NestFuzz: A Framework for Fuzzing Nested Virtualization Environments

Changmin Teng
Brown University

Abstract

With the rapid spread of cloud computing, virtualization technology, as its backbone, has seen wide adoption. In order to achieve better performance, CPU vendors have implemented hardware virtualization support such as Intel-VT and AMD-V to enhance the performance of VMMs. Utilizing these features, many hypervisors support nested virtualization which can be used to test the deployment of virtualized environments. However, given the complex logic behind the implementation of nested virtualization, many bugs exist in nested virtualization support for many hypervisors and there is yet an effective method to test for those bugs. This technical report proposes NestFuzz, a framework that sets up a nested virtualization environment and apply fuzzing to Intel VMX instructions.

1 Introduction

Cloud-based services have seen increasingly widened application these days, enables users to offload increasingly variated form of workloads to those services. Virtualization technology plays a key role by allowing cloud providers to better utilize and easier to manage hardware resources by sharing physical hardware between users and dynamically allocate the resources to fit each user's needs in a timely manner.

In order to achieve isolation between each VM and provide its user with a safe and private working environment, hypervisors have to simulate all the hardware interaction, including instruction handling, traps and interruption, memory address translation and port/memory-mapped I/O, etc. The naive approach will attempt to simulate all of them with software, which introduces prohibitively high overhead even with currently available hardware. One of the major approaches for hypervisors to achieve better performance is to offload some of the above functionalities to the CPU. CPU vendors have implemented hardware virtualization support such as Intel-VT and AMD-V to aid in the virtualization process. For example, Intel-VT is capable of setting a higher privilege level than ring-0 to implement hypervisor with the native OS running directly on top of the physical CPU without binary translation. Intel-VT also implements extended page table and I/O virtualization support to relieve the burden of managing guest address translation and I/O emulation.

However, with the increasingly widespread adoption of virtualization technology, bugs in major hypervisors also cause more damage than ever, especially when those bugs are turned into exploits that can sabotage the service and even steal information from the users. Many research focuses on discovering and analyzing vulnerabilities in hypervisors. One of the promising ways to do this is to stress-test the hypervisors by feeding them with random inputs and observe any malfunction, which is known as fuzzing. Fuzzing has shown great potential in finding bugs in user-space applications (e.g. AFL, Peach fuzzer) and even OS kernels [8]. Researches in hypervisor fuzzing so far have covered hypercall interfaces, instruction handling, and I/O etc [1, 3–7].

Other than those, there is another interface that is not getting attention so far, and yet possibly has many bugs due to the complexity of the codebase: nested virtualization support. In order to make the overhead as low as possible, the modern design uses a single layer of hardware virtualization instead of multiple layer virtualization [2]. This introduces a new interface to VMs which is the simulation of Intel VMX instructions. This technical report aims to fill this blank in nested virtualization by making the following contributions:

- Differentiating interfaces of nested hypervisor fuzzing from currently well-tested ones
- Propose an approach for nested hypervisor fuzzing
- Implement a proof of concept framework that shows the possibility of nested hypervisor fuzzing

2 Background

2.1 Intel Virtualization Technology (Intel VT)

Intel VT is a hardware virtualization support feature. It implements several features to aid the hypervisor in creating and managing VMs. For instance, it implements a ring -1 privilege mode, so that unmodified kernel code can run directly on physical CPU without binary translation, while having hypervisor limit its actions by trapping VCPU-sensitive instructions and other sensitive events such as interruption. Much like the isolation scheme in OS kernels, sensitive operations will trigger a VM-exit into the hypervisor and invoke the respective handler and after the handler has done its job, CPU will perform a VM-entry to resume the execution of the VM. One can also assign an extended page table (EPT) to act as guest physical memory, so that address translation will not cause a VM-exit. Intel VT introduces an in-memory data structure called VMCS to manage what event causes a trap into hyperviser and so on. To provide a concept of what exactly is done by hypervisors using Intel VT, here are the rough steps to enable VMX to apply a minimal VMCS:

- check `cpuid` and `IA32_FEATURE_CONTROL` MSR to make sure VMX feature is supported and enabled on current CPU
- check some specific fields in cr0 and cr4
- read from `IA32_VMX_BASIC` MSR and store the value in to the id field of VMXON region/VMCS later on
- create a data structure called VMXON region and execute `vmxon` instruction with the physical address of that region
- create a data structure called VMCS and execute `vmptrld` instruction with the physical address of that region
- use `vmread` and `vmwrite` to manipulate the behavior of current VM through modifiying VMCS

In addition, `vmptrst` instruction can be used to obtain current VMXON region.

Further steps should be taken to enable EPT and I/O virtualization support, which we will not elaborate on since they are not involved at this moment.

2.2 Nested Virtualization

Ben et.al [2] proposed a scheme of nested virtualization fully utilizing Intel VT to achieve flexibility and low overhead. This scheme is implemented in KVM and has achieved good stability over the years of extensive usage. They multiplexed multi-level CPU virtualization, memory (MMU) virtualization, and I/O virtualization with a single level of hardware support to avoid the high overhead.

The key to implementing such a feature is the proper management of VMCS. Suppose we have a host hypervisor, L_0, running a guest hypervisor, L_1, with a nested guest OS L_2 on the top. When L_0 created a VM for L_1, $VMCS_{0\to1}$ is used to manage sensitive actions in L_1. In the same way, $VMCS_{1\to2}$ is created when L_1 spwans L_2. In order to run L_2 directly on the CPU, the VMCS used here should be a combined version of $VMCS_{0\to1}$ and $VMCS_{1\to2}$, $VMCS_{0\to2}$ which applies the most restrictive rule that covers both VMCSs. By carefully setting handlers and VM-exit conditions via VMCS and correctly emulate VM-entires and exits for multi-layer virtualization, nested virtualization can be achieved with the nested VMs running directly on the CPU as much as possible, and also provides Intel VMX simulation within the VM. However, given the complex logic and abundance of edge cases of such implementation, many bugs could exist in the process. Therefore, we are focusing on VMX emulation as the target for fuzzing in this project.

2.3 Related Work

Several hypervisor fuzzing schemes have been proposed. For instance, IOFUZZ [6] applied random input on port-mapped I/O for random ports. McDaniel et.el [5] implemented a framework to apply Peach Fuzzer on virtual device drivers. Rule support in Peach Fuzzer can achieve more code coverage in an emulated device driver by generating rigid commands that suit each driver. Another possible approach is to exhaustively execute a random byte stream to test for VCPU. Domas [3] created a processor fuzzer on physical CPUs and discovered flaws in enterprise hypervisors, alongside undocumented instructions, hardware glitches, and bugs in the assembler and disassemblers. Amit et.al [1] applied Intel's CPU validation tool to VCPUs, which consists of a specific CPU sequence test and random input tests.

An example of more sophisticated fuzzing is VDF [4], where it uses AFL to apply evolutionary fuzzing to memory and port-mapped I/O for hypervisors. HYPER-CUBE [7] is implemented to conduct a comprehensive fuzzing for a device driver, hypercall, and direct I/O simulation. This is by far the most comprehensive fuzzing platform for hypervisors. However, to the extent of our knowledge, there is no fuzzing framework targeting nested virtualization.

3 Design

NestFuzz has a structure shown in Figure 1. The fuzzing process is started from the host OS and will first run a VM on top of host VM (KVM in the current implementation) with nested virtualization support enabled (i.e. with VMX emulation enabled). After the VM booted, we mount a kernel

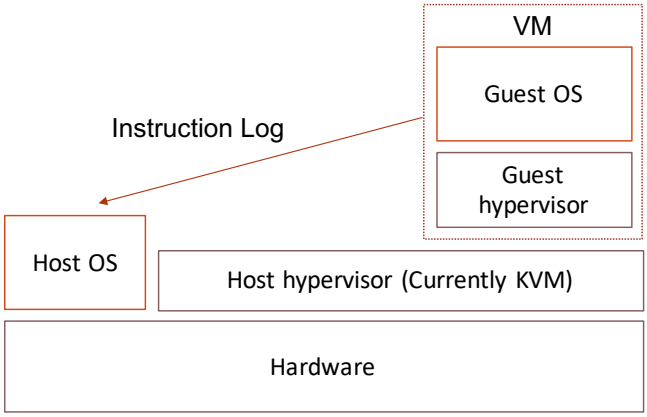

Figure 1: Structure of the fuzzer

module inside that VM, which will set up a minimal guest hypervisor beneath this VM. This results in this VM running on two layers of hypervisors, creating a nested virtualization environment.

Actual fuzzing process starts after this with guest hypervisor generates random Intel VMX instructions (`vmread`, `vmwrite`, `vmptrld`, `vptrst`) with random arguments and execute them on top of emulated VMX. Guest OS will compose a log with specific instructions executed and send them to host OS. When an instruction causes the VM or host hypervisor to crash, NetFuzz will put the result into the log and start a new VM to repeat the same procedure. The user can later analyze the log to find out what sequence of instructions caused the result.

4 Implementation

We have implemented NestFuzz on top of KVM on x86_64 architecture. Based on the scheme in figure 1, NestFuzz consists of three parts: the GNU expect script running in host OS, the minimal guest hypervisor running right beneath guest OS, a tester running in VM with two parts: a user-space tester and a testing kernel module that works in pair.

The GNU expect script controls the whole fuzzing process(VM startup, execution of the tester, etc.) and logging. the script starts the VM via the KVM-enabled qemu process and detects crashes by checking the return value of the qemu process. In case an instruction causes freeze instead of crashing the VM, a timeout is set after the tester starts so that when the VM is not responding (i.e stopped outputting logs), the VM will be terminated and a new VM will be created to continue fuzzing. With this scheme, there is a limitation that we can only make sure the crashes of the user-space part of KVM will be handled. For the ease of development, other components that run in the VM are stored on the host OS instead of in the image of the VM. They are accessed from the VM through a shared directory mounted through a 9p file system to guest OS right after its startup.

The guest hypervisor is built based on a prototype hypervisor called `mentor`, which is developed by the Secure Systems Lab (SSL) at Brown CS and is used to aid in discovering a bug in nested virtualization. `mentor` conducts the minimal setup listed in section 2.1, create and shift to EPT and implements a small hypercall interface. In NestFuzz, we extended the hypercall interface to support four additional VMX instructions: `vmread`, `vmwrite`, `vmptrld` and `vptrst` for use in the fuzzing. Note that currently `mentor` in the nested virtualization environment of this project will freeze while loading in a VM with multiple VCPUs, thus the VM can only have one VCPU in our setup. This does not have a significant effect at this moment but may need attention in further development.

The user-space part of the tester is a simple C program intended to be a bridge between the guest hypervisor and the host OS since the whole GNU expect scheme is based on running commands and getting output through a bash running on the guest OS. The user-space tester will simply randomly select one of the four supported VMX commands evenly and let the guest hypervisor execute them with random arguments if available. This requires using hypercall interface, which can only be successfully invoked from kernel space. The test kernel module is created for this propose, which will expose a `ioctl()` interface to the user-space tester to invoke the corresponding hypercall. The userspace tester prints the actual instruction to the serial console before executing it, which will be received by the expect script and put in the log. Upon a crash or a timeout, the expect script will append the execution result (timeout, exited normally or not) and the return value of the qemu to the end of the log.

5 Evaluation

We conducted our evaluation using Ubuntu 18.04.4 with kernel version 4.18.0-25-generic as both host and guest OS. Host OS is running on a VMWare virtual machine (host OS need not to be virtualized by design), with 4 VCPUs and 8 GB of main memory allocated. The VM is allocated 1 VCPU and 2GB of main memory. All of this setup is running on an Intel Core i7-8850H CPU @2.60GHz.

5.1 Performance

This experiment aims to evaluate how fast can NestFuzz feeds instructions to the target hypervisor. We simply let the fuzzing run for an hour and counted instructions executed and times VM started. We obtained a relatively stable result between several tests with 480 instructions executed and booting the VM 120 times within an hour. Among the four instructions, only `vmwrite` would cause a VM freeze since the only other instruction that could change the CPU state, `vmptrld`, will fail to execute when the VMXON region pointed to does not

have correct id field and does not do anything. This means on average four instructions can be executed in each VM session. Since waiting for the timeout and booting process takes a long time to execute compared to actual instructions, VM will be booted for roughly fixed times in a certain period, and instructions executed will be about four times of it.

5.2 Accuracy

During the development and experiment, we haven't been able to observe an actual crash of KVM. To make sure that NestFuzz can actually catch crashes, we injected one line of code into qemu source code to cause a segmentation fault. In such cases, the log shown following message indicating an error exit:

```
-- qemu: exited with status: 139 --
```

which means non-zero exit status can be used to detect crash in qemu.

6 Future Work

There are several aspects of the future improvement of NestFuzz. First is improving the performance. As can be seen from performance evaluation, VM boot time contributes to the vast majority of the execution time and severely limits the actual VMX instructions executed. A possible approach is to implement a minimal kernel as in HYPER-CUBE [7] to minimize boot time. Other than writing the minimal kernel, we will also need to implement a special channel for communication between host and guest OS.

The second is to enhance compatibility by granting the ability to detect and handle crashes of the kernel module part of KVM, and other hypervisors with kernel space components. Two possible approaches can be considered. One is to have a wrapper around the kernel module of hypervisors so that when the kernel module crashes, the wrapper can gather error information and restart the module as needed. This approach requires different implementation for each hypervisor which could introduce a large amount of work when trying to apply NestFuzz to other hypervisors. The other approach is to add another layer of virtualization and control the host OS/hypervisor altogether in a virtualized environment. This approach will need to explicitly establish a communication channel through the host OS, all the way to the guest OS, which could be tricky.

The last direction is to increase the effectiveness of the fuzzing process. Fuzzers these days will usually have some advanced fuzzing techniques instead of blindly generating random workloads such as coverage-based workload generation and rule-based fuzzing. Applying these to NestFuzz would increase the code coverage and thus make it likely to find new bugs. The challenge here is how to instrument the hypervisor to measure code coverage, and what rule is suitable for VMX fuzzing. At the same time, we can also seek to incorporate other VMX instructions (EPT modification, direct VMCS modification, privilege ring modification, MSR access, etc.) in the hope to get more code coverage.

Other than those, as a more trivial improvement, we could create a better user interface that enables more efficient analysis.

7 Conclusion

Current hypervisor fuzzing scheme covers instruction, I/O handling, and hypercall interface but not nested virtualization support which possibly contains a plethora of bugs. In this report, we pointed to this fact and proposed a fuzzing scheme that aims to fill this hole by targeting the Intel VMX subsystem. We constructed a simple fuzzer that uses GNU expect script to control an inner host VM, with testing modules and a minimalist hypervisor on the VM to generate workloads. This proof-of-concept system to show the possibility of automated fuzzing in nested virtualization environments. It also opens up the possibility of a more sophisticated and efficient fuzzing framework by optimizing in the aspect of performance, compatibility, and depth of fuzzing.

References

[1] Nadav Amit, Dan Tsafrir, Assaf Schuster, Ahmad Ayoub, and Eran Shlomo. Virtual CPU validation. In *Proceedings of the 25th Symposium on Operating Systems Principles - SOSP '15*, pages 311–327, 2015.

[2] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[3] Christopher Domas. Breaking the x86 ISA. In *Black Hat*, 2017.

[4] Andrew Henderson, Heng Yin, Guang Jin, Hao Han, and Hongmei Deng. VDF: Targeted Evolutionary Fuzz Testing of Virtual Devices. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10453 LNCS, pages 3–25. 2017.

[5] Lucas McDaniel and Kara Nance. Identifying Weaknesses in VM/Hypervisor Interfaces. In *2013 46th Hawaii International Conference on System Sciences*, 2013.

[6] Tavis Ormandy. An empirical study into the security exposure to hosts of hostile virtualized environments, 2007.

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

[8] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *Network and Distributed System Security Symposium (NDSS)*, 2017.