ae793f
B6	Wrong emulation of 'XADD Rx,Rx'	ee212297cd42
B7	Bit-ops emulation ignores offset on 64-bit	7dec5603b6b8
B8	SMSW emulation is incorrect in 64-bit	32e94d0696c2
B9	CMPXCHG16B emulated as 8B	aaa05f2437b9
B10	RDPMC checks the counter incorrectly	67f4d4288c35
B11	CMPXCHG emulation sets incorrect flags	37c564f2854b
B12	SGDT/SIDT with CPL=3 causes #GP	606b1c3e8759
B13	Loading segments ignores extended base	2eedcac8a97c
B14	LDTR/TR extended base is ignored	e37a75a13cda
B15	VEX-prefix is decoded incorrectly	d14cb5df5903
B16	No canonical check on near branches	234f3ce485d5
B17	#DB is injected when RF is set	4161a569065b
B18	RF=1 after instruction emulation	4467c3f1ad16
B19	POPF restores RF	163b135e7b09
B20	Broken vendor specific instructions check	3a6095a0173a
B21	Wrong error code on limit violation	3606189fa3da
B22	No #GP on descriptor load w/L=1&D=0	040c8dc8a5af
B23	Guest userspace can crash VM	a2b9e6c1a35a
B24	Spurious segment type checks	c49c759f7a68
B25	Wrong exception when using DR4/5	16f8a6f9798a
B26	Instructions that cross page boundary fault	08da44aedba0
B27	SYSCALL mistakenly clears FLAGS[1]	807c142595ab
B28	#GP exception instead of #SS	abc7d8a4c935
B29	Missing limit checks on RIP assignments	d50eaa18039b
B30	Segment privilege checked on each access	a7315d2f3c6c
B31	Wrong stack size on linear address calc.	1c1c35ae4b75
B32	MOVNTI minimum opsize is not respected	ed9aad215ff3
B33	No #GP when loading non-canonical base	9a9abf6b6127
B34	FLAGS are updated on faulting instructions	38827dbd3fb8
B35	MOV to CR3 cannot set bit 63	9d88fca71a99
B36	PUSH sreg is not emulated as new CPUs do	0fcc207c66a7
B37	Wrong flags on CMPS and SCAS	5aca37223626
B38	MOV-sreg to memory uses incorrect size	b5bbf10ee6b6
B39	DR6 incorrect on general detect exception	6d2a0526b09e
B40	Wrong mod/rm decoding	5b38ab877e5b
B41	Wrong address calculation on relative JMP	05c83ec9b73c
B42	No canonical check on far JMP	d1442d85cc30
B43	NULL dereference on PREFETCH	3f6f1480d86b
B44	Wrong CLFLUSH decoding	13e457e0eebf
B45	NULL dereference of memopp	a430c9166312

id	*description*	*patch*
B46	MOVBE reg/mem determined incorrectly	39f062ff51b2
B47	No RIP/RSP mask on 32-bit SYSEXIT	bf0b682c9b6a
B48	Immediate is considered as memory op.	d29b9d7ed76c
B49	Privileged instructions cause #GP on VM86	64a38292ed5f
B50	PUSHF on VM86 does not mask VM flag	bc397a6c914c
B51	Near branches operand size is incorrect	58b7075d059f
B52	#PF error-code does not indicate write	c205fb7d7d4f8
B53	Failure on em_call_far returns success	80976dbb5cb2
B54	No wraparound on LDT/GDT accesses	edccda7ca7e5
B55	POP [ESP] is not emulated correctly	ab708099a061
B56	Segment loads set access bit when it is set	e2cefa746e7e
B57	FNSTCW/FNSTSW cause spurious #PF	16bebefe29d8
B58	#GP on JMP/CALL using call-gate	3dc4bc4f6b92
B59	IRET does not clear IRET blocking	801806d956c2
B60	CMPXCHG does not set A/D	2fcf5c8ae244
B61	ARPL cause spurious exceptions	2276b5116e98
B62	CALL uses incorrect stack size	82268083fa78
B63	Wrong far RET opsize in 64-bit	16794aaaab66
B64	SYSENTER emulation is broken	f3747379accb
B65	32-bit operand wraparound fails	bac155310be3
B66	NULL dereferencing on SLDT/STR	63ea0a49ae0b
B67	MOV CR/DR does not ignore MOD	9b88ae99d2fe
B68	Mishandling REP-string 32-bit counters	ee122a7109e4
B69	Discrepencies on zero iterations rep-string	428e3d08574b
B70	Wrong call-far operand size in 64-bit mode	acac6f89574c
B71	BSF and BSR misbehave when source is zero	900efe200e31
B72	POPA emulation may not clear bits [63:32]	6fd8e1275709
	Debug	
B73	Incorrect MOV RSP to DR	a4ab9d0cf1ef
B74	#GP on MOV DR6 with [63:32]!=0	5777392e83c9
B75	RIP is not advanced on ICEBP	fd2a445a94d2
B76	RF=1 after skipped instruction	bb663c7ada38
B77	RF=0 on fault injection	d6e8c8545651
B78	RF=0 on interrupt during REP-string	b9a1ecb909e8
B79	DR6/7.RTM cannot be written	6f43ed01e87c
B80	DR7.GD is not cleared on #DB	6bdf06625d24
B81	Breakpoints do not consider base	82b32774c2d0
B82	DR6[3:0] not cleared on #DB	7305eb5d8cf1
B83	Wrong DR7 on task-switch when host debug	3db176d5b417
	Local-APIC	
B84	No relocation of APIC base	db324fe6f20b
B85	APIC broadcast does not work	394457a928e0
B86	Wrong local APIC mode	1e1b6c264435
B87	TSC-deadline is not cleared	fae0ba215734
B88	Spurious interrupt on TSC-deadline	1e0ad70cc195

id	*description*	*patch*
B89	Lost interrupt due to race	f210f7572bed
B90	No NMI with disabled LAPIC	173beedc1601
	MSRs	
B91	No #GP on invalid PAT CR	4566654bb9be
B92	No canonical checks on WRMSR	854e8bb1aa06
B93	CPUID limit not reflected	
B94	Fast-string not reflected	
B95	Entry failure on real-mode exception	3ffb24681cc4
B96	No #GP on ICR2 and DFR MSRs	c69d3d9bc168
B97	XD_DISABLE not reflected	
B98	MSR_IA32_BNDCFGS is corrupted after exit	9e9c3fe40bcd
	Task-Switch	
B99	Breakpoints are mistakenly disabled	1f854112553a
B100	CR3/LDTR are saved in TSS	5c7411e29374
B101	Incorrect CPL check on task-switch	2c2ca2d12f5c
B102	Wrong CS.DPL and RPL check	9a4cfb27f723
B103	Clear DR7.LE during task-switch	0e8a09969afb
	Reset	
B104	No INIT and reset differences	d28bc9dd25ce
B105	Exception delivery after reset	5f7552d4a56c
B106	BSP core cannot be reconfigured	58d269d8cccc
B107	CR2 is not cleared on reset	1119022c71fb
B108	DR0-DR3 are not cleared on reset	ae561edeb421
B109	LINT0 was enabled after boot	90de4a187518
	Other	
B110	VMX ignores compability mode	27e6fb5dae28
B111	Perf. counters cause exit storm	671bd9934a86
B112	XSAVES sets all XSTATE_BV bits	df1daba7d1cb
B113	CPL!=0 on protected mode entry	ae9fedc793c4
B114	CR reads ignore compatibility mode	1e32c07955b4
B115	PDPTE[7] is not always reserved	5f7dde7bbb3c
B116	CR3 reserved bits are incorrect	346874c9507a
B117	Wrong reserved bits in page tables	cd9ae5fe47df

Table 3: *Summary of the bugs we found and their associated patches (clickable in the digital format of the paper).*

References

[1] AGESEN, O., MATTSON, J., RUGINA, R., AND SHELDON, J. Software techniques for avoiding hardware virtualization exits. In *USENIX Annual Technical Conference (ATC)* (2011).

[2] ALBERTINI, A. x86 oddities. http://code.google.com/p/corkami/wiki/x86oddities, 2011.

[3] ALKASSAR, E., HILLEBRAND, M., PAUL, W., AND PETROVA, E. Automated verification of a small hypervisor. In *Verified Software: Theories, Tools, Experiments (VSTTE)*, vol. 6217 of *Lecture Notes in Computer Science*. Springer, 2010, pp. 40–54.

[4] AMIT, N. Increase the number of fixed MTRR regs to 10. http://comments.gmane.org/gmane.linux.kernel/1727771, 2014.

[5] AMIT, N. Two CPU conformance issues in KVM/x86. http://article.gmane.org/gmane.comp.emulators.kvm.devel/133306, 2015.

[6] ARCANGELI, A. Using Linux as hypervisor with KVM. CERN Computing Seminar http://indico.cern.ch/event/39755/material/slides/0.pdf, 2008.

[7] BAILEY, M. The economics of virtualization: Moving toward an application-based cost model. www.vmware.com/files/pdf/Virtualization-application-based-cost-model-WP-EN.pdf International Data Corporation (IDC), 2009.

[8] BELLARD, F. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference (ATC)* (2005), pp. 41–46.

[9] BEN-YEHUDA, M., DAY, M. D., DUBITZKY, Z., FACTOR, M., HAR'EL, N., GORDON, A., LIGUORI, A., WASSERMAN, O., AND YASSOUR, B.-A. The turtles project: Design and implementation of nested virtualization. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)* (2010).

[10] BENNÉE, A. Validating and defending QEMU TCG targets. KVM Forum, 2014.

[11] BEULICH, J. x86-64: properly handle FPU code/data selectors. Linux Kernel Mailing List http://lkml.org/lkml/2013/10/16/258, 2013.

[12] BHARGAVA, R., SEREBRIN, B., SPADINI, F., AND MANNE, S. Accelerating two-dimensional page walks for virtualized systems. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)* (2008).

[13] BITTMAN, T. J., DAWSON, P., AND WARRILOW, M. Magic quadrant for x86 server virtualization infrastructure. Tech. Rep. ID:G00268538, Gartner, Inc., July 2015. http://www.gartner.com/technology/reprints.do?id=1-2JFZ1KP&ct=150715.

[14] BITTMAN, T. J., MARGEVICIUS, M. A., AND DAWSON, P. Magic quadrant for x86 server virtualization infrastructure. Tech. Rep. ID:G00262673, Gartner, Inc., 2014.

[15] BOHRA, A. E., AND CHAUDHARY, V. VMeter: Power modelling for virtualized clouds. In *IEEE Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)* (2010), pp. 1–8.

[16] BONZINI, P. KVM: x86 emulator: emulate MOVAPS and MOVAPD SSE instructions. Linux Kernel Mailing List http://lkml.org/lkml/2014/3/17/384, 2014.

[17] BULYGIN, Y., LOUCAIDES, J., FURTAK, A., BAZHANIUK, O., AND MATROSOV, A. Summary of attacks against BIOS and secure boot. DEF CON, 2014.

[18] CITRIX SYSTEMS. Xen security advisories. http://xenbits.xen.org/xsa/. Visited: Mar 2015.

[19] CITRIX SYSTEMS. Citrix XenServer 6.2.0 administrator's guide, 2014.

[20] COHEN, E., PAUL, W., AND SCHMALTZ, S. Theory of multi core hypervisor verification. In *SOFSEM: Theory and Practice of Computer Science*. Springer, 2013, pp. 1–27.

[21] CORBET, J. Safe device assignment with VFIO. LWN.net, http://lwn.net/Articles/474088/, 2012.

[22] DALL, C., AND NIEH, J. KVM/ARM: The design and implementation of the Linux ARM hypervisor. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)* (2014), pp. 333–348.

[23] DARROW, B. Is live migration coming to Amazon Web Services? smart money says yes. Gigaom, 2014.

[24] DARROW, B. Xen security issue prompts Amazon, Rackspace cloud reboots. Gigaom, 2015.

[25] ELHAGE, N. Virtunoid: Breaking out of KVM. *Black Hat USA* (2011).

[26] ERSEK, L. Open virtual machine firmware (OVMF status report). http://www.linux-kvm.org/downloads/lersek/ovmf-whitepaper-c770f8c.txt, 2014.

[27] gcova test coverage program. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, 2015.

[28] GOOGLE, INC. Google cloud platform FAQ. http://cloud.google.com/compute/docs/faq. Visited: Feb 2015.

[29] GRUSKOVNJAK, J. Advanced exploitation of Xen hypervisor Sysret VM escape vulnerability. VUPEN Vulnerability Research Team (VRT) Blog, 2012. http://www.vupen.com/blog/20120904.Advanced_Exploitation_of_Xen_Sysret_VM_Escape_CVE-2012-0217.php.

[30] HONIG, A. Security hardening of KVM. KVM Forum, 2014.

[31] INTEL CORPORATION. Intel 64 and IA-32 Architectures Software Developer's Manual. Reference number: 325462, 2014.

[32] INTEL CORPORATION. Intel virtualization technology for directed I/O, architecture specification, Rev. 2.3, 2014.

[33] INTEL CORPORATION. Intel Xeon processor E5 family specification update. Reference number 326510-015, 2014.

[34] KERNEL BUG TRACKER. Bug 86161. http://bugzilla.kernel.org/show_bug.cgi?id=86161, 2014.

[35] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. KVM: the Linux virtual machine monitor. In *Ottawa Linux Symposium (OLS)* (2007), vol. 1, pp. 225–230.

[36] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. KVM: the Linux Virtual Machine Monitor. *Ottawa Linux Symposium (OLS)* (2007).

[37] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., ET AL. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles (SIGOPS)* (2009), pp. 207–220.

[38] LAADAN, O., VIENNOT, N., AND NIEH, J. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2010), pp. 155–166.

[39] LAWTON, K. P. Bochs: A portable pc emulator for unix/x. *Linux Journal 1996*, 29es (1996), 7.

[40] LE, M., AND TAMIR, Y. Rehype: Enabling VM survival across hypervisor failures. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)* (2011), pp. 63–74.

[41] LEINENBACH, D., AND SANTEN, T. Verifying the Microsoft Hyper-V hypervisor with VCC. In *FM 2009: Formal Methods*. Springer, 2009, pp. 806–809.

[42] MARTIGNONI, L., MCCAMANT, S., POOSANKAM, P., SONG, D., AND MANIATIS, P. Path-exploration lifting: Hi-fi tests for lo-fi emulators. *ACM SIGARCH Computer Architecture News (CAN) 40*, 1 (2012), 337–348.

[43] MARTIGNONI, L., PALEARI, R., FRESI ROGLIA, G., AND BRUSCHI, D. Testing system virtual machines. In *ACM International Symposium on Software Testing and Analysis (ISSTA)* (2010), pp. 171–182.

[44] MARTIGNONI, L., PALEARI, R., ROGLIA, G. F., AND BRUSCHI, D. Testing CPU emulators. In *ACM International Symposium on Software Testing and Analysis (ISSTA)* (2009), pp. 261–272.

[45] MATTSON, J. Running nested VMs. http://communities.vmware.com/docs/DOC-8970, 2015.

[46] MCCOYD, M., KRUG, R. B., GOEL, D., DAHLIN, M., AND YOUNG, W. Building a hypervisor on a formally verifiable protection layer. In *IEEE Hawaii International Conference on System Sciences (HICSS)* (2013), pp. 5069–5078.

[47] MCKENNEY, P. E. Reducing OS jitter due to per-CPU kthreads. Linux 3.19:Documentation/kernel-per-CPU-kthreads.txt.

[48] MICROSOFT. ”0x20001” stop error when you start a Linux VM in Windows Server 2008 R2 SP1. KB2550569, 2011.

[49] MICROSOFT. Cross-page memory read or write operation crashes virtual machine. KB2894485, 2013.

[50] NATAPOV, G. KVM: VMX: mark unusable segment as nonpresent. http://comments.gmane.org/gmane.comp.emulators.kvm.devel/111948, 2013.

[51] NAYSHTUT, A. KVM: x86: emulate MOVDQA. http://bugs.launchpad.net/ubuntu/+source/linux/+bug/1330177, 2014.

[52] NGUYEN, A., RAJ, H., RAYANCHU, S., SAROIU, S., AND WOLMAN, A. Delusional boot: Securing hypervisors without massive re-engineering. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)* (2012), pp. 141–154.

[53] POPEK, G. J., AND GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. *Communications of the ACM (CACM) 17* (1974), 412–421.

[54] The QEMU machine protocol (QMP). http://wiki.qemu.org/QMP. Accessed: Aug 2015.

[55] RED HAT, INC. Red Hat vulnerabilities. http://access.redhat.com/security/cve/, 2014.

[56] RED HAT, INC. Bug 1167595. Bugzilla http://bugzilla.redhat.com/show_bug.cgi?id=1167595, 2015.

[57] RED HAT, INC. Linux containers compared to KVM virtualization. http://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Resource_Management_and_Linux_Containers_Guide/sec-Linux_Containers_Compared_to_KVM_Virtualization.html, 2015.

[58] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security (CCS)* (2009), pp. 199–212.

[59] ROEDEL, J. SVM: Keep intercepting task switching with NPT enabled. KVM mailing list, http://thread.gmane.org/gmane.comp.emulators.kvm.devel/80905, 2011.

[60] ROTITHOR, H. Postsilicon validation methodology for microprocessors. *IEEE Design & Test of Computers 17*, 4 (2000), 77–88.

[61] SINITSYN, V. Understanding the Jailhouse hypervisor. LWN.net, http://lwn.net/Articles/578295, 2014.

[62] SLAVICIC, S., AND CAMPBELL, B. XP machine freeze. KVM mailing list, http://comments.gmane.org/gmane.comp.emulators.kvm.devel/133956, 2014.

[63] SOUNDARARAJAN, V., AND ANDERSON, J. M. The impact of management operations on the virtualized datacenter. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)* (2010), pp. 326–337.

[64] STEINBERG, U., AND KAUER, B. NOVA: a microhypervisor-based secure virtualization architecture. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)* (2010), pp. 209–222.

[65] SZEFER, J., KELLER, E., LEE, R. B., AND REXFORD, J. Eliminating the hypervisor attack surface for a more secure cloud. In *ACM Conference on Computer and Communications Security (CCS)* (2011), pp. 401–412.

[66] THE LINUX KERNEL ORGANIZATION. Kernel bug tracker. http://bugzilla.kernel.org. Visited: Aug 2014.

[67] UBUNTU BUG TRACKER. Bug 1268906. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1268906.

[68] UBUNTU BUG TRACKER. Bug 924247. http://bugs.launchpad.net/ubuntu/+source/qemu-kvm/+bug/924247,.

[69] WEINS, K. Xen bug drives cloud reboot: Survey shows users undeterred. http://www.rightscale.com/blog/cloud-industry-insights/xen-bug-drives-cloud-reboot-survey-shows-users-undeterred, 2014.

[70] WIKIPEDIA. In-target probe. https://en.wikipedia.org/wiki/In-target_probe. Accessed: Aug 2015.

[71] WILHELM, F., AND LUFT, M. Security assessment of Microsoft Hyper-V. ERNW Newsletter 43, 2014.

[72] WILLIAMSON, A. KVM: x86 emulator: emulate MOVNTDQ. Linux Kernel Mailing List http://lkml.org/lkml/2014/7/11/569, 2014.

[73] XEN PROJECT. Nested virtualization in Xen. http://wiki.xenproject.org/wiki/Nested_Virtualization_in_Xen, 2014.