Characterizing the VM-Hypervisor Attack Surface

Diego Perez-Botero
Princeton University, Princeton, NJ, USA
diegop@princeton.edu

Ruby B. Lee
Princeton University, Princeton, NJ, USA
rblee@princeton.edu

ABSTRACT

Virtual Machines (VMs) have become commonplace in modern computing, as they enable the execution of multiple isolated Operating System instances on a single physical machine. This increases resource utilization, makes administrative tasks easier, minimizes fault propagation, improves state encapsulation for migration-based load balancing, and provides many other benefits. Unfortunately, with the rise of Cloud Computing, situations in which co-hosted VMs are untrusted to one another have led to security concerns, taking into account that resources are shared and mediated by a Hypervisor that may be targeted by a rogue VM.

VM Exits have been viewed as an exploitable mechanism for rogue VMs to compromise co-hosted VMs' security properties (confidentiality, integrity, availability). We put that belief to test by conducting experiments on a Xen 4.0.1 Hypervisor. After characterizing what would be considered normal VM Exit behaviour for standard Cloud Computing workloads, we attempt to generate denial-of-service conditions for co-hosted machines with attacks based on the CPUID and RDTSC instructions. Pushing the hypervisor to the limit with massive amounts of VM Exits proves to have the same effect as running simple co-hosted CPU hogs. Meanwhile, brute-forcing the hypervisor's implementation of the CPUID instruction also fails to generate the desired results.

In spite of not being able to negatively affect a co-hosted VM's security properties in our test bed, the thinking process described in this paper sets a good foundation for future research on VM Exit security.

1. INTRODUCTION

Intel supports virtualization of x86 processor hardware by way of an additional set of architectural features (i.e. new instructions and control data structures) referred to as Virtual-Machine Extensions (VMX) [4]. While the Hypervisor runs on VMX root mode, guest VMs run on VMX non-root mode. This allows the Hypervisor to retain control of processor resources, given that the non-root operations are restricted. Throughout a VM's lifetime, certain sensitive instructions (e.g. CPUID, GETSEC, INVD) and events (e.g. exceptions and interrupts) cause *VM Exits* to the Hypervisor. These VM Exits are handled by the Hypervisor, who decides the appropriate action to take and then transfers control back to the VM via a *VM Entry*. A Virtual Machine Control Data Structure (VMCS) stores the data needed by the Hypervisor to restore the guest VM's state once it has handled the VM Exit and also contains information regarding the VM Exit's cause and nature. As such, the VMCS acts as a limited communication channel between the Hypervisor and the hosted VMs. While useful, the direct VM-Hypervisor interaction that VM Exits facilitate has been regarded as an attack vector in previous work [7]. Furthermore, bugs related with the handling of VM Exits have been found in popular Hypervisors, such as Xen [10] [15].

We set out to thoroughly characterize VM Exits from a security perspective. Our objectives for this paper are to:

1. Analyze the information that is exchanged between a VM and the Hypervisor during a VM Exit and assess the possibility of performing attacks by sending malicious input through that communication channel.
2. Run different types of applications (I/O intensive, computationally intensive, etc.) and study the properties of the most common VM Exits.
3. Determine the performance cost of VM Exits and ponder whether an attacker in control of a VM could use them to threaten the availability of co-hosted VMs.

2. TEST BED

Given that Intel's Software Developer's manual [4] covers every possible low-level detail regarding VMX non-root operation in a 64-bit and 32-bit Intel x86 architecture with virtualization extensions, the nature of this investigation is purely empirical. We are interested in uncovering new knowledge about the effects of VM Exits on the VM-Hypervisor interactions and will only refer to x86 architecture traits if it helps in the interpretation of our data.

Our test bed consists of a machine with an Intel Core2Duo T7100 (1.8GHz) CPU and 3GB DDR2 667 MHz RAM, on top of which a Xen 4.0.1 Type-I (bare metal) hypervisor will run. The Dom0 host is a paravirt_ops[1] kernel based on Linux 2.6.32.50. Meanwhile, all DomU hosts are fully-virtualized Ubuntu 10.04.3 VMs.

[1]wiki.xen.org/xenwiki/XenParavirtOps.html

3. VM EXITS DURING NORMAL OPERATION

The first step towards understanding VM Exits in practice is to analyze their behaviour under normal conditions. For this purpose, we prepared three different workload sets:

- **Idle System:** the simplest type of workload is an idle system. It will show us how OS background tasks behave, giving us an idea of typical OS-generated "noise" when running more elaborate workloads.
- **Apache HTTP Server[2]:** web servers are a popular workload in the Cloud Computing context. They are IO-intensive processes, constantly interacting with Network Interface Cards and non-volatile storage devices.
- **Freebench[3] benchmark suite:** while web servers generate low CPU loads, there exists another important spectrum of applications that is very CPU-dependent. Compression, 3D rendering, encryption/decryption, and simulations of complex systems (e.g. weather and earthquakes) are just a few examples. The Freebench benchmark suite runs 10 CPU-heavy tests in sequential order to evaluate the processing power of an execution environment.

[2]http://httpd.apache.org/
[3]http://code.google.com/p/freebench/

3.1 Basic Trends

After obtaining traces of the aforementioned workloads, we found that only 11 of the 56 possible reasons for VM Exits manifested themselves. The following is a list of that subset and their descriptions [4]:

Table 1: VM Exit Subset Present in Traces

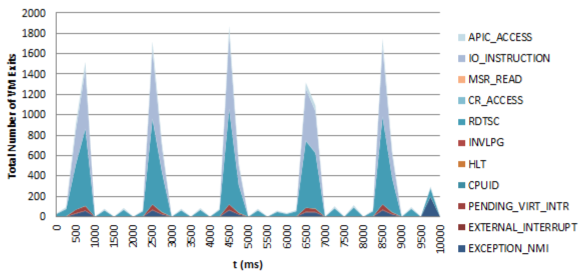
EXIT_REASON	Description
EXCEPTION_NMI	Software caused an exception or a non-maskable interrupt (NMI) occurred. NMIs indicate non-recoverable errors.
EXTERNAL_INTR	An external interrupt arrived. These are caused by software calls to the OS and/or by I/O operations.
PEND_VIRT_INTR	Pending virtual interrupt.
CPUID	Guest software attempted to execute CPUID instruction, which provides processor identification information.
HLT	Guest software attempted to execute HLT instruction, which stops instruction execution and places the processor in a HALT state.
INVLPG	Guest software attempted to execute INVLPG instruction, which invalidates a TLB entry.
RDTSC	Guest software attempted to execute RDTSC instruction, which returns the current value of the processor's time-stamp counter.
CR_ACCESS	Guest software attempted to access CR0, CR3, CR4, or CR8 x86 control registers.
MSR_READ	Guest software attempted to execute RDMSR instruction, which loads the contents of a 64-bit model specific register (MSR).
IO_INSTRUCTION	Guest software attempted to execute an I/O instruction.
APIC_ACCESS	Guest software attempted to access memory at a physical address on the APIC-access page. (APIC stands for "advanced programmable interrupt controllers" and is an Intel technology used in multi-processor systems).

Note: A sampling interval of 250 ms was used for our traces, which means that all calculations of *Total Number of VM Exits*, *Peak Frequency* and *Average Frequency* were made at that granularity.

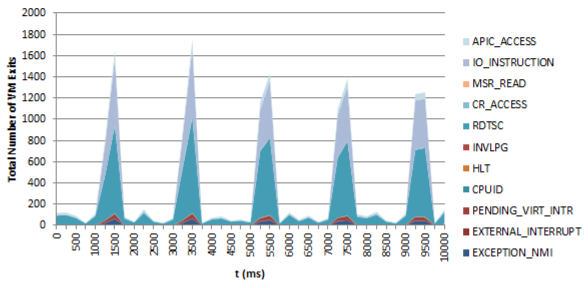
3.1.1 Idle Workloads

As shown in Figure 1(a), the standard Ubuntu 10.04.3 background processes have a cyclic behaviour that generates RDTSC and IO_INSTRUCTION VM Exit peaks every 2000 ms, accompanied by some INVLPG and EXCEPTION_NMI exits. These patterns can be caused by TLB misses or some sort of timeout on the TLB entries. Aside from the recurrent peaks, the total number of VM Exits at a given time is always below 100 exits per 250 ms when idle.

Figure 1.(b) serves as evidence that the Apache web server does not affect the overall VM Exit pattern when no HTTP packets are being received by the host. This makes sense, taking into account that the server will most likely execute a blocking socket-related line of code, suspending its execution while it waits for incoming connections. VM Exits are associated with sensitive instructions and events, so very few of them should be generated by the web server under these conditions.



(a) VM Exits generated by idle VM



(b) VM Exits generated by VM running Apache HTTP Server without any incoming network traffic

Figure 1: VM Exit traces of VMs with little to no load

3.1.2 Apache HTTP Server with httperf

After knowing what to expect from an idle system, we initiated a performance test on the Apache HTTP Server by using httperf[4]. Figure 2 presents the results of generating a sustained 100 HTTP requests/second load. The total amount of VM Exits per 250 ms interval is almost always above 3000, sometimes reaching 5000. This is roughly twice the amount of VM Exits that were seen in the idle runs. To no surprise, there is a constant presence of APIC_ACCESS exits, which are usually associated with interrupts for I/O operations. Nonetheless, RDTSC and EXCEPTION_NMI exits are the most popular exit types by far. The latter could be due to exception handling code in the Apache server's code base, since exceptions are a commonly used construct in large software projects.

[4]http://www.hpl.hp.com/research/linux/httperf/

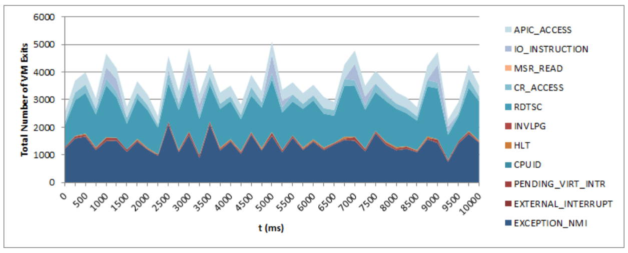


Figure 2: VM Exit traces of Apache HTTP Server under a sustained load of 100 requests/second

3.1.3 Freebench benchmark suite

The Freebench benchmark suite runs 10 tests representing a variety of mostly CPU-heavy workloads. The varied nature of the benchmark leads to irregular VM Exit traces, such as the one shown in Figure 3. There are notorious peaks of EXCEPTION_NMI exits, which most probably serve as evidence of the presence of exception handling code blocks inside numeric methods (e.g. root-finding algorithms). As can be observed in the graph, some 250 ms intervals are exposed to as many as 60,000 VM Exits. It should also be noted that non-peak behaviour, shown inside the "zoom-in" box, follows the same pattern of idle workloads and is around the same order of magnitude (2000 exits per 250 ms time interval). This is coherent with what is expected from a CPU-intensive program, since normal computation (e.g. assign, add, multiply) does not require sensitive instructions besides those associated with memory management (e.g. INVLPG).

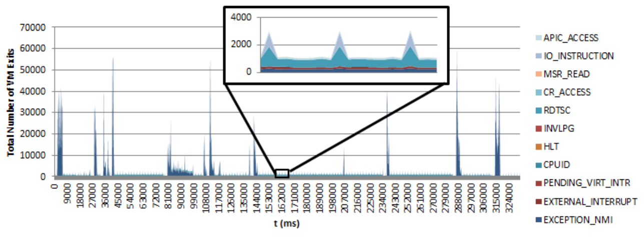


Figure 3: VM Exit traces of a Freebench benchmark suite run

3.2 VM Exit Frequencies

Now that the reasons behind the main characteristics of the different traces have been explained, we proceed to analyze VM Exit frequencies quantitatively. Once again, keep in mind that the sampling interval used was 250 ms.

From the subset of VM Exits that were generated by our test workloads, EXCEPTION_NMI, RDTSC, IO_INSTRUCTION and APIC_ACCESS stand out in all traces when it comes to average frequency (see Table 2). This can be explained by the fact that they are all involved in standard program procedures (exception handling, system time queries, and I/O operations).

Table 2: Average Frequency of Some VM Exit Types

	Avg. Freq. (s^{-1})			
EXIT_REASON	idle	apache	httperf	freebench
EXCEPTION_NMI	85	1833	6914	7113
EXTERNAL_INTR	4	8	59	386
PEND_VIRT_INTR	35	34	140	30
CPUID	0	1	2	3
HLT	7	11	73	2
INVLPG	1	21	40	50
RDTSC	704	892	3784	2418
CR_ACCESS	9	58	545	47
MSR_READ	0	0	0	0
IO_INSTRUCTION	464	463	458	483
APIC_ACCESS	80	99	1058	448

While EXCEPTION_NMI, RDTSC, IO_INSTRUCTION and APIC_ACCESS are still important when the focus is switched to peak frequencies, two additional VM Exits also stand out in Table 3: CR_ACCESS and INVLPG. Since they are associated with Control Registers and Translation Lookaside Buffers, which are maintained by the hardware and the OS, their low average frequencies are just a reflection of their relative independence from the overall application execution. They only come into play for rare events (full TLB flushes and temporary changes in the behaviour of the CPU).

Table 3: Peak Frequency of Some VM Exit Types

	Peak Freq. (s^{-1})			
EXIT_REASON	idle	apache	httperf	freebench
EXCEPTION_NMI	30772	147688	232004	230616
EXTERNAL_INTR	64	192	216	1232
PEND_VIRT_INTR	324	296	520	320
CPUID	32	96	160	136
HLT	56	68	256	76
INVLPG	588	2560	3256	8196
RDTSC	4852	6424	7732	6072
CR_ACCESS	1568	3256	2296	1084
MSR_READ	32	0	0	32
IO_INSTRUCTION	3660	3708	3648	4912
APIC_ACCESS	524	564	2328	744

From Tables 2 and 3, one can infer that EXCEPTION_NMI exits come in bursts, given that they have blistering peak frequencies (230,000/s in the httperf and freebench traces) and more moderate, yet significant, averages (around 7,000/s in the same traces). After going through Xen's source code, we notice that, even though EXCEPTION_NMI exits are common, the information flow in the VM-to-Hypervisor direction is very limited during those exits. If the exit is due to a software exception, the hypervisor updates the guest VM's EIP register (instruction pointer). Meanwhile, a non-maskable interrupt just leads to an update in the guest VM's control registers along with an exception directed to the OS.

3.3 VM Exit Performance Hit

We define the *performance hit* of a VM Exit as the downtime resulting from it being handled. In other words, a VM Exit's performance hit is the time between the VM_EXIT event that hands control to the hypervisor and the subsequent VM_ENTRY event that returns control to the VM.

Table 4 contains average performance hits for the 11 types of VM Exits that were detected in our test bed. Notice that they are measured in *microseconds (μs)*. In general, abstract VM Exit types, such as EXCEPTION_NMI and EXTERNAL_INTR, record the highest downtimes because they are highly dependent on the application's code (an I/O operation with a 1024 byte buffer takes longer than one with a 4096 byte buffer). On the other hand, VM Exits that directly translate to a well-specified instruction (e.g. CPUID, INVLPG and RDTSC) provide very consistent VM_EXIT-to-VM_ENTRY times regardless of the workload being run.

Table 4: Average Performance Hit for Some VM Exit Types

	Avg. Performance Hit (μs)			
EXIT_REASON	idle	apache	httperf	freebench
EXCEPTION_NMI	3.63	2.44	3.97	8.76
EXTERNAL_INTR	6.95	45.53	21.16	1799.64
PEND_VIRT_INTR	3.41	2.92	2.72	3.11
CPUID	2.93	2.46	2.49	2.52
HLT	5.10	5.52	2.87	5.56
INVLPG	3.92	2.66	2.62	2.87
RDTSC	3.97	6.29	5.36	3.11
CR_ACCESS	6.52	6.36	3.97	4.34
MSR_READ	4.26	0.00	0.00	1.94
IO_INSTRUCTION	2.49	2.23	2.31	2.27
APIC_ACCESS	3.62	3.58	2.62	2.47

Another interesting measure, contained in Table 5, is the proportion of total execution time that is spent inside the hypervisor's VM Exit-handling code. We will ignore the EXTERNAL_INTR row, since those exits are generated by system calls and/or I/O operations and the subsequent VM_ENTRY event is accompanied by the final result, so the applications are still making progress by delegating their operations to the hypervisor, which does not constitute downtime.

In general, it can be observed that VM Exits do not contribute a lot of overhead to the total execution time (around 7% for freebench, 5% for the httperf trace, and close to 1% for the idle workloads). This is consistent with the findings of Keller et al. [7], who documented a 1% performance improvement when the hypervisor is eliminated (and thus eliminating the VM Exits).

Table 5: % Of Total Execution Time Dedicated to Handling Some VM Exit Types

	% Of Total Execution Time			
EXIT_REASON	idle	apache	httperf	freebench
EXCEPTION_NMI	0.031	0.448	2.746	6.232
EXTERNAL_INTR	0.003	0.035	0.125	69.381
PEND_VIRT_INTR	0.012	0.010	0.038	0.009
CPUID	0.000	0.000	0.001	0.001
HLT	0.004	0.006	0.021	0.001
INVLPG	0.000	0.006	0.010	0.014
RDTSC	0.280	0.561	2.029	0.751
CR_ACCESS	0.006	0.037	0.216	0.020
MSR_READ	0.000	0.000	0.000	0.000
IO_INSTRUCTION	0.115	0.103	0.106	0.110
APIC_ACCESS	0.029	0.036	0.277	0.111

4. THREAT MODEL

Our *Trusted Computing Base (TCB)*, shown in figure 4, is comprised by the hardware and the hypervisor, excluding its VM Exit Handler module. At the same time, if dealing with a Cloud Computing scenario, the Cloud Provider is trusted. Meanwhile, Co-hosted VMs are untrusted.

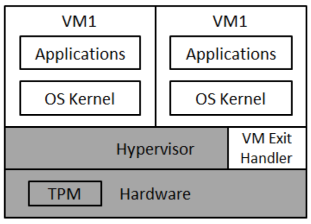


Figure 4: Our Trusted Computing Base (TCB). Gray components are trusted. White components are outside the TCB.

5. ATTACKS

Before attacks can be performed, our limitations need to be identified:

- The attacks should rely on VM Exits as their primary source of success.
- While not required, it is desirable that the attacks run in user space and not in kernel space, since that increases their usage scenarios. For example, some cloud providers provide their own kernels and do not allow root access (e.g. "sudo" commands in linux).

The second limitation described above reduces the VM Exits that should be used as our attack vector. Most VM Exits that directly translate to a well-specified instruction (e.g. INVLPG and HLT) require privilege level 0 (a program running in kernel mode). Out of the 56 different VM Exit types, CPUID and RDTSC are good candidates because they can usually be triggered at any privilege level (see Figure 5) and their generation is straightforward. In addition, the fact that they are used by normal workloads (as seen in Section 3) means that they must be supported by most, if not all, hypervisors.

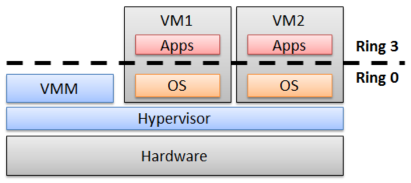


Figure 5: Privilege Levels During Hypervisor Operation

5.1 CPUID Flood Attack

Our first attack attempts to generate a denial-of-service condition for co-hosted VMs by flooding the hypervisor with CPUID VM Exits. A call to CPUID uses the EAX register for input purposes and, in turn, returns the requested information in registers EAX, EBX, ECX, and EDX. We use 0x8000000A as the input to CPUID because, according to the CPUID specification, it is an invalid input that returns the same information as 0x0B (Extended Topology Enumeration leaf) [4]. This way, we are testing the hypervisor's implementation of the CPUID instruction for an obscure detail and, at the same time, we ask for the most computationally expensive output. Listing 1 contains the assembly language code needed to generate an infinite loop of cpuid calls with input 0x8000000A:

```
.section .text
.global main
main:
   movl $0x8000000A, %eax
   cpuid
   jmp main
```

Listing 1: CPUID Flood Attack Code (AT&T Assembly Language)

The resulting VM Exit trace, shown in Figure 6, is dominated by CPUID exits. Some CPUID bursts exceed the 100,000 exits per 250 ms interval, which is more than 2,500 times the maximum CPUID peak frequency reported by normal workloads.

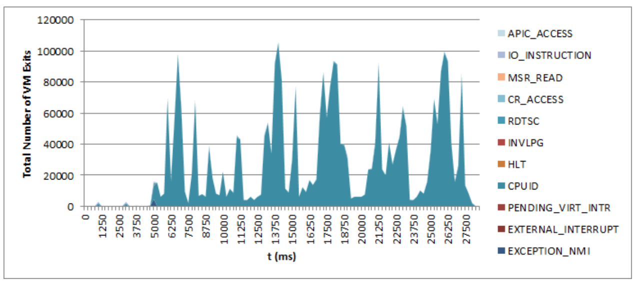


Figure 6: VM Exit traces of CPUID Attack

For comparison purposes, we also used a simple CPU Hog program, which is the typical application that leads to performance degradation in co-hosted VMs. Listing 2 shows the C language code corresponding to our CPU Hog (a simple single-threaded infinite loop).

```c
int main (void)
{
        while (1);
        return 0;
}
```

Listing 2: CPU Hog Code (C Language)

The CPU Hog's VM Exit trace (Figure 7) reports three main exit types: EXTERNAL_INTERRUPT, RDTSC and APIC_ACCESS. The total number of VM Exits per 250 ms interval is around 800, which is not remarkable. Therefore, any effect that this program has over a co-hosted VM will be attributed to the sustained CPU consumption and not to its characteristics in terms of VM Exits.

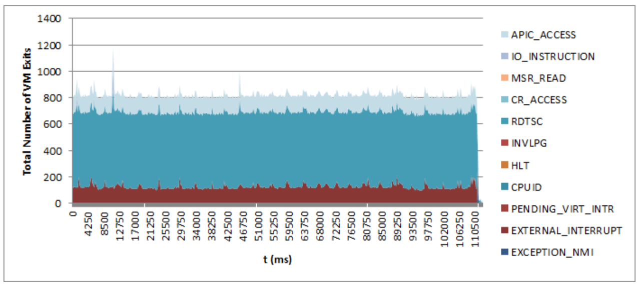


Figure 7: VM Exit traces of CPU Hog

We ran Freebench on our victim VM with the "-a3" option to find the best-of-3 times for various scenarios. As can be seen in Table 6, the CPUID flooding attack does indeed hinder the co-hosted VM's performance (by 22.62%). Also, the presence of 4 CPUID attackers results in more than 6 times the overhead generated by a single attacker, so there is initially some non-linear growth with respect to the number of attackers. On the other hand, rogue VMs running CPU Hogs have the exact same effect over the benchmark's execution time.

As evidenced by these tests, CPUID and CPU Hog, which have completely different VM Exit patterns, both succeed in stealing computational resources from a victim VM because of what they do have in common: an infinite loop. Looking back at Section 3.3, where we conclude that VM Exits do not account for more than 7% of the total execution time of a normal program, one can understand why flooding VM Exits does not drastically improve the chances of achieving a denial-of-service condition.

Table 6: CPUID Attackers vs CPU Hogs

Scenario	Freebench Time (s)	% Overhead
No Attacker	254.32	0.00
1 CPUID Attacker	311.84	22.62
1 CPU Hog	311.95	22.66
4 CPUID Attackers	624.64	145.61
4 CPU Hogs	628.97	147.31

5.1.1 *Special Scenarios in which CPUID Flooding can do more harm than CPU Hogging*

The CPUID Flooding technique may have the upper hand over the more simplistic CPU Hog technique in certain deployments. They had the same effect when running on our test bed because our Xen 4.0.1 hypervisor was sharing all of the available CPU cores among the co-hosted VMs.

Suppose that the hypervisor is configured so that it allocates a separate CPU core to each VM. If CPU Hog is run again in a rogue VM under such conditions, co-hosted victims would not notice any performance degradation because the excessive CPU consumption would not propagate to other cores. Meanwhile, the CPUID attack could still affect co-hosted VMs, since all VM Exits would still have to be handled by the hypervisor, which is a single entity shared among all co-hosted VMs.

5.1.2 *RDTSC*

To rule out any CPUID-specific factors from the reasons behind the lackluster results obtained with the CPUID Flood attack, we decided to employ the RDTSC instruction to create an RDTSC Flood Attack as well. The Freebench time with one RDTSC attacker was 290.01 seconds, which represents an even lower overhead than the one obtained with CPUID Flood. Thus, we confirmed that VM Exit flooding is not a strong strategy for denial-of-service attacks against co-hosted VMs.

5.2 CPUID Brute-Force Attack

The CPUID instruction still leaves room for denial-of-service attacks targetted at faulty implementations of the hypervisor's VM Exit Handler. Taking into account that the IA-32 CPUID instruction receives a 32-bit input value (unsigned int) through the EAX register, it is feasible to run a program that tests all possible input values with hopes that a bug similar to the one reported in [10] ends up crashing the hypervisor. Listing 3 shows the assembly language code for such a program. The call to printf is used to know what EAX value is currently being tested. The process starts with EAX=0x1 and increments the input by 1 unit until the 32-bit value overflows and becomes 0x0 right after being 0xffffffff.

```
  .section  .rodata
formatString:
  .string "%u\n"
  .text
.globl main
  .type main, @function
main:
  pushl %ebp
  movl  %esp, %ebp
  andl  $-16, %esp
  subl  $32, %esp
  movl  $0, 28(%esp)
loopIteration:
  movl  $formatString, %eax
  movl  28(%esp), %edx
  movl  %edx, 4(%esp)
  movl  %eax, (%esp)
  call   printf

  movl  28(%esp), %eax
  cpuid
  addl  $1, 28(%esp)

  cmpl  $0, 28(%esp)
  jne loopIteration
  movl  $0, %eax
  leave
  ret
```

Listing 3: CPUID Brute-Force Attack Code (AT&T Assembly Language)

We ran the program for several hours and tested every possible value of EAX. The end result: we did not detect any faulty behaviour in Xen's VM Exit Handler.

6. OTHER POSSIBLE ATTACKS

The VM Exit attack surface is considerably large, with 56 different exit types that can be exposed to vulnerability checks similar to the one described in Section 5.2. While we have limited ourselves to VM Exits that can be generated by user-space applications and have direct mappings to 64-bit and 32-bit Intel x86 instructions, we believe that out-of-the-box thinking can reveal at least one way to exploit VM Exits in order to alter the confidentiality, integrity, or availability guarantees of co-hosted VMs.

6.1 Kernel Mode Linux

As mentioned in Section 5, most VM Exits that directly translate to well-specified instructions require a privilege level of 0. Even though coding a rogue OS to run VM Exit attacks in kernel mode might be an option, it would definitely be an arduous task and changes to the attacks would most likely lead to full kernel recompiles. Fortunately, Toshiyuki Maeda [9] provides a way to patch linux kernels so that binaries placed in a specific directory run as part of Ring 0. This enables future proof-of-concept attacks involving any type of VM Exit.

7. COUNTERMEASURES

No VM Exit exploits were identified this time around. Nonetheless, in case VM Exits are proved to expose unpatchable vulnerabilities, the NoHype [7] approach is a sensible countermeasure. The NoHype architecture for Cloud Computing eliminates the hypervisor layer and places VMs directly on top of the physical hardware while still being able to run multiple VMs at the same time.

The focus of this paper are VM guests running on full virtualization mode (HVM mode in Xen), which require VM Exits due to the fact that the entire system (i.e. BIOS, HDD, CPU, NIC) is being emulated. Operating Systems can be left unmodified if running on HVM mode. On the other hand, another type of VM guests are those that are *paravirtualized*, which are presented with an interface that is closer (or identical) to the one of the real underlying hardware. Consequently, direct access to the shared system resources makes VM Exits unnecessary. Unfortunately, this low level of hardware abstraction is only compatible with kernels that have been modified to work with the specific para-API being exposed by the hypervisor [11]. While paravirtualization is a good alternative to avoid VM Exits, it leads to a closer control of the underlying hardware by the guest VMs, giving way to other security concerns.

8. RELATED WORK

VM Exits have received a considerable amount of attention from researchers interested in optimizing their execution times. For instance, Jian et al. [5] shortened the network I/O path by modifying the VM Exit handling of those I/O events and Wang et al. [14] proposed a mechanism to track VM Exits online so that their processing can be optimized in real time. The idea of VM Exits being a security threat is fairly recent and is yet to be formally examined.

Security-related research on hypervisors has resulted in ways of detecting the presence of specific hypervisors. Ferrie [3] [2] describes how a guest VM can detect (and even crash) popular hypervisors (e.g. BOCHS, VirtualPC, Parallels) running below it. Nonetheless, Ferrie's attacks that cause VMMs to crash do not rely on VM Exits; rather, they rely on glitches in the virtualized hardware triggered by specific instruction sequences. There has also been some significant research in the topic of VMM rootkits (BluePill[13], Vitriol[1] and SubVirt[8]), which migrate the running OS into a VM and then intercept accesses to hypervisor memory and hardware devices. The deployment methods of the three documented rootkits do not exploit VM Exits, but have to handle them appropriately after the hijacking occurs. Incidentally, when we went through the Xen 4.0.1 VM Exit-handling code, we noticed that the hypervisor protects itself against guest VMs trying to use VT-x extensions (e.g. VMCLEAR, VMLAUNCH, VMRESUME, VMXOFF, VMXON), which play a part in the Vitriol rootkit, by triggering INVALID_OP exceptions.

9. CONCLUSIONS AND FUTURE WORK

VM Exits remain relatively unexplored as a potential attack vector for malicious guest VMs in contexts like Cloud Computing, where co-hosted VMs are untrusted to one another. While we have made progress in deepening the understanding of VM Exits under normal workloads and have tried two different VM Exit-oriented denial-of-service attacks without success, there is still not enough information to come to the final verdict on whether VM Exits pose a significant security threat to fully-virtualized environments.

Future research on this topic could use Kernel Mode Linux, discussed in Section 6.1, to analyze the VM-Hypervisor communication channel for the various VM Exit types. It would also be interesting to have a test bed with other Type-I (bare metal) hypervisors besides Xen, such as KVM[5]. Last but not least, we invite theoreticians to investigate whether a VM Exit Handler like Xen's could be formally verified to prove its correctness.

10. REFERENCES

[1] D. A. Dai Zovi. Hardware virtualization rootkits. `http://www.theta44.org/software/HVM_Rootkits_ddz_bh-usa-06.pdf`.

[2] P. Ferrie. Attacks on more virtual machine emulators - symantec advanced threat research. `http://pferrie.tripod.com/papers/attacks2.pdf`.

[3] P. Ferrie. Attacks on virtual machine emulators - symantec advanced threat research. `http://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf`.

[4] Intel. Intel 64 and ia-32 architectures software developer's manual combined volumes:1, 2a, 2b, 2c, 3a, 3b, and 3c, Oct. 2011.

[5] Z. Jian, L. Xiaoyong, and G. Haibing. The optimization of xen network virtualization. In *Proceedings of the 2008 International Conference on Computer Science and Software Engineering - Volume 03*, CSSE '08, pages 431–436, Washington, DC, USA, 2008. IEEE Computer Society.

[6] P. A. Karger. Performance and security lessons learned from virtualizing the alpha processor. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 392–401, New York, NY, USA, 2007. ACM.

[7] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. Eliminating the Hypervisor Attack Surface for a More Secure Cloud. In *ACM Conference on Computer and Communications Security*, Oct. 2011.

[8] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. Subvirt: Implementing malware with virtual machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 314–327, Washington, DC, USA, 2006. IEEE Computer Society.

[9] T. Maeda. Kernel mode linux : Execute user processes in kernel mode. `http://www.yl.is.s.u-tokyo.ac.jp/~tosh/kml/`.

[10] P. Matousek. Bug 706323 - (cve-2011-1936) cve-2011-1936 kernel: xen: vmx: insecure cpuid vmexit. `bugzilla.redhat.com/show_bug.cgi?id=706323`.

[11] R. McCarty. Paravirtualization explained. `http://searchservervirtualization.techtarget.com/tip/Paravirtualization-explained`.

[12] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick. Xen 3.0 and the Art of Virtualization. In *Proceedings of the 2005 Ottawa Linux Symposium*, July 2005.

[13] J. Rutkowska. Subvirting vista kernel for fun and profit. `http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf`.

[14] X. Wang, B. Zhang, H. Chen, X. Jin, Y. Luo, X. Li, and Z. Wang. Detecting and analyzing vm-exits. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, pages 2273–2277, Washington, DC, USA, 2010. IEEE Computer Society.

[15] H. Wenlong. Bug 702657 - (cve-2011-1780) cve-2011-1780 kernel: xen: svm: insufficiencies in handling emulated instructions during vm exits. `bugzilla.redhat.com/show_bug.cgi?id=702657`.

[5]http://www.linux-kvm.org/