Detecting Hardware-Assisted Virtualization

Michael Brengel, Michael Backes, and Christian Rossow

CISPA, Saarland University

Abstract. Virtualization has become an indispensable technique for scaling up the analysis of malicious code, such as for malware analysis or shellcode detection systems. Frameworks like Ether, ShellOS and an ever-increasing number of commercially-operated malware sandboxes rely on hardware-assisted virtualization. A core technology is Intel's VT-x, which — compared to software-emulated virtulization — is believed to be stealthier, especially against evasive attackers that aim to detect virtualized systems to hide the malicious behavior of their code.
We propose and evaluate low-level timing-based mechanisms to detect hardware-virtualized systems. We build upon the observation that an adversary can invoke hypervisors and trigger context switches that are noticeable both in timing and in their side effects on caching. We have locally trained and then tested our detection methodology on a wide variety of systems, including 240 PlanetLab nodes, showing a high detection accuracy. As a real-world evaluation, we detected the virtualization technology of more than 30 malware sandboxes. Finally, we demonstrate how an adversary may even use these detections to evade multi-path exploration systems that aim to explore the full behavior of a program. Our results show that VT-x is not sufficiently stealthy for reliable analysis of malicious code.

1 Introduction

Malicious code continues to be a major security threat. The economics of cyber crime tempt attackers to improve their attacks in both quantity and quality. As such, analysts are confronted with a large number of sophisticated new attacks on a daily basis. This sheer volume of threats renders manual analysis impractical, which is why defenders seek to *automate* the analysis of potentially malicious code. In terms of automated analysis, defenders usually prefer *dynamic* over *static* analysis, since malware is usually heavily obfuscated [18, 21, 31]. While ideas exist to cope with this problem [4,11,12,25,27,28], in practice, a satisfying notion of static code analysis automation is still far from being established.

Dynamic analysis executes unknown programs in a controlled environment and monitors this execution to look for malicious behavior. The large number of dynamic malware analysis systems demonstrates their utility [6]. The security industry has also taken up the concept of malware sandboxes and one can choose from a variety of open-source and commercial systems, such as Cuckoo [19], Joe Sandbox, GFI Sandbox, VMRay or FireEye.

Realizing the benefits of dynamic analysis, attackers started to evade sandboxes. Evasion enables an attacker to discover that the execution takes place in a controlled environment and to suppress any malicious behavior because of this insight. In its simplest form, evasion leverages the fact that most dynamic code analysis systems use some kind of virtualization solution and that these solutions usually come with artifacts such as specific device drivers or known hardware serial numbers, for example. By probing for those artifacts, the attacker can detect the analysis system and suppress any malicious behavior. Note that this approach relies on the assumption that virtualized code execution is equivalent to dynamic code analysis. While this is not generally true, e.g., due to cloud computing, typically attack targets can be assumed to operate on native systems. Virtual machine (VM) detection approaches are widely popular among attackers in the wild. Malware (e.g., the families Dyre, CryptoWall, Shifu, Kronos and Shylock) hide their actual behavior if they are executed in a VM.

When it comes to the choice of virtualization solution, defenders usually build upon hardware-assisted virtualization techniques such as Intel VT-x or AMD-V. Besides being faster due to hardware support, hardware-assisted virtualization also greatly reduces the number of artifacts, giving the attacker less room for simple evasion. Analysis systems such as Ether [5] and CXPInspector [30] use VT-x to analyze malware in a transparent manner. The authors of Ether have shown that malware that does not show behavior on other software-virtualized systems suddenly becomes active in their systems, which highlights the importance of hardware-assisted virtualization.

In this paper, we therefore aim at a more generic form of evasion. While an artifact indicates that the system might be virtualized, there is no semantic connection between the presence of an artifact and the concept of virtualization. Instead, we follow the intuition that virtualized guests operate more slowly than native systems. To this end, we propose three timing-based and assembly-level mechanisms to detect hardware-assisted virtualization *from user space*, i.e., without using privileged instructions. We first consider measuring the execution time of special x86 instructions, that cause a trap to the hypervisor, which will not occur during native execution. We then discuss how a hypervisor might try to hide the involved timing artifacts, and propose a second detection technique which exploits the Translation Lookaside Buffer (TLB) and which cannot be protected against in the same fashion. We leverage timing differences during accesses to pages whose address translations have been cached in the TLB, but whose cache entries a hypervisor has evicted due to the limited size of the TLB. We then consider the stealthiness of those two approaches. Given that the first two methods use special instructions excessively, we argue that those methods are not stealthy. Therefore, we propose a third method which is stealthier in that it limits the use of possibly suspicious instructions and resorts to a different timing source.

We evaluate our methods on a large variety of native and VT-x-based systems and show that the detection methods have a 99.4% true positive rate on average. We then turn to a few practical use cases. First, we deploy our detection routine

on 31 public malware sandboxes, all of which we can detect using the described methods. Second, we demonstrate that even a commercial sandbox with anti-evasion features falls for our caching-based detection mechanism. Finally, we show how an adversary may combine the detection results to evade multi-path exploration systems and demonstrate this with the use case of ShellOS [29], a VT-based shellcode detection framework.

2 Background

2.1 Hardware Virtualization

Most malware analysis systems that use hardware-assisted virtualization rely on Intel VT (or VT-x) as an underlying virtualization technique. With VT-x, the Virtual Machine Monitor (VMM, or *hypervisor*) can create and launch multiple virtual machines. Once the VM is running, the guest can return to the hypervisor; this is called a *VM Exit*. The guest can explicitly invoke the exit handler of the hypervisor, e.g., to establish communication between the host and the guest. After the hypervisor has performed the desired operations, control can be returned to the guest; this is called a *VM Entry*.

In addition to the *explicit* calls to the exit handler, the hypervisor also *implicitly* traps in certain occasions. We will use exactly these implicit traps as part of our timing side-channels, and thus briefly explain them in the following. For example, VM Exits are implicitly caused if the guest executes a sensitive instruction. This behavior is crucial, since it gives the hypervisor the chance to emulate important data structures and monitor the VM. Intel specifies all such VMM trap instructions in their manual. Since VM Exits are an inherent difference between virtualized and native executions, we use them as a way of detecting the presence of a VT-x hypervisor. While VT-x offers the possibility to disable traps for some instructions, Intel *enforces* VM Exits on a selected set of instructions in hardware. We also argue that in order to monitor the guest, the hypervisor has to use some kind of traps, which also gives additional space for evasion.

2.2 Translation Lookaside Buffer

One of the side effects of hypervisors that we will use targets the TLB, as outlined in the following. Modern operating systems use the concept of virtual memory to give each running process the impression of having a large contiguous space of memory, whereas in reality there is a mapping between virtual and physical memory in a non-contiguous manner. Resolving this mapping is called a *page walk*. Since a page walk can be costly, hardware developers introduced the TLB, which caches the mapping from virtual pages to physical pages. When a process accesses a virtual address v, it first checks if the virtual page of v is in the TLB. If it is, the physical address can be obtained from the TLB. Otherwise, the MMU needs to do a page walk and then caches the result in the TLB. Therefore, when

accessing multiple addresses on the same page, it is likely that only the first access is slow relative to all subsequent accesses, as long as the TLB entry is not evicted in the meantime.

If we switch from a process p_1 to a process p_2 (context switch), the TLB will most likely contain invalid entries, since p_1 and p_2 have their own virtual memory and thus use different address space mappings. The simplest way to deal with this problem is to completely flush the TLB upon context switches. However, this strategy had severe performance penalties, as every VM Exit causes a context switch. To cope with this problem, Intel introduced a feature called VPID. With VPID enabled, the TLB entries are tagged with the ID of the virtual machine. As a consequence, when a virtual address is accessed, the MMU will only consider TLB entries which are tagged with the current VM's ID. Hence, there is no need to flush the TLB, resulting in better performance.

3 Threat Model

Throughout the remainder of this paper, we envision an adversary that has implemented an arbitrary program (e.g., malware) and tries to detect if this program is being executed in a virtualized environment. We aim to explore generic evasion attempts, i.e., those that (i) do not focus on particular analysis environments, but instead on inherent characteristics of such systems, (ii) an approach that is independent from the malicious payload that the adversary may aim to hide, and (iii) mechanisms that are not restricted to a certain operating system. All these requirements make our methods applicable to a wide set of programs, explicitly including typical malware targeting Windows or Linux.

Furthermore, we restrict ourselves to developing detection mechanisms that operate purely in user mode, i.e., unprivileged code that executes in ring 3. This assumption varies from existing approaches that aim to detect virtualization by using privileged instructions, such as reading the contents of the page tables or using nested virtualization. Approaches using privileged code (ring 0) are well known to be effective, but may raise suspicions to an analyst or even during automated program analysis. In contrast, our assumption on user-mode execution is in line with the use case of in-the-wild malware, such as a myriad of banking trojans, droppers, clickbots, spambots, denial-of-service bots and even targeted malware—all of which typically run in user space. The most notable exceptions are malware families with a kernel-mode rootkit, which, however, could also use our proposed user-mode detection methods. For example, a user-space dropper could try to detect virtualization prior to installing further modules (such as kernel-mode rootkits), such that the second- or third-stage malware samples are not exposed to the analyst. In fact, this concept is common in the wild [15, 26].

Finally, we assume that the actual target systems of an attacker, i.e., those that the attacker aims to infect, are *not* virtualized. While it is conceivable that an attacker may miss target systems that are indeed virtualized, widespread malware will still be successful in infecting the vast majority of native systems.

4 Timing-Based VT-x Detection

VT-x was invented with the goal to increase the performance as well as the transparency of virtualization. In this section, we aim to undermine that transparency by proposing three timing-based methods to detect virtualization.

4.1 Measuring Elapsed CPU Cycles

The first two proposed detection methods are based on a technique to accurately determine the execution time of machine code. To this end, we measure the number of CPU cycles that elapse when a piece of code is executed. To do so, we use the `rdtsc` instruction to read the CPU's time stamp counter (TSC), which is incremented every clock cycle. Upon execution of `rdtsc`, the `edx` register is loaded with the high-order 32 bits and `eax` holds the low-order 32 bits of the TSC. Reading the TSC before and after the machine code helps us to measure the number of cycles that have elapsed. We can thus execute `rdtsc`, save the TSC, execute the instructions to be measured, execute `rdtsc` again and subtract the saved TSC from the current TSC. To get more accurate results, we need to serialize the instruction stream to prevent out-of-order execution. We use the `mfence` instruction, which will serialize `rdtsc` with respect to load-from-memory *and* store-to-memory instructions.

This method over-approximates the execution time. This is a bias introduced by the measurement code, which also consumes CPU cycles. If necessary, to counteract this influence, we can measure the measuring overhead and subtract it from the measured time. To this end, we measure how long it takes to execute no code. We then subtract this overhead from subsequent measuring results to get a more realistic measurement of the actual clock cycles. We will use this technique for our implementation of a TLB-based VT detection that demands a higher measuring accuracy.

Finally, measurements may not be accurate due to context switches that occur during the measurement phase, in which another process would execute and implicitly increase the TSC. To tackle this problem, we repeatedly measure the same sequences of instructions, record the time of each execution, and then use the *minimum* of all measurements. Our assumption here is that at least one out of these many executions will not be clobbered by a context switch.

4.2 Method 1: Detecting VM Exit Overhead

Based on the timing measurements, we will now describe our first method to detect hardware virtualization. We follow the intuition that a VM Exit consumes CPU cycles. In particular, we leverage the fact that some CPU instructions provoke a VM Exit, which does *not* occur on native systems. The Intel manual specifies over 30 of such instructions, most of which are privileged and thus not usable in ring 3. Also, trapping of some instructions can be disabled during the VM setup phase. We will thus leverage `cpuid`, the only unprivileged instruction whose hypervisor traps cannot be disabled.

A naïve approach to detect VT-x would be to measure the execution time of a `cpuid` instruction and determine a threshold above which VT-x is detected. The threshold could be determined by comparing the execution time on native and virtualized systems. Whenever executing `cpuid` is below this threshold, the environment is believed to be native, and virtualized otherwise. However, this simple measurement suffers from the inaccuracy that the absolute execution time of instructions varies per CPU type, and it is thus not trivial to determine a reasonable absolute threshold. Instead, we follow the intuition that two different instructions execute similarly slower or faster on a different CPU model. We thus propose to compare the execution time of `cpuid` with the execution time of another instruction in terms of a ratio.

We proceed as follows: We measure both `cpuid` and a baseline instruction i times, compute the ratio and check if the ratio is larger than some threshold α. We chose `nop` as the baseline instruction, but found that other fast instructions are equally suitable (e.g., we experimented with `add`, `lea`, and `sub`). Given the short execution time of the baseline instruction, the measurement overhead dominates our measurements (as described in Section 4.1). Still, the execution time of `cpuid` is significantly higher than the baseline instruction. Thus, we decided not to deduct the overhead. To account for context switches, we compare these two minima and return true if the ratio r exceeds our threshold α, which will be determined in Section 5.

4.3 Method 2: Detecting TLB Evictions

The first method is susceptible to hypervisors that manipulate the TSC value. For example, upon each `cpuid` trap, the hypervisor can deduct the time that it spent in the exit handler from the value returned to `rdtsc`, which effectively evades our first detection method. We will now describe a second method, which does not fall for this evasion. Our second idea is again to detect whether the hypervisor was invoked, though this time by inspecting cache evictions in the TLB.

The idea behind our detection is to identify TLB entry evictions that were caused by VM Exits. The intuition is that the hypervisor needs to execute code and has to access data (e.g., the VM Exit reason), all of which requires memory translations. The CPU will cache these memory translations in the TLB. This inevitably changes the TLB, which we aim to detect. To this end, we allocate fresh memory pages and access them, e.g., using a read operation. This precaches the pages to the TLB. We then iterate twice over the pages and measure the time to access the pages. However, in the second iteration, prior to reading, we execute a `cpuid`. This will trigger a VM Exit, causing a context switch, evicting entries of the pre-filled TLB. In such a case, we can measure a notable difference in access times when revisiting the evicted page.

The details of the detection are as follows: After allocating n pages, we access all pages once to fill the TLB. We access a page by writing a 4-byte value to the start of the page. We then access all pages again i times, and for each iteration we record the maximum access time. Out of those maxima we then compute

the minimum. The intuition behind this is that we want to find the minimum time that it took for the slowest of all n page accesses. Using the minimum also eliminates outliers. In addition, compared to similar measures like the median, computing the minimum has a relatively small assembly code size and a lower computational complexity. Then, we repeat this step, but execute `cpuid` before each memory access. To set up equivalent conditions, we also access each page once before entering the second outer loop. Finally, the ratio of both minima t_0 and t_1 is computed and compared against a threshold β. Again, we refer the reader to the evaluation for choosing the detection threshold.

A few implementation details require attention. First, after accessing a page, we use the `clflush` instruction to invalidate the corresponding cache lines from all levels of the processor cache. Doing so makes it more likely that timing discrepancies actually stem from the TLB (and not from other types of caches). Second, when we measure the page access time, we subtract the overhead of the measuring method. We do this since the difference in access time for TLB misses tends to be rather small and the influence of measurement overhead would thus be too high otherwise.

Finally, we need to know how many pages we should allocate. The difficulty here is that we have to prefill approximately as many pages as will fit in the TLB. On the one hand, we need enough pages to fill the TLB and to guarantee that the hypervisor evicts our TLB entries. On the other hand, choosing too many pages will itself cause TLB evictions in all loops, regardless of `cpuid`, and therefore undermine our detection. However, the exact number of pages is unpredictable and depends heavily on the TLB size and its page eviction strategy. To counter this problem, we will perform the detection multiple times with various page numbers and consider a system to be running VT-x if at least one detection finds a suspicious ratio. That is, we chose to run the algorithm for $n = 64, 128, \ldots, 4096$ pages, which covers typical TLB sizes.

4.4 Method 3: Stealth VM Exit Detection

The two aforementioned methods have the drawback that they make excessive use of two instructions (`cpuid` and `rdtsc`) that may not be as common in benign user-mode programs. This may raise suspicion in the analysis system, automatically flagging the program as malicious. Therefore, we advance the current state of research by studying *stealth* timing-based mechanisms, i.e., ones that do not use suspicious instructions. To this end, we (i) have to avoid excessive use of `cpuid` to trap the hypervisor, and (ii) require a reliable but less suspicious and non-modifiable source for timing information.

To invoke the hypervisor, we first inspected less suspicious alternatives to `cpuid`. However, the alternatives are either privileged or their VMM traps are optional. Thus, we explore detection methods that *conservatively* use `cpuid`. Note that `cpuid` is not suspicious *per se*. However, calling `cpuid` thousands of times—as done in our first two methods—may seem suspicious. As we will show, our discovered method will not trigger more than 3–4 `cpuid` calls on virtualized systems, which makes this behavior indistinguishable from benign usage.

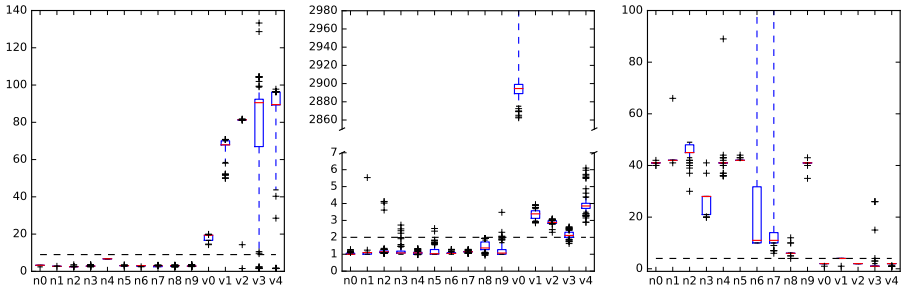
Replacing the timing source (`rdtsc`) is trickier, as reliable and accurate timing sources are scarce and only accessible with privileged instructions. To measure the duration of a `cpuid` instruction, we therefore leverage parallel computation. Technically, we create a program which spawns a thread B in its main thread A. B continuously executes `cpuid` and increments a counter c after each execution. After spawning B, A performs a busy wait until it notices that B has started by examining c. After that, we wait a small amount of time by executing a short busy loop after which we can inspect c. We determine that the system is virtualized if the counter is smaller or equal than some threshold γ, following the intuition that more `cpuid` executions fit in the time window on a native system without context switches to they hypervisor. We will see in Section 5 that the number of `cpuid` instructions being executed will be fairly low for virtualized systems, which greatly contributes to the stealthiness of the detection

5 Evaluation

5.1 Local Experiments

The accuracy of our VT-x detection methods greatly depends on reasonable detection thresholds. To determine the thresholds, we conducted the following experiment: We created a tool that measures the ratio of the execution time of `cpuid` to the execution time of `nop`. In addition, we measured the ratio of page accesses as described. To account for different TLB sizes and eviction strategies, we measured this ratio for $n = 64, 128, 256, \ldots, 4096$ pages, where we use the maximum of those ratios as the final ratio to on which base the detection. We measure the access time of `nop` and `cpuid` 1000 times and take the minimum, and the page accesses are measured 500 times for each page size. Finally, we also measure the number of times `cpuid` was executed during the busy loop of method 3. To create realistic conditions, we executed the tool 100 times each on ten native Windows/Linux systems. The systems used nine distinct CPUs. In addition, we executed the tool in 5 different Windows/Linux VMs using various hypervisors running on Windows, Mac OS and Linux.

Method 1: VM Exit Overhead (M1) Figure 1a shows the `cpuid/nop` ratio. The x-axis shows the hosts and the y-axis the distribution of the ratios in a boxplot. The boxes show the 25–75% intervals, the whiskers indicate 10–90% intervals. For nine out of the ten native hosts (n0–n9), the ratio is between 2 and 4. Host n4 is an outlier, with its ratio being constantly about 6.7. Hosts 11–15 represent VMs (v0–v4). The majority of the ratios of the VMs are significantly larger than the ratios of the native hosts. There is quite some variation among the different VMs. This can mainly be attributed to the hypervisor. For example, VMs 1 and 2 were running on the same physical machine; VM 1 was using VMware, whereas VM 2 was using VirtualBox. Given those results, we choose the threshold $\alpha = 9$ as indicated by the horizontal dashed line in the figure. Everything above this line indicates a VM, whereas everything below this line is



(a) M1 *(b) M2* *(c) M3*

Fig. 1: *Boxplots of the local experiments. Hosts n0–n9 are native; hosts v0–v4 are virtualized. The horizontal lines show the detection threshold that we derived from these local experiments:* $\alpha = 9$ *(left),* $\beta = 2$ *(center) and* $\gamma = 4$ *(right).*

determined to be a native machine. We chose the threshold to tune the detection rates of the local experiment towards a low false positive rate: Out of the total of 1000 runs of the tool on the 10 native systems, we detect all systems as native systems, which gives us a true negative (TN) rate of 100% and thus a false positive (FP) rate of 0%. The executions on the VMs resulted in 17 ratios being below 9, which implies a true positive (TP) rate of 96.6% and a false negative (FN) rate of 3.4%. An adversary may adapt the detection thresholds depending on her goal. In particular, it is reasonable for an attacker to trade off false positives for false negatives, since losing a single victim is less critical than having the malware exposed in an analysis environment.

Method 2: TLB Eviction Overhead (M2) Figure 1b shows the computed ratios of the TLB-based measurements. As we can see, page accesses are more prone to variance than the previous measurements. This can be explained by the fact that the time window in which context switches would bias the measurement is significantly longer for the second method. While in the first method a context switch is only critical during the `cpuid` measurements, the second method assumes that the process is not interrupted during the entire measurement period, as otherwise the cached TLB entries would be overwritten (even on native systems). We will discuss how the system load may influence the measurements in Section 6. That being said, we see that for native hosts the median of the ratios is still about 1 and the majority of the ratios is in the range $[0.9, 2]$.

Similar to the previous results, we can again see a clear difference between native and virtual machines, although the effect is smaller this time. This is caused by the fact that a TLB miss is rather cheap compared to a context switch caused by a VM Exit. VM v0 is an outlier, with the ratio being constantly larger than 2600. We suspect that this is caused by some VMware artifact, as the ratios of the other hypervisors are three orders of magnitude smaller. Using these results, we set the threshold $\beta = 2$ for M2. That is, out of the 1000 native

runs we misclassify 17 runs, resulting in a TN rate of 98.3% and thus a FP rate of 1.7%. On the VMs, we detect 24 out of 500 runs as native, which results in a TP rate of 95.2% and a FN rate of 4.8%.

Finally, the attacker could combine both of the above methods and determine that a system is virtualized if at least one of the methods says so. By doing this, we would misclassify only 3 out of 500 virtualized executions, decreasing the FN rate to 0.6%. Note that this would not change the FP rate, as only the TLB detection produced FPs.

Method 3: Counting Thread (M3) Finally, we evaluate our stealth method M3 separately from the non-stealth methods M1 and M2. Figure 1c shows the results of the thread counting measurements, where the y-axis indicates how often `cpuid` was executed. We can observe that the counter is much larger for the native hosts as the `cpuid` instruction is faster in that case. In contrast, the virtual hosts execute `cpuid` only a few times due to the significant overhead caused by the VM Exit. Using the gathered results, we determine the threshold $\gamma = 4$ as indicated by the horizontal line. Everything above this line is considered a native system and everything below it is detected as a virtualized system. This yields a TN rate of 99.9% (n8 fails the test once) and a TP rate of 99.2% (v3 fails this test 4 times).

5.2 Evasion Detection

Although the ratios of methods M1 and M2 work quite well in a normal setting, they may be biased if virtualized sandboxes hide timing artifacts to evade VM detection methods. For example, a hypervisor could trap `rdtsc` and set the TSC to a constant c before returning to the guest. The ratio for both M1 and M2 would then always be 1, which hides the hypervisor from our detection methods. To detect such anti-detection methods, we add sanity checks on top of our detection. In the following, we will describe four sanity checks, which we will use throughout the remainder of the evaluation for the `rdtsc`-based methods. The thresholds of these sanity checks were derived from the absolute values gathered from our local experiment in Section 5.1.

Measuring overhead: First, we execute the measuring method without any instructions to get the measuring overhead. We will use these results to detect costly `rdtsc` traps. The measurements showed that `rdtsc` never takes more than 100 cycles to execute (including the overhead introduced by `mfence` and moving the TSC into a temporary register). To account for variance, we relax this threshold and say that the system is virtualized if executing `rdtsc` takes 500 or more cycles.

Execution time of `cpuid`: Second, we inspect the absolute execution time of `cpuid` to get a lower bound in order to detect if a hypervisor returns suspiciously low values. Given the results of the measurements, we determine that a hypervisor tampers with the TSC if executing `cpuid` takes less than 20 cycles.

Execution time of `nop`: Third, we verify if a hypervisor returns suspiciously high values for the execution time of a `nop` instruction. Given the results, we determine that a system is virtualized if executing a `nop` instruction takes 500 or more cycles.
TLB delta: Fourth, the execution time overhead introduced by a TLB miss tends to be rather small compared to the time of a cached page access. The ratio is thus sometimes not sufficient to detect a hypervisor, especially if both values are rather large. To tackle this problem, we additionally inspect the absolute difference between t_0 and t_1 in M2. We say that a system is virtualized if $r \geq \beta$, or if $t_0 - t_1 \geq 150$, a carefully chosen threshold. We observed that only five runs resulted in a difference larger than 150. However, in those runs the ratio was above 2, which means that we will not change the results for the local experiments by including the TLB delta in the detection. Compared to the previous three thresholds, this one was chosen rather aggressively. We did this, since the previous parameters are used only to detect trivial anti-detection methods and for this purpose a conservatively chosen threshold is sufficient. However, in the case of TLB-based detection the window for optimization gets quite narrow, which is why we chose a more aggressive optimization. We refer to this optimized method as M2'.

5.3 PlanetLab

We then deployed our detection on a wider variety of systems. To this end, we identified 239 stable nodes in the PlanetLab [3] testbed. To check if a PlanetLab node is virtualized, we verified if the hypervisor bit returned by `cpuid` (with leaf 1) is set. Although a stealthy hypervisor does not have to set this bit, we find it unlikely that stealthy hypervisors are present in PlanetLab. We found that 233 of the 239 nodes are native and six are virtualized. On all hosts, we performed our detection methods 10000 times each, with $\alpha = 9$, $\beta = 2$, $\gamma = 4$ and $i = 1000$.

M1 We detected 232 out of the 233 non-virtualized nodes as native systems with 100% confidence. On the remaining native node, 3 out of the 10000 runs misclassified the node as virtualized. Overall, this gives us a true negative rate of 99.99%. All of the six virtualized nodes were correctly classified as virtualized with 100% confidence, which gives us a true positive rate of 100%.

M2 We detected the non-virtualized nodes as native systems with an overall true negative rate of 99.96%. However, the detection rate for 7 out of those 232 nodes was below 99% and as low as 73.1% for one node. We suspect that this is due to high system load, since concurrent memory accesses cause TLB evictions, which does not favor M2. Out of the six virtualized nodes, we detected 3 nodes with 100% confidence and failed to detect the remaining three virtualized nodes. We discovered that this is because our detection assumes a page size of 4 KiB, which was not true for the PlanetLab nodes in question as they are using the Page Size Extension feature of x86, which allows for pages larger than the traditional 4 KiB limit for performance reasons. This is not a restriction for an adversary in

practice, since huge pages are not enabled by default in Windows user space. While huge pages are used by recent Windows kernels, enabling huge pages in user-space processes on Windows requires the "lock pages in memory" privilege.

M3 For M3, we had to use a different set of 185 stable nodes in the PlanetLab testbed, since the nodes of the first two methods were not available any longer when the experiments for the third method was conducted. Out of those nodes, 182 were native and 3 nodes were virtualized. We detected both the virtualized and the native systems with a confidence of 100% giving us a true positive and a true negative rate of 100% each.

5.4 Sandboxes

So far we have only tested our methods in a controlled environment. In the following, we will apply our detection methods in real-world use cases that are relevant in practice: malware sandboxes. We thus uploaded the detection tool (cf. Section 5.1) to 17 malware analysis services, such as VirusTotal and ThreatExpert. Despite the fact that bare-metal sandboxes have been proposed [13,14], we argue that most automated sandboxes are virtualized to cope with the masses of user submissions. We thus assume that a sandbox uses VT-x, provided that the reported CPU brand string indicates a processor model which supports hardware-assisted virtualization.

To collect the detection results, we configured the tool to send the results to a server under our control. In total, we received 76 measurement reports. We did not have any influence on how often our sample was run per sandbox. Some of the sandboxes repeatedly executed the same sample, i.e., increasing the number of reports. In total, 74 of 76 reports indicated a processor model which supports VT-x. The remaining two reports came from the same sandbox and showed the CPU brand string "Intel(R) Atom(TM) CPU D525 @ 1.80GHz". For this sandbox, none of our detection methods indicated the presence of a hypervisor. We group the remaining reports based on the host names of the analysis systems, which clusters these 74 reports into 31 sandboxes. For a detailed overview of the results, see Figure 2.

We observe that method 1 detects 89.19% of the sandboxes. The improvements for method 2 increased the detection rate from 14.86% to 60.81%. By combining method 1 and method 2—that is, we detect a system if M1 or M2 detects it—we can detect 90.54% of the sandboxes. By combining M1 with the improved version of M2, we can improve again and get a detection rate of 95.95%. This shows that the proposed methods work well in a real-world setting.

A few sandboxes remain undetected, which is where the sanity checks help. The sanity checks themselves have a rather low detection rate of 18.92%. Worst, the `cpuid` sanity check has a detection rate of 0%. But the `nop` and `rdstc` measures alone are also not very helpful. However, recall that the sanity checks are mainly relevant for systems that tamper with the TSC to evade detection. When used in combination with the proposed two methods, they significantly improve the detection rate. By combining the sanity checks with M1 and the

Sandbox	M1	M2	M2'	M1+M2	M1+M2'	RDTSC	CPUID	NOP	S	S+M1+M2'
s_0	5/5	2/5	3/5	5/5	5/5	0/5	0/5	0/5	0/5	5/5
s_1	10/10	2/10	5/10	10/10	10/10	0/10	0/10	0/10	0/10	10/10
s_2	4/4	0/4	4/4	4/4	4/4	0/4	0/4	0/4	0/4	4/4
s_3	2/2	0/2	0/2	2/2	2/2	0/2	0/2	0/2	0/2	2/2
s_4	1/1	0/1	1/1	1/1	1/1	0/1	0/1	0/1	0/1	1/1
s_5	1/3	0/3	0/3	1/3	1/3	0/3	0/3	3/3	3/3	3/3
s_6	1/1	0/1	1/1	1/1	1/1	0/1	0/1	0/1	0/1	1/1
s_7	3/3	1/3	1/3	3/3	3/3	0/3	0/3	0/3	0/3	3/3
s_8	1/1	0/1	0/1	1/1	1/1	0/1	0/1	1/1	1/1	1/1
s_9	3/3	0/3	2/3	3/3	3/3	0/3	0/3	0/3	0/3	3/3
s_{10}	6/6	0/6	3/6	6/6	6/6	0/6	0/6	0/6	0/6	6/6
s_{11}	1/1	0/1	1/1	1/1	1/1	0/1	0/1	0/1	0/1	1/1
s_{12}	1/1	1/1	1/1	1/1	1/1	0/1	0/1	0/1	0/1	1/1
$s_{13} \dots s_{17}$	3/5	0/5	3/5	3/5	4/5	4/5	0/5	4/5	4/5	5/5
s_{18}	6/6	0/6	4/6	6/6	6/6	0/6	0/6	0/6	0/6	6/6
s_{19}	3/3	0/3	1/3	3/3	3/3	0/3	0/3	0/3	0/3	3/3
s_{20}	1/1	1/1	1/1	1/1	1/1	0/1	0/1	1/1	1/1	1/1
s_{21}	3/3	0/3	1/3	3/3	3/3	0/3	0/3	0/3	0/3	3/3
s_{22}	1/1	0/1	1/1	1/1	1/1	0/1	0/1	0/1	0/1	1/1
s_{23}	2/2	2/2	2/2	2/2	2/2	0/2	0/2	0/2	0/2	2/2
s_{24}	0/2	1/2	2/2	1/2	2/2	2/2	0/2	2/2	2/2	2/2
s_{25}, s_{26}	2/2	0/2	1/2	2/2	2/2	0/2	2/2	2/2	2/2	2/2
s_{27}	1/1	1/1	1/1	1/1	1/1	0/1	0/1	0/1	0/1	1/1
s_{28}	1/3	0/3	3/3	1/3	3/3	3/3	0/3	3/3	3/3	3/3
s_{29}	3/3	0/3	3/3	3/3	3/3	0/3	0/3	0/3	0/3	3/3
s_{30}	1/1	0/1	0/1	1/1	1/1	0/1	0/1	0/1	0/1	1/1
	66/74	11/74	45/74	67/74	71/74	9/74	0/74	14/74	14/74	74/74
	89.19%	14.86%	60.81%	90.54%	95.95%	12.16%	0.00%	18.92%	18.92%	100.00%

Fig. 2: *Detection results per sandbox (1st major column) for both methods, the improved version of method* 2 *and their combinations (2nd major column), the sanity checks (3rd major column) and the overall combination (4th major column). Sandboxes* $s_{13}, \dots, s_{17}$ *and* s_{25}, s_{26} *are aggregated due to similar hostnames*

improved version of method 2, we can detect all 74 hosts, resulting in a true positive rate of 100%.

As the observed sandboxes are single-core VMs, there was no possibility to evaluate M3 using this dataset. Instead, we successfully detected these hosts as single-core systems using a simple load detection. In short, the load detection method leverages the fact that the OS cannot schedule two threads in parallel on single-core systems (or on systems with a high load). The load detection method thus schedules two threads in parallel and measures how many rescheduling events occur in a given time span. Rescheduling is an indicator for (i) high load or (ii) a single-core system. Although only allowing multi-core systems seems to restrictive at first, it is worth noting that this is also done by malware in practice to avoid sandboxes. For example, the popular banking trojan Dyre shows its real behavior only if executed on multi-core systems.

In conclusion, with high reliability, we could detect the virtualization in sandboxes using our timing-based methods. This demonstrates how a malware author may use similar techniques to hide the actual behavior of the malware.

5.5 Stealth Sandbox

Seeing our success in identifying sandboxes, we now test whether cutting-edge commercial sandboxes can evade our detection. We contacted a popular vendor that advertises their sandbox technology as being evasion-resistant. The sandbox is based on VT-x and implements a thin hypervisor. The vendor graciously allowed us to run our detection sample in their latest development version of the sandbox. We supplied the same tool to the system as for the other experiments.

In the first experiment, we managed to detect the presence of a hypervisor with M1 and M2'. We then interacted with the vendor and explained our detection methodologies. It turns out that their anti-evasion tricks mainly focused on removing system artifacts, and did not yet counter timing attacks. The vendor then implemented a countermeasure to tackle timing-based detection. Technically, the improved hypervisor measures its own execution time and then subtracts its overhead from the actual TSC. VT-x features the `TSC_OFFSET` parameter so that the hypervisor can adjust the TSC accordingly—an anti-evasion technique that was mentioned by Dinaburg et al [5].

We then re-ran our experiment on the updated sandbox. This time, the timing-based detection failed to detect VT-x. In fact, the measured ratios in M1 looked quite similar to the ones measured on native systems, rendering M1 ineffective. However, M2' could still identify the virtualization. This is also intuitive, as the anti-evasion technique does not take into account overhead caused by caching. M2' does not measure the time spent in the hypervisor, but instead measures caching side effects of context switches.

Similar to the public sandboxes, the commercial sandbox can also be detected using M3 in combination with the load detection mechanism, as the sandbox operates on a single-core system. In fact, our load check is more reliable and cannot easily be fooled, unlike to existing methods such as reading the number of cores using the Windows PEB. Thus it presents a generic detection method for single-core systems.

5.6 ShellOS

Finally, we turn to systems that aim to detect potential shellcode in arbitrary code. Snow et al. proposed ShellOS [29], a VT-x-based framework that deploys heuristics to detect shellcode. ShellOS uses a small kernel coupled with hardware-assisted virtualization by using the KVM hypervisor to detect and analyze shellcode. To this end, ShellOS uses the heuristic proposed by Polychronakis et al. [23], monitoring and detecting accesses to the Windows PEB. We will evaluate whether we can combine our VT-x detection with an evasion for ShellOS. Our goal is to convert a shellcode that ShellOS flags as malicious to one that is not detected. The obvious idea in the form of `if (vm) {benign} else {shellcode}` will, however, not work, since ShellOS

```
push MAGIC
; is_vm := vm_detect()
if (not is_vm)
    pop edx
mov esi, [fs:edx+0x30-MAGIC]
...
```

Fig. 3: *ShellOS evasion.*

uses the concept of *execution chains*. ShellOS will execute the payload from each possible offset and will therefore flag the payload as malicious if it starts executing from the `shellcode` offset. To understand how an adversary may overcome this problem, recall that Windows shellcode usually starts by reading the PEB with an instruction like `mov esi, [fs:edx+030]` where `edx` is set to zero beforehand. The malicious behavior of all the following instructions depends on this first instruction, which in turn depends on the content of one register (`edx` in the example). Using this insight, we can evade ShellOS's execution chains with the idea illustrated in Figure 3. We push some magic random number `MAGIC` > 0 on the stack, and only if the detection indicates the presence of a native system, we pop this magic number into `edx`. We then use this magic number in the calculation of the PEB in the first instruction of the shellcode. If ShellOS executes the code from any offset other than the actual start, the `edx` register will not contain the right value to correctly calculate the PEB offset. If ShellOS starts executing from the actual start, our detection will detect VT-x and the code will not be flagged as malicious, since the PEB will not be calculated correctly in this case either.

We injected the VM Exit-based detection method (M1) into the template in Figure 3 with parameters $i = 20$ and $\alpha = 9$. We reduced the number of iterations to 20, since ShellOS imposes timeout limits on the payloads to be executed. We injected several shellcodes with our detection methods, but we intentionally left out the determining ratio check to provoke malicious behavior. All those shellcodes were flagged as malicious by ShellOS. After including the ratio check, ShellOS failed to detect the modified shellcodes, while the same payloads were properly executed on native systems.

We did the same for TLB-based detection (M2). Unfortunately, ShellOS responded with errors and the debug traces indicated that something went wrong. The ShellOS authors could not resolve the problem in the time available, which is why we cannot evaluate TLB-based detection for this use case. Since ShellOS does not support threading, we unfortunately could not evaluate M3 on ShellOS.

6 Discussion

In this section we discuss several aspects of our proposed methods, including countermeasures, limitations when facing target systems that are virtualized, other virtualization methods than VT-x and finally how an adversary can use our methods to defeat multi-path exploration techniques

6.1 Countermeasures

Our detection methods are based on timing differences introduced by VM Exits. To evade detection, the hypervisor may tamper with the time-stamp counter (TSC) or avoid VM Exits. We first discuss techniques that aim to evade detection. In addition, we describe a technique for how our detection methods themselves can be detected at the hypervisor level.

Tampering with Timing Resources In principle, the hypervisor has complete control of the timing resources. The hypervisor could disable `rdtsc` for use in ring 3, subtract the VM Exit overhead or try to return sane estimated values. Disabling the instruction is not a beneficial option, since the usage of `rdtsc` is not malicious *per se*. Subtracting the VM Exit overhead works for protecting against M1 as described in Section 5.5. It will, however, fail for the TLB based detection method M2, for reasons described in the same section. Estimating sane return values is not feasible in practice as it requires complete knowledge about the emulated hardware.

Not Trapping `cpuid` Our detection methods rely on `cpuid` to trigger VM Exits. Disabling these exits by the hypervisor would thus evade detection. However, according to the Intel manuals, the VM Exit caused by `cpuid` cannot be disabled by the hypervisor. Therefore, a potential hardware-based solution could be to make traps to `cpuid` optional. But even then it would be likely that other timing biases remain, as hypervisors may need to trap on system calls or page faults to inspect host-based behavior.

Detecting Suspicious Timing-Based Evasion Instead of evading our methods, the hypervisor can just try to *detect* them. This is useful for sandboxes that just aim to find suspicious programs, rather than exploring their behavioral profile. Our first two methods can be detected by the hypervisor due to the excessive use of `rdtsc` and `cpuid`. If encountered too often, the program is at least suspicious. However, this is no help to sandboxes that not only flag a program as malicious, but also aim to reveal program behavior. Finally, as we have shown, an attacker can even avoid excessive use of suspicious instructions by using counter-based as opposed to timing-based detection.

6.2 Virtualization on Target Systems

The main use case of our methods is attacks that aim to evade virtualized analysis systems, such as sandboxes or other types of code analysis systems (like ShellOS). Such evasions are effective if we assume that the actual target systems (the victims) are *not* virtualized. However, research has suggested that virtualization will become more widespread in the future [2, 9, 32].

We argue that VM detection will still play an important role in the future. First, when looking at consumer devices, the degree of virtualization is negligible. Thus mass-targeting malware (such as ransomware, spambots, denial-of-service bots, or banking trojans) does not risk losing many potential victims by declining to run on VMs, while it can hide its behavior in virtualized sandboxes. In addition, even in targeted attacks, an adversary may be able to first spy on the exact environment, including the virtualization technology (if any), and then adapt her attack accordingly.

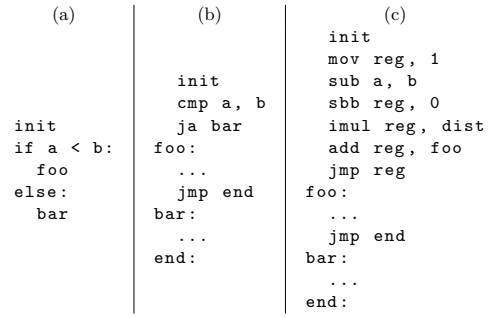


Fig. 4: *Overcoming multiple execution paths.*

6.3 Multi-Path Exploration

Facing the problem of evasion, researchers proposed new techniques to analyze the hidden behavior of evasive malware. A generic approach to counter evasive behavior is exploring *multiple execution paths* (or multi-path exploration), as introduced by Moser et al. [17]. The idea is to explore all input-driven parts of the program to trigger behavior that may otherwise remain hidden. For instance, assume program (a) in Figure 4, which first initializes, then does a check, and if the check succeeds, executes `foo`, or `bar` otherwise. Using multiple execution paths, we can explore all possible executions of the program, namely `init`→`foo` and `init`→`bar`, regardless of the values of `a` and `b`.

Kolbitsch et al. [16] proposed to identify and escape execution-stalling code during dynamic analysis. Their idea is to invert conditional branches once they are detected as such, e.g., turn a *greater-than* operation to a *less-or-equal*, with the goal to eventually explore all code regions. Related to this, Egele et al. [7] proposed *Blanket Execution*, a technique to cover all instructions within a function at least once during dynamic analysis [7]. In addition, the commercial sandbox vendor VMray has announced a technique to trigger dormant program functionality by inverting branch conditions, leveraging the history of recent branches in the Processor Tracing history of recent Intel CPUs[1].All these techniques aim to explore the entire functionality of a program or its functions.

We can overcome these approaches by converting conditional branches to indirect jumps/calls. Doing so removes the outgoing edge of the source basic blocks, as the address of the target basic block is computed dynamically. For example, consider the assembly representation of program (a) in Figure 4 (b). Multi-path execution would identify the conditional jump `ja bar` as a control flow decision provided that `a` or `b` is considered interesting. As a consequence, both possible execution paths will be identified and executed regardless of the actual values of `a` and `b`. Now let `dist` be the distance between `foo` and `bar` and consider program (c). If `a` is smaller than `b`, then the `sub a, b` instruction will set the carry flag. Hence, the register `reg` will contain the address of `foo`

[1] http://www.vmray.com/back-to-the-past-using-intels-processor-trace-for-enhanced-analysis/

and therefore the program will jump to `foo`. Conversely, if $a \geq b$, `reg` will hold the address of `bar` to which the program will jump. Programs (b) and (c) are thus semantically equivalent. However, in (c), one cannot identify the branching alternatives, since the conditional direct jump `ja bar` has been replaced with the unconditional direct jump `jmp reg`. By applying this transformation, an attacker can undermine multi-path execution to hide the malicious behavior of the program.

6.4 Non-Intel Virtualization

In this paper we have limited our analysis to Intel VT-x. We have shown that VT-x is the dominating virtualization technique used by sandboxes. In future work, we investigate whether our results are also applicable to other virtualization techniques, such as AMD-V. Due to the inherent timing artifacts introduced by the hypervisor, it is likely that the same concepts also apply to AMD-V. However, AMD-V enables the hypervisor to *disable* the `cpuid` trap, which is a notable difference between the two virtualization solutions. Therefore, AMD-V represents a stealthier alternative to VT-x. However, the presence of traps is inherent to the concept of sandbox monitoring, which likely introduces detection vectors other than `cpuid` traps.

7 Related Work

7.1 Virtualization Detection

In 2006, Ferrie [8] was one of the first to evaluate the transparency of software emulators, reduced privileged guests and hardware-assisted virtualization. Ferrie found bugs in the software emulators Hydra, Bochs and QEMU, which could be used to detect their presence. For reduced-privilege guests of the time, such as VMware, Virtual PC and Parallels, Ferrie found several artifacts. Ferrie concluded that reduced privilege guests cannot hide their presence, as their design does not allow them to intercept non-sensitive instructions, which will always imply detectable behavior.

Ferrie also mentioned timing-based attacks to detect hardware-assisted virtualization [8], such as measuring the overhead caused by VM Exits or considering a TLB-based attack. However, despite giving the general idea, he did not describe technical challenges, nor demonstrate or evaluate the methods. Research similar to Ferrie's work was conducted by Paleari et al. [20] in 2009 and by Raffetseder et al. [24] in 2007 to detect software emulators by exploiting implementation bugs. Raffetseder et al. additionally examined VT-x and observed that it is not possible to disable caching in a virtualized environment. We were inspired by these initial ideas, but are the first to go into detail and bring them to a realistic setting of user-mode code. In addition, we are the first to present a thorough evaluation of timing-based attacks to detect hardware-assisted virtualization in real-world use cases. Third, we also proposed a stealth method

that is much harder to detect than existing approaches. Finally, we have demonstrated the risk that adversaries may combine the detection mechanisms to evade multi-path exploration systems.

In 2006, Rutkowska proposed a rootkit based on hardware-assisted virtualization called *Blue Pill*[2]. Blue Pill is a thin hypervisor which will virtualize the existing OS once started. By doing so, the hypervisor (and therefore the attacker) gains full control over the OS. Blue Pill can be detected in the same way any VM can be detected by using our methods. Garfinkel et al. [10] proposed a TLB-based solution without using timing resources to detect such rootkits by manipulating the page table entries. Manipulating those entries is, however, not possible in user mode and therefore outside of the range of our threat model where malware usually operates.

7.2 Sandboxes and Evasion

In addition to the discussed VM detection methods, others have documented the problem of evasive malware through real-world studies. Chen et al. [2] were the first to study the behavior of malware that tries to detect VMs. They ran 6900 different malware samples under different environments and noted that 4% of the samples reduced their malicious behavior in the presence of a VM.

Balzarotti et al [1] propose a system that records the system call trace of a program when it is executed on a non-virtualized reference system. This trace is then compared against the trace of the same program being executed on a virtualized system, revealing possibly split behavior of the malware. Lindorfer et al. [16] describe a similar system and additionally introduce techniques for distinguishing between spurious differences in behavior and actual environment-sensitive behavior to cope with false positives.

The phenomenon of environment-aware malware forced sandbox maintainers and researchers to develop more transparent systems which are harder to detect. Dinaburg et al. proposed Ether [5], a malware analysis system using hardware-assisted virtualization which aims at remaining transparent to malicious software. Ether tries to maintain a clear time-stamp counter, as discussed in Section 6. However, Ether is still prone to TLB-based detection methods, which the authors of Ether classify as "architectural limitations" of VT-x. Additionally, Pék et al. presented a detection method for Ether [22]. Their method builds upon the observation that between two `rdtsc` instructions, Ether tends to increase the TSC by only 1 as long as there are no trapping instructions in between. This is an implementation-specific artifact that is not necessarily shared by other sandboxes.

To cope with the fundamental difficulties of creating a truly transparent hypervisor, Kirat et al. have suggested using bare-metal sandboxes [13, 14]. Although bare-metal sandboxes can evade VM-based detection mechanisms, they are less scalable and harder to maintain than virtualized sandboxes. Alternatively, Moser et al proposed multi-path exploration systems [17]. We have shown

[2] http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html

that adversaries can evade those systems and render them ineffective. The same holds for the brute force mechanism deployed by ShellOS [29], which simply aims to execute shellcode from every possible offset.

8 Conclusion

We have shown that hardware-assisted virtualization can be reliably detected by an adversary using timing-based measures. Unfortunately, not all of these methods can be detected as such, nor is there an effective and transparent way to evade all of them. This threatens important virtualized security infrastructures, such as malware sandboxes and state-of-the-art shellcode detection systems. Our attacks against multi-path exploration systems demonstrated that there is a need for further research to restore the guarantees that the full (possibly hidden) behavior of malicious code can be revealed with dynamic code analysis.

References

1. Balzarotti, D., Cova, M., Karlberger, C., Kruegel, C., Kirda, E., Vigna, G.: Efficient Detection of Split Personalities in Malware. In: Proceedings of NDSS (2010)
2. Chen, X., Andersen, J., Morley, M.Z., Bailey, M., Nazario, J.: Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware. In: DSN (2008)
3. Chun, B., Culler, D., Roscoe, T., Bavier, A., Peterson, L., Wawrzoniak, M., Bowman, M.: PlanetLab: An Overlay Testbed for Broad-coverage Services. SIGCOMM Comput. Commun. Rev. (2003)
4. Coogan, K., Lu, G., Debray, S.: Deobfuscation of Virtualization-obfuscated Software: A Semantics-based Approach. In: Proceedings of CCS (2011)
5. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: Malware Analysis via Hardware Virtualization Extensions. In: CCS (2008)
6. Egele, M., Scholte, T., Kirda, E., Kruegel, C.: A Survey on Automated Dynamic Malware-analysis Techniques and Tools. ACM Comput. Surv. (2012)
7. Egele, M., Woo, M., Chapman, P., Brumley, D.: Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In: USENIX Security (2014)
8. Ferrie, P.: Attacks on Virtual Machine Emulators. Tech. rep., Symantec (2006)
9. Franklin, J., Luk, M., McCune, J.M., Seshadri, A., Perrig, A., van Doorn, L.: Towards Sound Detection of Virtual Machines. In: Botnet Detection: Countering the Largest Security Threat. Springer (2008)
10. Garfinkel, T., Adams, K., Warfield, A., Franklin, J.: Compatibility is Not Transparency: VMM Detection Myths and Realities. In: Proceedings of USENIX HotOS (2007)
11. Kang, M.G., Poosankam, P., Yin, H.: Renovo: A Hidden Code Extractor for Packed Executables. In: Proceedings of the 2007 ACM Workshop on Recurring Malcode (2007)
12. Kinder, J.: Towards Static Analysis of Virtualization-Obfuscated Binaries. In: WCRE (2012)
13. Kirat, D., Vigna, G., Kruegel, C.: BareBox: Efficient Malware Analysis on Bare-metal. In: Proceedings of ACSAC (2011)

14. Kirat, D., Vigna, G., Kruegel, C.: Barecloud: Bare-metal Analysis-based Evasive Malware Detection. In: Proceedings of the USENIX Security (2014)
15. Kwon, B.J., Mondal, J., Jang, J., Bilge, L., Dumitras, T.: The Dropper Effect: Insights into Malware Distribution with Downloader Graph Analytics. In: CCS (2015)
16. Lindorfer, M., Kolbitsch, C., Milani Comparetti, P.: Detecting Environment-sensitive Malware. In: Proceedings of RAID (2011)
17. Moser, A., Kruegel, C., Kirda, E.: Exploring Multiple Execution Paths for Malware Analysis. In: Proceedings of the S&P (2007)
18. Moser, A., Kruegel, C., Kirda, E.: Limits of Static Analysis for Malware Detection. In: Proceedings of ACSAC (2007)
19. Oktavianto, D., Muhardianto, I.: Cuckoo Malware Analysis. Packt Publishing (2013)
20. Paleari, R., Martignoni, L., Roglia, G.F., Bruschi, D.: A Fistful of Red-pills: How to Automatically Generate Procedures to Detect CPU Emulators. In: Usenix WOOT (2009)
21. Ugarte Pedrero, X., Balzarotti, D., Santos, I., Bringas, P.G.: SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In: S&P (2015)
22. Pék, G., Bencsáth, B., Buttyán, L.: nEther: In-guest Detection of Out-of-the-guest Malware Analyzers. In: CCS (2011)
23. Polychronakis, M., Anagnostakis, K.G., Markatos, E.P.: Comprehensive Shellcode Detection Using Runtime Heuristics. In: Proceedings of ACSAC (2010)
24. Raffetseder, T., Kruegel, C., Kirda, E.: Detecting System Emulators. In: ICISC (2007)
25. Rolles, R.: Unpacking Virtualization Obfuscators. In: Usenix WOOT (2009)
26. Rossow, C., Dietrich, C., Bos, H.: Large-Scale Analysis of Malware Downloaders. In: Proceedings of DIMVA (2013)
27. Royal, P., Halpin, M., Dagon, D., Edmonds, R., Lee, W.: PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In: CCS (2006)
28. Sharif, M., Lanzi, A., Giffin, J., Lee, W.: Automatic Reverse Engineering of Malware Emulators. In: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy (2009)
29. Snow, K.Z., Krishnan, S., Monrose, F., Provos, N.: ShellOS: Enabling Fast Detection and Forensic Analysis of Code Injection Attacks. In: Proceedings of USENIX Security (2011)
30. Willems, C., Hund, R., Holz, T.: CXPInspector: Hypervisor-based, hardware-assisted system monitoring. Horst Görtz Institute for IT Security: Technical report (2012)
31. You, I., Yim, K.: Malware Obfuscation Techniques: A Brief Survey. In: Proceedings of BWCCA (2010)
32. Zhao, X., Borders, K., Prakash, A.: Virtual Machine Security Systems. In: Advances in Computer Science and Engineering. Springer (2009)