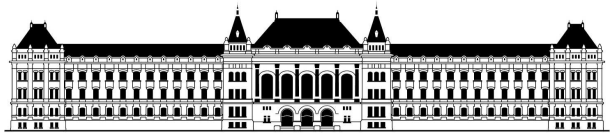

Budapest University of Technology and Economics
Department of Networked Systems and Services

New Methods for Detecting Malware Infections and New Attacks against Hardware Virtualization

Collection of Ph.D. Theses
of
Gábor Pék

Supervisor:
Levente Buttyán, Ph.D.

Budapest, Hungary
2015

1 Introduction

In my dissertation, I aim at answering various system security questions which still raise hot debates among field experts. First and foremost, the detection of unknown malware has been studied over many years, however, targeted attacks seems to successfully exploit existing preventive and reactive solutions. Interestingly, most of these attacks still use well-known, traditional methods to hide their presence. One such a method is the injection of hostile code into benign system processes. As it has been demonstrated by recent targeted campaings (e.g., Flame), the way of injection can be exotic enough to evade signature based approaches and various heuristics. At the same time, deploying such detection/analyis solutions into production environments is not trivial to perform in every cases as machines need to be restarted. In critical infrastructures, for example, neither interruption of operation nor installing arbitrary software on the system are allowed.

To address these problems, in the first part of my dissertation, I introduce a novel memory forensics approach using paging event analysis of Microsoft Windows operating systems to detect code injection attacks. To the best of my knowledge, I am the first who build upon these events in such detail to detect malicious codes. To demonstrate and evaluate my approach, I propose an analysis technique together with a framework called Membrane which could successfully detect unknown code injector malware with high accuracy even in the noisiest system processes (i.e., `explorer.exe`). More precisely, I reached 91-98% detection rate on Windows XP machines, and 75-86% under Windows 7 when malware injected into `explorer.exe` which is the noisiest system process according to my observations. This framework has been implemented as a snapshot-based memory forensics tool and a live monitoring system for dynamic analysis.

Then, I suggest a new system monitoring tool for live systems to analyze and catch malware infections. I describe my design and implementation of an on-the-fly virtualization platform, and I also propose a novel system call tracing method that can be used to observe the behavior of the analyzed system with configurable tradeoff between transparency and granularity. My proof-of-concept implementation leverages AMD64 processors with SVM (Secure Virtual Machine) technology and 64-bit Windows Vista/7 operating systems.

Cloud computing and virtulization services have interwoven our everyday life as they provide flexible and maintainable ways to scale up our IT solutions to the market needs. Meanwhile, however, these technologies brought field-specific security concerns (e.g., multitenancy) that were not even raised by vendors for many years. This lead us to build mission critical systems atop an entirely new hardware and software stack the security of which has not been designed and implemented properly. These security issues range from the illusion of providing perfect transparency in virtualization to the inproper handling of VM images in multitenant setups.

In the second part of my dissertation, I propose two novel attacks against I/O virtualization, one based on the modification of the Peripheral Component Interconnect express (PCIe) configuration space and the other based on the creation of host-side Non-Maskable Interrupts (NMIs). While the former was discovered manually, the latter was revealed by my fuzzer, called PTFuzz, that I built to automatically reveal low-level problems during DMA operations. In addition, PTfuzz revealed another unexpected hardware behaviour during testing interrupt attacks. More precisely, my interrupt attack (i.e., XSA-59, CVE-2013-3495) is the consequence of a misunderstanding between the hardware and software which allows guest-to-VMM escapes [1] independently from the VMM software itself [2]. In addition, it is the only interrupt attack to date that works on configurations in which all available hardware protections are turned on. While this attack has been first reported to vendors (e.g., Xen, KVM, VMware) during the

[1] Theoretically arbitrary code execution is possible, but current system implementations are vulnerable to DoS

[2] Tested and verified on Xen 4.3 and KVM 3.5

spring of 2013, no real solution could still be employed. The reason for this lies in the fact that hypervisor vendors have to handle every Intel CPU chipset separately [3] so as not to limit key functionalities with their workarounds. Unfortunately, the list of these chipsets is endless, so the vulnerability cannot be put to rest in future hypervisor releases.

Finally, I introduce new detection attacks against the *out-of-the-guest* malware analysis framework *Ether* [DRSL08a]. More specifically, I present an in-guest timing attack which was supposed to be prevented by the out-of-the-guest malware analyis system called Ether, but for various reasons, it actually does not prevent it, and an attack based on the detection of some specific settings that Ether makes in the system configuration. I also propose a generic technique to detect hardware virtualization platforms based on the verification of the presence of CPU specific design defects (i.e., errata).

[3] `http://www.gossamer-threads.com/lists/xen/devel/360060`

2 New Results

2.1 New Methods for Detecting Malware Infections

THESES 1: *I propose a novel memory forensics-based technique together with a tool called Membrane which builds upon paging event analysis to detect unknown code injection attacks on Windows execution environments.*

Recent years' targeted attack have shown that even the most advanced systems can be compromised. Some of these targeted attacks used sophisticated intrusion techniques [CER14] and others were quite simple [Ali14]. These malware attacks typically employed a sequence of steps to compromise a target system. A majority of these malware codes have information gathering and information stealing capabilities. Once they have access to their target, these malware codes typically perform a number of operations to cover their traces and remain undetected. Intuitively, the more the malware can persist in the target system, the more information it can collect. Often, the attacks persist for years in the target systems and the attackers get access to a substantial amount of confidential information (as reported for example in [Man13]). It is reasonable to assume that these long-term operations leave a noticeable trace, yet many examples show that the complexity and rich features of contemporary operating systems leaves ample space for the attackers to operate.

Code injection is one of the key techniques that malware employs to achieve persistence. Code injection happens when the malware adds or replaces the functionality of existing code to execute its added components. It is typically possible as, for example, the Windows operating systems provides various methods (e.g., legitimate API functions, registry entries) to achieve this. Thus, code injection usually exploits the conditions given by a legitimate process. That is why code injection is used to achieve desirable properties, such as evasion of detection or bypassing restrictions enforced on a process level.

There has been efforts to develop various memory and disk forensics techniques to pinpoint system anomalies caused by such malware infections [IM07]. One of the biggest problem with current memory forensics techniques is that they only utilize memory locations that were actively used by the OS at the moment of acquisition. That is, important information about injections can be lost if the malware or part of it was inactive when the memory was grabbed. Furthermore, the rich feature set of Windows allows miscreants to build unique injection techniques (e.g., Flame's injection mechanism) that evades signature-based protections. Thus, the understanding of Windows memory management is a promising avenue to detect when malicious code is injected into already running processes.

I explore the realm of Windows memory management, systematically identify key paging events and build upon the details of these paging events to detect malicious behavior. Next, *Membrane* an anomaly-based memory forensics tool is designed and implemented to detect code injection attacks by malicious software. Membrane is based on the popular memory forensics framework Volatility [Vol14]. Membrane performs detection by analyzing the number of memory paging events. This approach is different from approaches in related work because it focuses on detecting anomalies (symptoms) concerning paging events of malware code injection behavior instead of the code injection actions themselves.

Table 1: Detecting generic and targeted code injecting malware on our WinXP and Win7_1 VMs with different network connections in `explorer.exe`. We used a $K = 5$ cross-validation iteration for the measurements. Note that the targeted samples of our test-only dataset were executed only under Win7_1.

Cross-validation dataset from generic samples				
Internet	**VM**	**ACC %**	**TPR %**	**FPR %**
no	WinXP	98.67	98.82	1.18
	Win7_1	73.59	75.92	28.46
yes	WinXP	92.81	90.88	5.45
	Win7_1	79.45	82.69	23.59
Cross-validation dataset from generic samples with no additional processes				
no	Win7_1	77.16	86.00	34.73
Test-only dataset from generic and targeted samples				
no	WinXP	100	100	-
	Win7_1	100	100	-

THESIS 1.1: *I propose a novel memory-forensics technique together with a tool called Membrane which is able to detect a wide range of code injection techniques. My results indicate that Membrane can detect code injection malware behavior with 86-98% success on Windows platforms even in one of the noisiest system processes such as* `explorer.exe`

To evaluate Membrane, an extensive set of experiments were executed with two network containment configurations: (i) no Internet connection is enabled, (ii) real Internet connection is enabled with a carefully crafted containment policy following the suggestions of Rossow *et al.* [RDG+12]. When Internet connection was enabled, NAT was used with the following containment policy: a) known C&C TCP ports were enabled (e.g., HTTP, HTTPS, DNS, IRC) b) TCP ports were redirected with supposedly harmful traffic (e.g., SMTP, ports not registered by IANA) to a INetSim network simulator [INe14] that was also configured, and c) rate-limitation was also set on analyzed VMs to mitigate DoS attacks.

Out of the 194 generic code injector malware, 128 samples targeted `explorer.exe` which turned to be the most popular injection target. That is why the used random-forest classification algorithm works with these samples. The evaluation process comprises three parts: (i) code injectors are evaluated on VMs with preinstalled and started benign applications with K-fold cross-validation, (ii) same as the previous point, but no benign processes are installed and started, (iii) the best performing random forest classificator is chosen from the cross-validation process to evaluate the test-only dataset of generic targeted malware.

In cases (i) and (ii), prepared Windows XP (i.e., WinXP) and Windows 7 (i.e., Win7_1) environments were infected in each network containment configuration. After retrieving snapshots with Membrane, the classification algorithm was executed to find malicious infections. Table 1 summarizes the results and strengthens some of my key observations.

Running processes raise the noise of certain system processes. This observation manifests in lower detection ratio as the (i) and (ii) versions of Win7_1 show it. My next observation is that the execution environment also affects detection accuracy as the *TPR* of Win7_1 and WinXP setups show it.

While the detection rates are fairly good in case of Windows XP, the results are worse for

Windows 7 snapshots due to the increased baseline paging event activity of `explorer.exe` in Win7_1. We can increase the detection accuracy by stopping legitimate processes and thus offloading `explorer.exe`. We see this increase when the TPR in Table 1 increases from 75.92% to 86%.

THESIS 1.2: *I show that the detection accuracy is influenced by the target process, the execution environment and the benign processes running on an analyzed system. All my measurements were performed on real system setups without using simulation. Additionally, I demonstrate that my approach is agnostic to the injection technique itself, thus a wide range of techniques can be detected. Furthermore, I demonstrate that recent targeted attacks using code injection can also be pinpointed.*

In order to *manually* evaluate prominent features, ratios between the median of clean *and* malicious feature vectors were calculated in a given $p \in P$ process, where P represents all the processes in a given execution environment (e.g., WinXP, Win7_1). This approach also enables to define how the execution environment and the examined process can influence detection accuracy. More precisely, I define

$$r_{\mu}(G_p^{(f)}, M_p^{(f)}) = \frac{\mu(G_p^{(f)})}{\mu(M_p^{(f)})} \tag{1}$$

as the *median ratio* for process $p \in P$, where $f \in F$ feature points to one of the paging entry types (i.e., F) that were restored. Due to the limited number of snapshots a given process was evaluated, I use *median* instead of expected values. Malicious frequencies for f feature, process p and n snapshots are denoted by $M_p^{(f)} = (m_{p_1}^{(f)}, m_{p_2}^{(f)}, ..., m_{p_n}^{(f)})$, while the benign ones are marked with $G_p^{(f)} = (g_{p_1}^{(f)}, g_{p_2}^{(f)}, ..., g_{p_n}^{(f)})$.

In the next series of experiments, recent targeted cyber attacks injecting into `explorer.exe` were evaluated on Win7_1 from the families of Snake, Shale, Epic and Turlae and Flame. Note that the former four belong to the Uroburos campaign revealed by G Data in 2014 [G D14]. Internet connection was allowed in accordance with the containment policy I described earlier. Periodic snapshots were recorded for 2 hours, one in each 5 minutes (i.e., $n = 24$). As Figure 1 shows, most of the feature cardinalities increased (i.e., r_{μ} ratio is lower than one) after the VM was infected. While Demand Zero paging events are flapping, Zero PTEs remained relatively stable in most of the cases, except for Flame.

I furthermore evaluated samples injecting into other system processes. I choose `services.exe` as it is a popular target and demonstrates well the advantages of my approach. Single snapshots were created from each sample of a given malware family. Then, I calculated the median ratio to evaluate the results. As Figure 2 shows, most of the examined paging events increased significantly, especially the Zero feature for the samples of ZAccess. Interestingly, the number of Demand Zero entries decreased for the samples of Mydoom and Redyms, which can be attributed to the commitment of previously allocated pages that are not accessed normally by `services.exe`. As the ratios are lower than in case of `explorer.exe` injections, I conclude that the *target process influences the detection accuracy.* This observation is also strengthened by the baseline ratios I discuss below.

Various execution environments were profiled for baseline paging event behavior to understand why detection rate decreases in the special case when the malware injects into `explorer.exe` on Win7_1. Similarly to median ratio, I define the *standard deviation ratio* to estimate the noise of certain features in given setups:

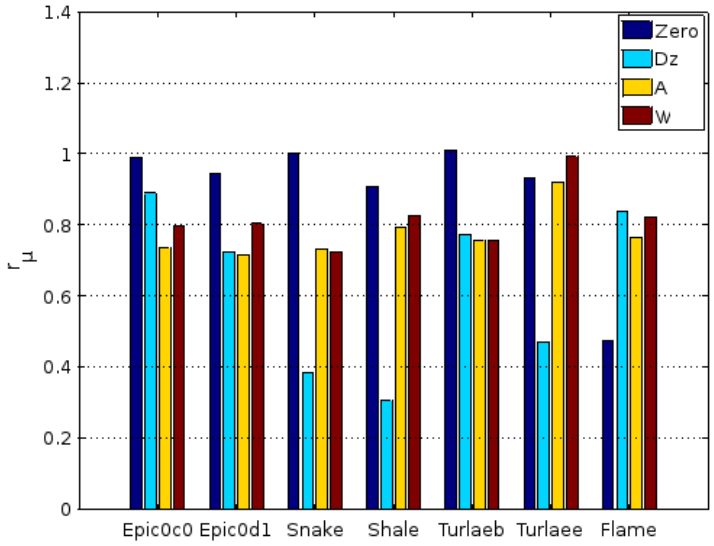

Figure 1: Evaluating targeted samples injecting into `explorer.exe` with features Zero, Demand Zero (Dz), Accessed (A) and Writable (W) in Win7_1.

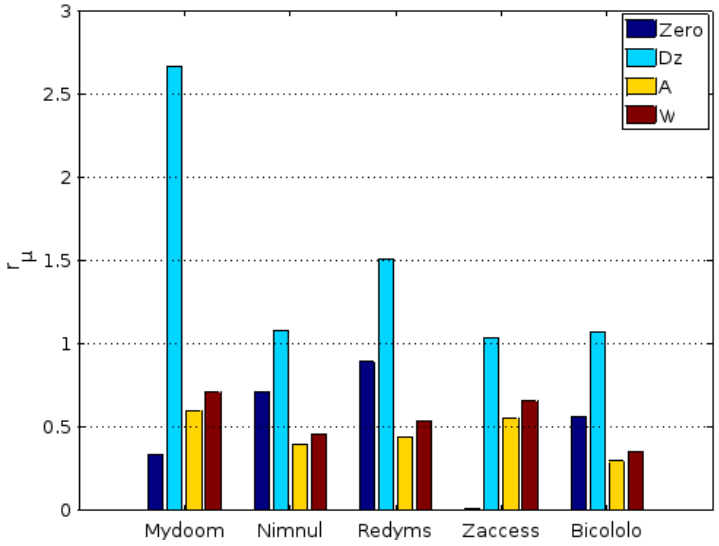

Figure 2: Evaluating multiple samples from different malware families injecting into `services.exe` with features Zero, Demand Zero (Dz), Accessed (A) and Writable (W) in Win7_1.

$$r_\sigma(G_p^{(f)}, M_p^{(f)}) = \frac{\sigma(G_p^{(f)})}{\sigma(M_p^{(f)})} \tag{2}$$

Then, I carefully calculated the median and standard deviation ratios of certain per process features for different system processes. More precisely, first 24 snapshots were recorded in 2 hours (i.e., one snapshot in every 5 minutes) with Membrane from our WinXP, Win7_1, Win7_2 execution environments without starting additional processes. Then, 24 snapshots were recorded again, but additional benign processes were also started. At this point, I calculated some of our features and compared the results between the two executions. To demonstrate how system noise is influenced by running processes, only an MS calculator was started in the first $n = 24$ snapshots for Win7_1c, and launched a Dropbox client application for the second run.

My observations from basline ratios are the following.

- System process noise is influenced by other running processes. One of the noisiest processes is `explorer.exe` indicated also by low standard deviation ratio in Table 2 (i.e., $r_{\sigma}^{Zero} = 0.051$, $r_{\sigma}^{Vp} = 0.039$, $r_{\sigma}^{W} = 0.033$) in Win7_1. As `explorer.exe` is responsible for launching graphic shell for users after successful login, this behaviour is not surprising.

  However, when the experiment was repeated by starting only a Dropbox client (i.e., Win7_1c), some of the noises dropped significantly (i.e., $r_{\sigma}^{Zero} = 1.728$, $r_{\sigma}^{Vp} = 0.423$) by keeping fairly good median ratios (i.e., $r_{\mu}^{Zero} = 1.046$, $r_{\mu}^{Dz} = 1.056$). At the same time, other system processes (e.g., `winlogon.exe`, `services.exe`) are not influenced by other processes with that extent. For example, another popular injection target `services.exe` comes with $r_{\mu}^{Zero} = 0.958$ $r_{\sigma}^{Zero} = 0.613$, $r_{\mu}^{W} = 1.001$ and $r_{\sigma}^{W} = 0.31$ on Win7_1.

- System process noise is influenced by the operating system version. Execution environments WinXP and Win7_* share some common features (i.e., Zero PTE, Demand Zero, Valid Prototype). By comparing these features I concluded that the baseline ratios of WinXP are closer to one than in its Win7_* counterparts for almost all the system processes. While $r_{\sigma}^{Zero} = 0.051$, $r_{\sigma}^{Dz} = 0.121$, $r_{\sigma}^{Vp} = 0.039$ for `explorer.exe` on Win7_1, $r_{\sigma}^{Zero} = 0.711$, $r_{\sigma}^{Dz} = 1.328$, $r_{\sigma}^{Vp} = 0.89$ on WinXP. Median ratios are quite similar, however: $r_{\mu}^{Zero} = 1.044$, $r_{\mu}^{Dz} = 0.819$, $r_{\mu}^{Vp} = 1.151$ on Win7_1 and $r_{\mu}^{Zero} = 1.024$, $r_{\mu}^{Dz} = 0.998$, $r_{\mu}^{Pv} = 0/0$ on WinXP. Note that the latter "0/0" indicates the both of the compared samples have zero medians. This difference also holds for other system processes such as `winlogon.exe`: $r_{\sigma}^{Zero} = 0.543$, $r_{\sigma}^{Dz} = 0.021$, $r_{\sigma}^{Vp} = 0.659$ on Win7_1 and $r_{\sigma}^{Zero} = 0.964$, $r_{\sigma}^{Dz} = 1.571$, $r_{\sigma}^{Vp} = 0.738$ on WinXP.

THESIS 1.3: *I propose a live version of Membrane called Membrane Live which comes with the same detection capabilies as Membrane. The measurements with Membrane Live proved that the analyis environment does not influence the baseline distances of the monitored system. Membrane Live was implemented as a hypervisor extension using Virtual Machine Introspection.*

I propose Membrane Live which is an extension for the DRAKVUF [LMP+14] malware analysis system using Virtual Machine Introspection (VMI). Membrane Live implements the same functionality as Membrane, however, it records numerous paging events after every invoked kernel function which DRAKVUF hooks. In this way, the in-depth behavior of different code injection attacks can be understood in together with the anomaly they cause. To achieve this, another guest environment was also set up called *Win7_3* that was monitored with Membrane Live. Membrane Live was only used to measure baseline distances and compare these results with that of Membrane.

To do that, entirely clean execution traces with $n = 400$ entries in Win7_3 were compared. As Table 2 shows it, there is no significant difference between the clean baseline distances of system processes while being monitored with Membrane and Membrane Live. For any feature, the closer the ratio indicator to one the compared traces are more similar.

THESIS 1.4: *I show that the injected malicious functionalities influence the cardinalities of different paging events in different ways. Thus, the natures of malware functionaility impacts the detection accuracy.*

To understand how the injected malicious functionalities influence process paging events, some of such functionalities were implemented and inserted by the remote thread injection technique into `winlogon.exe` as Table 3 shows it. This process (i.e.,`winlogon.exe`) seemed to be a good option, as it has low baseline noise, and thus, it is easier to detect malicious activities inside it.

Table 2: Baseline ratios of various detection features for the WinXP, Win7_1, Win7_1c, Win7_2 execution environments by analyzing and comparing $n = 24$ snapshots with Membrane. The abbreviation are: Zero - Zero PTE, Cw - Copy-on-write, Dz - Demand Zero, Vp - Valid Prototype, Mfp - Mapped File Prototype, A - Accessed, W - Writeable. Baseline ratios for Win7_3c was recorded and analyzed by Membrane Live by comparing execution traces with $n = 400$ entries. Ratios for unselected features (i.e., with no or minimal information gain) in given execution environments are marked with "-".

			Features						
Process	**VM**	**Ratios**	**Zero**	**Cw**	**Dz**	**Vp**	**Mfp**	**A**	**W**
explorer	WinXP	r_μ	1.024	1.166	0.9980	0/0	0/0	-	-
		r_σ	0.711	0.903	1.328	0.890	0	-	-
	Win7_1	r_μ	1.044	-	0.8190	1.151	-	1.0410	1.090
		r_σ	0.051	-	0.121	0.039	-	0.189	0.033
	Win7_1c	r_μ	1.046	-	1.056	1.214	-	0.821	0.796
		r_σ	1.728	-	3.172	0.423	-	0.700	0.097
	Win7_2	r_μ	0.960	-	1.1050	1.9420	-	1.274	1.017
		r_σ	0.638	-	1.803	0.688	-	1.294	1.204
	Win7_3c	r_μ	0.998	-	1.051	1	-	1.380	1.052
		r_σ	1.621	-	1.436	0/0	-	1.553	1.6230
services	WinXP	r_μ	1.000	1.2380	1.001	0/0	0/0	-	-
		r_σ	0.905	0.891	1.056	1.106	1.341	-	-
	Win7_1	r_μ	0.958	-	1.000	1.442	-	0.859	1.0010
		r_σ	0.613	-	0.160	1.1100	-	0.304	0.310
	Win7_1c	r_μ	0.9710	-	1.001	0.689	-	0.949	0.897
		r_σ	0.475	-	0.471	0.5440	-	0.997	1.0610
	Win7_2	r_μ	0.994	-	0.914	2.5	-	1.1310	1.001
		r_σ	0.961	-	1.734	2.786/0	-	1.2530	1.1220
	Win7_3c	r_μ	1.0000	-	1.019	1.000	-	0.813	0.999
		r_σ	0.930	-	0.983	0	-	0.898	0.930
winlogon	WinXP	r_μ	1.002	1	0.997	1.000	1	-	-
		r_σ	0.964	1.001	1.571	0.738	1.242	-	-
	Win7_1	r_μ	1.004	-	1.013	1.041	-	0.880	0.969
		r_σ	0.543	-	0.021	0.659	-	0.628	0.456
	Win7_1c	r_μ	0.998	-	1.002	0.88	-	0.9240	0.9040
		r_σ	1.262	-	1.288	0.913	-	0.9010	1.100
	Win7_2	r_μ	1.000	-	1.007	1	-	1.058	0.998
		r_σ	1.161	-	1.296	0/0	-	1.0200	1.0680
	Win7_3c	r_μ	1	-	1	1	-	1.0000	1
		r_σ	1	-	0/0	0/0	-	0.913	1
lsass	WinXP	r_μ	0.993	0.920	0.995	0.5	0	-	-
		r_σ	0.967	0.831	2.632	0.851	8.944	-	-
	Win7_1	r_μ	1.0090	-	1.009	1.085	-	1.024	1.118
		r_σ	0.801	-	0.051	0.594	-	0.290	0.159
	Win7_1c	r_μ	0.959	-	1.0060	0.856	-	0.900	0.891
		r_σ	0.853	-	1.087	5.709	-	1.664	2.526
	Win7_2	r_μ	1.009	-	0.992	1	-	1.211	1.001
		r_σ	1.147	-	1.315	0/0	-	1.135	3.611
	Win7_3c	r_μ	0.999	-	0.973	1	-	0.999	1.0000
		r_σ	0	-	1.026	0/0	-	1.477	0.579

Table 3: Comparing paging event cardinalities between different injected malware functionalities (i.e., ❶ Keylogger, ❷ Data exfiltration, ❸ Registry operations) and the clean execution of Windows 7 (i.e., Win7_1c) for `winlogon.exe`. Column *WinAPI calls* highlights some of the key WinAPI functions we called in a given functionality. All the functionalities were injected into the target process by using `CreateRemoteThread` detailed in Table 4.

Malware func.	**WinAPI calls**	r_{μ}^{Zero}	r_{σ}^{Zero}	r_{μ}^{A}	r_{σ}^{A}	r_{μ}^{Dz}	r_{σ}^{Dz}	r_{μ}^{W}	r_{σ}^{W}
❶	CreateFile ↻ GetAsyncKeyState ↻ MapVirtualKeyEx ↻ GetKeyNameText ↻ WriteFile ↻ CloseHandle ↻	0.98	0.86	0.98	0.89	1.01	1.22	0.53	1.83
❷	socket bind listen accept ↻ recv ↻ CreateFile ↻ ReadFile ↻ send ↻ CloseHandle ↻	0.86	0.58	1.07	0.59	0.01	0.42	0.02	0.28
❸	RegOpenKeyEx(HK*) RegQueryInfoKey RegEnumKeyW ↻ RegSetValueEx ↻ RegEnumValueW ↻ RegOpenKeyExW ↻ RegQueryValueExW ↻ RegCloseKey ↻ RegCloseKey	0.99	0.72	0.77	0.72	2.78	1.15	0.02	0.76

To calculate the corresponding median and strandard deviation ratios $n = 24$ snapshots were recorded in 2 hours from each of the cases and I compared them with Win7_1c.

First, a keylogger was injected into our target processes by starting an infinite loop which records all the keystrokes in an internal buffer. Interestingly, this functionality is quite silent and only the number of writeable pages increases enough (i.e., $r^{W}_{\mu_{baseline}} = 0.90$ and $r^{W}_{\mu_{keylogger}} = 0.53$).

As only a small buffer is required to store keystrokes, which can be flushed into a few pages, this functionality seems to be difficult to detect on its own.

The second injected functionality is a server application which reads a small text file with the size of a few KBs and keeps exfiltrating it to its client. I intentionally chose a file which is small enough. This allows me to see if this functionality can be detected by my approach. As Table 3 demonstrates it, the process noise increased fairly well (e.g., $r^{Dz}_{\sigma_{baseline}} = 1.288$ and $r^{Dz}_{\sigma_{dataexfiltration}} = 0.42$). On the other hand, mean ratios dropped, thus, key feature cardinalities increased (e.g., $r^{W}_{\mu_{baseline}} = 0.90$ and $r^{W}_{\mu_{dataexfiltration}} = 0.02$). The reader might also realize that the key paging event, Zero PTE, also increased. This is not surprising if we recall Table 4. As the exfiltrated file needs to be mapped into the process address space and Windows uses lazy evaluation, only Zero PTE entries are created until the file is first accessed.

Finally, a registry traversal functionality was implemented which iterates through all the HKEY_* keys and adds certain new values under them. According to Table 3, the process noise is increased in together with the number of writeable pages.

THESIS 1.5: *I show that the fact of code injection changes the paging event cardinalities of the target process. I demonstrate this statement by analyzing a concrete injection method using remote thread creation in a target process.*

The most relevant paging events to detect code injection were selected by calculating per process Gini indexes. In order to understand why these paging events were chosen by this algorithm, the details of Windows internals had to be explored by using various resources [RSI12, Wil11, Rea14]. For dynamic analysis, WinDbg was used.

Table 4 shows an example for a typical code injection attack using a certain sequence of WinAPI functions. More precisely, first `VirtualAllocEx` is called to allocate a *private* memory in the target process to store the name of the DLL to load into that process. The corresponding kernel function, `NtAllocateVirtualMemory` creates and initializes a single VAD entry with *RW* protection bits set, *zeroes out the PTE* (i.e., zero type software PTE is created) of the allocated memory range and sets up the page fault handler (#PF). The `NtProtectVirtualMemory` kernel function is also called after `NtAllocateVirtualMemory`, which sets software PTE protection bits, thus the software PTE is changed to be Demand Zero (Dz). When `WriteProcessMemory` tries to access the page, the page fault handler is invoked. At this point, the page fault is caught, hardware PTE is created and initialized (i.e., Valid and Write hardware PTE bits are set) using the previously created VAD entry as a template. This lazy memory allocation mechanism allows Windows to build page tables for only accessed pages. When #PF finished, the execution is handed to the `NtWriteVirtualMemory` kernel function, which writes the name of the malicious DLL to be loaded by the target process. Note that `NtAllocateVirtualMemory` and `NtWriteVirtualMemory` run in the context of the target process, that is why they are allowed to write process-private memory locations. As the allocated pages were accessed and modified, the hardware MMU now sets the accessed (A) and dirty (D) bits on the PTE, respectively. In the next step, `CreateRemoteThread` is called by setting the newly created thread's start address to the `LoadLibrary` WinAPI function. At this point, `LoadLibrary` is invoked by the newly created thread with the name of the malicious DLL to be loaded. As the table shows, `LoadLibrary` invokes various kernel functions. By doing so, first a file handle is returned by `NtOpenFile`, that is handed to `NtCreateSection` to create a *Section* object for the the corresponding DLL. Due to the lazy memory management of Windows no hardware PTE is created at this point, however, only Prototype PTE kernel structures are initialized. I emphasize here, that PPTEs are initialized with `EXECUTE_WRITECOPY` protection masks (i.e., we note that with the RWX and Cw marks in the Table) to enable Copy-On-Write operations. To support memory sharing the DLL is now mapped as view of the section object (i.e., file-backed-image is mapped) by the

Table 4: Injecting DLL into a process by using ❶ `VirtualAllocEx`, ❷ `WriteProcessMemory`, ❸ `CreateRemoteThread` and ❹ `LoadLibrary` WinAPI functions. Note that the correspoding kernel functions of `VirtualAllocEx` and `WriteProcessMemory` are executed also in the context of the target process. In this way, Windows guarantees that no address space violation occurs between the originator and target processes when loading the new DLL. Note that the *start address (sa)* parameter of `CreateRemoteThread` means the start address of operation when the new thread is created. While the *HW PTE* column indicates the hardware PTE bits being set when the corresponding kernel function is called, the *SW PTE* column refers to the software PTE subtype used to handle invalid pages. Similarly, column *PPTE* and *VAD entry* is associatd with the status of Prototype PTEs and VAD entries. While the upper part of the table refers to the memory region allocated for the *name of the DLL*, the lower part indicates table entry changes for the memory region allocated for *the DLL* itself.

Win API	**Parameters**	**Kernel function**	**HW PTE**	**SW PTE**	**PPTE**	**VAD entry**
Table entry changes on the memory region of the DLL name						
❶	protection=RW	NtAllocateVirtualMemory		Zero		RW
		NtProtectVirtualMemory		Dz		RW
❷	size=len(dll_name)	#PF (NtWriteVirtualMemory)	V,W			RW
		NtWriteVirtualMemory	V,W, A,D			RW
❸	sa= &LoadLibrary	NtCreateThreadEx	V,W, A,D			RW
Table entry changes on the memory region of the DLL						
❹	dll_name	NtOpenFile				
		NtCreateSection			R,W, X,Cw	
		NtMapViewOfSection		Zero	R,W, X,Cw	R,W, X,Cw
		NtQuerySection		Zero	R,W, X,Cw	R,W, X,Cw
		NtClose		Zero	R,W, X,Cw	R,W, X,Cw
		#PF (on first access)	V,W, Cw		R,W, X,Cw	R,W, X,Cw

`NtMapViewOfSection` kernel function, which also creates a VAD entry by using the Prototype PTEs and sets up the page fault handler to build hardware PTEs when the mapped DLL is first accessed. Finally the opened file handler was closed by `NtClose`.

As Table 4 shows it, the presented injection technique creates *Zero software PTEs* and *hardware PTEs with Valid, Writeable, Accessed, Dirty and Copy-on-Write bits set.* This example shows fairly well, how some simple WinAPI functions influence the management of memory in Windows OSs. At the same time, this rich behaviour allows us to observe how different code injection methods manipulate per process PTEs.

THESES 2: *I propose a new system monitoring approach which uses on-the-fly live virtualization that enhances either the transparency or performance of existing methods (e.g., Nitro [PSE11] and Ether [DRSL08b]). Thus, my approach does not require to create a copy of the system to be analyzed, neither it requires the installation of analysis tools on the analyzed system itself. It is free from all limitations of prior approaches.*

An important limitation of existing host-based anomaly detection approaches is that they require either to run the system to be analyzed in an isolated (usually virtualized) environment, or to install some analysis tools on the analyzed system itself. In the first case, one needs to create a virtualized copy of the analyzed system *and* its original environment (e.g., other servers in the same network) in order to run both together in the isolated analysis environment. Note that if the copy of the analyzed system runs alone, then the malware may detect the change in its environment and modify its behavior in order to escape detection. Creating a faithful copy of the operating environment of the analyzed system, however, is a major problem that requires a lot of resources and may cause interruptions in the operation of the live system. There is also a high risk that the copy will actually not be sufficiently faithful, which may jeopardize the entire malware detection process. In the second case, when some analysis tools are installed on the analyzed system itself, the problem is that the analysis will not be transparent, meaning that the installed analysis tools may be detected by the malware. In addition, in some environments such as, for instance, in IT systems of critical infrastructures, neither interruption of operation nor installing arbitrary software on the system are allowed.

I propose a new system monitoring approach that enhances either the transparency or performance of existing methods (e.g., Nitro [PSE11] and Ether [DRSL08b]). In addition, my approach does not require to create a copy of the system to be analyzed, neither it requires the installation of analysis tools on the analyzed system itself. Thus, it is free from all limitations of prior approaches. My approach takes advantage of the increased availability of hardware assisted virtualization capabilities of modern CPUs, and its basic idea is to launch a hypervisor layer on the live system on-the-fly, without stopping and restarting it or any of the running applications. This hypervisor runs at a higher privilege level than the OS itself, thus, it can be used to observe the behavior of the analyzed system in a transparent manner, without installing any analysis tools on the analyzed system itself. At the same time, the live system stays in its original operational environment, so the malware may not detect any suspicious change. At the same time, I also propose a novel system call tracing method that is designed to be configurable in terms of transparency and granularity.

THESIS 2.1: *I propose a new system monitoring algorithm that is configurable in terms of transparency and granularity. Also, I measured the performance of my approach which comes with a neglectable performane loss on memory intensive-, and drops maximally by 35% on certain CPU intensive opertaions. As a result, the monitored system remains stable and usable while being monitored.*

In contrast to previous out-of-the-guest system call tracing methods introduced by Ether [DRSL08b], I designed and implemented a new, and general method that is fully compatible with 64-bit systems. In 32-bit mode the `SYSENTER` instruction is used to prepare fast system calls. However, the `SYSENTER` instruction is not supported by AMD64, so we rely on another fast system call instruction (`SYSCALL`) that is compatible with both Intel and AMD processors.

In the following, I introduce the details of my novel system call tracing method for 64-bit mode. To monitor the system calls of an unmodified target OS from the hypervisor, I have to observe context changes induced by VMEXITs. However, similarly to `SYSENTER`, `SYSCALL` does

not generate VMEXIT by default when being executed, thus this problem have to be solved by a specific method. One way to handle this issue is advised in [DRSL08b] for the SYSENTER instruction. By translating this concept to SYSCALL, target address of this instruction has to be modified, which is stored in *STAR registers (STAR, CSTAR, LSTAR), to an address being paged out. When the processor accesses this address, a page fault is raised (#PF) and a VMEXIT context change occurs. By catching the page fault in the hypervisor, the original jump address of SYSCALL is reloaded into *STAR and it is re-executed. The problem with this solution is that the overwhelming number of page faults generated in an OS during its normal operation induces a large performance degradation which makes the OS drastically slower. Moreover, due to swapping, the number of raised page faults increases significantly on machines with lower memory capacity. Furthermore, a system with more processes generates more page faults as well.

For this reason, another approach has to be figured out, where the number of VMEXITs do not depend on the hardware and the number of executed applications. My new method, similarly to [PSE11], is based on invalid opcode exceptions instead of page faults to mitigate the performance degradation problem of previous solutions. As a first step of my implementation, I unset the SCE (system call enable) bit of the guest's EFER register, which makes SYSCALL instructions unknown for the processor. As a consequence, when being executed, an invalid opcode exception (#UD) is generated that induces a corresponding VMEXIT (VMEXIT_EXCEPTION_UD). At this point, the system call number can be retrieved in the hypervisor from the EAX register and the guest's user-space data structures via the GS selector. As the #UD exception does not increases the instruction pointer (RIP), I do not have to bother with alignments to re-execute the SYSCALL. In contrast to [PSE11],EFER.SCE bit is enabled here again for higher transparency. However, depending on the chosen transparency-granularity tradeoff, later I disable this bit again. To achieve this, I chose to generate another VMEXIT, which puts extra performance overhead to this solution, but makes the transparency of the analysis controllable. Thus, when an invalid opcode exception is handled, and the required features are extracted by reading the guest memory, a hardware breakpoint is inserted (by means of debug registers DR0 and DR7) to an address that has to be reached before the next chosen system call depending on the chosen transparency-granularity level. This breakpoint can either be placed on a user-, or kernel-space code, however, it predestinates if user-, or kernel-level objects can be extracted when handling it. See Figure 3 to read the details of the invalid opcode exception handler algorithm.

When the guest's instruction pointer reaches our hardware breakpoint, a VMEXIT_EXCEPTION_DB is raised in the processor that is handled by the registered trap. This handler disables system calls again by disabling EFER.SCE for the guest mode OS and deletes the breakpoint. Finally, OS objects can be extracted here as well. The details of this algorithm is described in Figure 4.

Algorithm for Handling Invalid Opcode Exceptions (#UD)

1. Read the guest's EFER value from VMCB (Virtual Machine Control Block)
2. If EFER.SCE is set, then RETURN.
3. Retrieve the candidate syscall number from VMCB.EAX.
4. If the syscall number is valid, then
   (a) Set the EFER.SCE bit. /* Enable syscalls again for the guest */
   (b) Set DR7.G0 and DR7.GE bits /* Enable global exceptions for DR0 */
   (c) Set DR0 to an appropriate address. /* Depends on data extraction */
   (d) Extract user-space data objects via VMI.

Figure 3: Algorithm for handling invalid opcode exceptions.

Algorithm for Handling Debug Exceptions (#DB)

1. Read the guest's `EFER` value from VMCB
2. Unset the `EFER.SCE` bit.
3. Unset `DR7.GO` and `DR7.GE` bits /* Disable global exceptions for `DR0` */
4. Extract data structures. /* Depends on CPL */

Figure 4: Algorithm for handling debug exceptions.

Another important contribution of this work is to demonstrate that this new system call tracing method guarantees better performance than the well-accepted method [DRSL08b, SG10] applied on x86 systems. First of all, I registered traps for #`UD` and #`PF` in the hypervisor to measure the number of invalid opcode exceptions and page faults that occur in a system by default. To demonstrate the performance differences between the two methods, I first counted the number of page faults and invalid opcode exceptions generated by the same system under various conditions. To achieve this, a trap handler is registered for #`UD` in NBP, and I used `TraceView`, available as a part of the Windows Driver Kit [Mic12], to count the number of page faults at the same time. Note that I could not measure the number of #`PF` with NBP due to the low response time of the monitored system. My measurements demonstrate that the number of page faults highly depends on the configuration of the machine such as the processes being launched.

Table 5: Counting the number of page faults and invalid opcode exceptions being raised under native operation on two CPU cores. The examined time interval is 2.5 minutes, and the statistics are calculated from samples with 12 elements.

	#UD		#PF	
Configuration	Mean	Std	Mean	Std
1. No extra process	0	0	16935	15839
2. Extra processes	0	0	43547	24155

To verify that extra processes increase the number of page faults, I measured the number of exceptions right after the reboot of the target system and after starting the benign applications. These results are summarized in Table 5. The most interesting part of this measurement is that I did not observe any invalid opcode exceptions (even with lengthened time window of 30 minutes) under the normal operation of the OS. On the other hand, as we can see the number of page faults highly depend on the active processes.

I also measured the performance loss of our system call tracing method by means of the Passmark Performance Test [Pas12]. I executed the tests 12 times in each configuration, and calculated the corresponding statistical metrics (mean and standard deviation) for them. I evaluated the performance of the CPU in native operation, when it is Blue-Pilled and when it is monitored by my extension.

As Table 6 demonstrates, NBP itself does not cause performance overhead to the system, so it offers ideal preconditions for our monitoring extension. When monitoring the system, the observed performance loss ranges between 30.7% (Floating Point) and 35.4% (Compression), which is still acceptable in practice as the OS actually remains usable. Note that I configured the system-call tracer to perform maximally, thus we re-enabled system call monitoring right after the previously evaluated `SYSCALL` instruction. However, the performance degradation of

Table 6: Results of Passmark CPU performance test for the unmonitored, blue-pilled and monitored system. The first column lists the operations executed by the test, while the numbers show the mean and standard deviation of execution times.

	Native		Blue-Pilled		Monitored	
Operations	Mean	Std	Mean	Std	Mean	Std
1. Integer Math (MOps/Sec)	344.5	2.40	346.0	1.13	228.0	82.35
2. Floating Point Math (MOps/Sec)	1072.3	8.30	1072.4	5.63	742.3	226.88
3. Find Primes (Thousand Primes/Sec)	322.9	3.22	323.7	1.14	220.1	74.44
4. SSE (Mill. Matrices/Sec)	9.4	0.06	9.4	0.10	6.3	2.17
5. Compression (KBytes/Sec)	2063.1	42.84	2072.6	6.99	1333.1	430.76
6. Encryption (MBytes/Sec)	9.7	0.03	9.7	0.03	6.1	2.08
7. Pyhsics (Frames/Sec)	95.9	1.98	95.6	1.99	63.7	17.57
8. String Sorting (Thousand Strings/Sec)	1243.6	27.86	1254.0	11.46	843.1	235.83

Table 7: Results of Passmark Memory performance test for the Unmonitored, Blue-Pilled and Monitored system. The first column lists the operations executed by the test, while the numbers show the mean and standard deviation of execution times.

	Native		Blue-Pilled		Monitored	
Operations	Mean	Std	Mean	Std	Mean	Std
1. Allocate Small Block (MBytes/Sec)	3348.4	33.0	3322.8	26.21	3067.1	217.8874
2. Read Cached (Mbytes/Sec)	1351.5	2.44	1351.7	2.22	1350.6	2.9738
3. Read Uncached (Mbytes/Sec)	1290.5	3.03	1286.7	2.92	1283.9	21.3572
4. Write (MBytes/Sec)	1256.8	8.67	1240.4	21.55	1243.9	43.5518
5. Large RAM (Operations/Sec)	2212.7	11.71	2193.4	27.83	2242.9	52.1760

the monitored system was still influenced by the executed debug print operations to verify the result of traces. Furthermore, I benchmarked the memory overhead of the system as well under the same conditions. As Table 7 shows, the monitoring extension induces negligible memory overhead in the range of 0.1% (Read Cached) and 8.5% (Allocate Small Blocks).

The related publications are [PLV+15, PB14, BPBF12b, BPBF12a].

2.2 New Attacks against Hardware Virtualization

THESES 3: *I propose two novel attacks against I/O virtualization, one is based on the modification of the Peripheral Component Interconnect express (PCIe) configuration space and the other is based on the creation of host-side Non-Maskable Interrupts (NMIs). While the former was discovered manually, the latter was revealed by my fuzzer, called PTFuzz, that I built to automatically reveal low-level problems during DMA operations. Additionally, this latter attack works on contemporary hardware and software even when all the available protection mechanisms are turned on. I demonstrate both of my attacks on Xen 4.2 and KVM 3.5. I show furthermore that such vulnerabilites are not unique as PTFuzz also identified other anomalies in Intel chipsets. My attacks strengthened the belief that both hardware and software vendors should raise the bar for more complete protection in their products.*

In the last ten years, a large number of papers [SLQP07, WJ10, MLQ+10, GAH+12, ANW+10, KSRL10] have been presented to either secure, or enhance the performance and capabilities of VMMs. Several works [WR11, Woj08b, Woj08a, RT08, LSLND10, Pat12] also mention and implement possible attacks. Unfortunately, most of them are described only from a theoretical point of view, and only a few have been actually implemented and thoroughly tested in realistic settings. Moreover, even when a proof-of-concept implementation exists, it is often difficult to understand what the prerequisites are for the attack to work, what the real impact is, and to which extent the results can be generalized to other environments and/or VMMs. Most of these questions are difficult to answer, and it is not uncommon also for experts to disagree on these points. Finally, to make things even more complex, current VMMs are rapidly evolving. Each new release contains new technologies that can potentially introduce new vulnerabilites as well as new countermeasures that can make existing attacks obsolete.

For example, several techniques have recently been introduced to increase the efficiency and security of I/O operation for guest virtual machines (VMs). *Direct device assignment* (also known as device passthrough) is such a mechanism, where the VMM assigns a device exclusively to one VM instead of sharing it with other virtual machines. This is achieved by directly mapping the device into a VM address space, redirecting the corresponding interrupts to the correct VM. Clearly, assigning the hardware to be directly controlled by a VM improves the performance. At the same time, this approach also introduces a wide range of security problems that eventually led hardware manufacturers to introduce hardware assisted protection extensions for the CPU and chipset.

For this reason, I propose new attacks against I/O virtualization for contemporary hardware and software. Each of my attacks was executed against different hardware and VMM configurations, according to two possible attack scenarios. In the first scenario, I assume that the attacker has full access to a guest machine configured with a pass-through device. This is a common setup for IaaS cloud providers that offer, for example, direct access to video cards (e.g., Amazon EC2 Cluster GPU).

In the second scenario, I assume that the attacker is able to control or compromise the privileged VM. Even though this case is certainly more difficult to achieve, it still represents an important threat model that needs to be carefully evaluated. However, unrestricted access to the privileged VM does not give full privileges over the physical machine [Woj08b]. In other words, the VMM is specifically designed to be protected against a malicious privileged VM.

Attacks were also launched on KVM's host OS to identify the differences with Xen's privileged guest VM (i.e., Dom0). Note that I performed this test only for completeness, as host OS privileges on KVM are equivalent of having entire control over the VMM as well.

THESIS 3.1: *I propose new Memory Mapped I/O attack via the configuration space of PCI Express devices against legacy VMMs.*

On the x86 architecture, a PCI device can be accessed in two different ways: using a Port Mapped I/O (PIO) or using a Memory Mapped I/O (MMIO) mechanism. Each PCI device configuration is stored in the device configuration memory. This memory is accessible either by using special PIO registers or through an MMIO space.

The configuration space is typically accessed by the BIOS or the operating system kernel to initialize or configure the Base Address Registers (BAR). Base Address Registers are defined by the PCI standard and used to specify the address at which the device memory is mapped in the PIO or MMIO address spaces.

Access to configuration space registers is usually emulated for fully virtualized guests, and in some cases also for privileged VMs. In this case, whenever a guest accesses a configuration space, the request is intercepted by the VMM, which incurs a significant performance overhead. Therefore, in order to improve the performance, some VMMs (e.g., KVM) allow to directly pass PIO or MMIO accesses [YBYW08], except for the accesses targeting the device configuration memory.

PCI Express (PCIe) devices have an extended configuration space that can be accessed via traditional memory operations (i.e., MMIO). To test this situation, I implemented a *new* device configuration space attack that can be launched via the MMIO with the goal of manipulating the memory mapped registers of the target device similarly to previously known PIO attacks [DA07].

To achieve this, I addressed the PCIe configuration space of the targeted devices by using their source-id [Fle08], and then tried to modify some of their configuration registers (e.g., BAR). Similarly to the PIO attack, my MMIO attacks works on legacy VMMs, where the MMIO address space is not emulated.

THESIS 3.2: *I propose a Non-maskable Interrupt (NMI) injection attack which was tested and verified on both Xen 4.2 and KVM 3.5.0, however, every VMM, which runs on a platform with System Error Reporting enabled, can be affected. In order to be sure that the attack overcomes all the available protection mechanisms, I enabled DMA and interrupt remapping as well as x2APIC mode on the privileged VM/host (this configuration was known to be safe against all known interrupt attacks).*

To test for the presence of low-level problems in the interrupt generation and handling phase, I designed and implemented a tool called PTFuzz, by extending Intel's e1000e network driver. PTFuzz is optimized to launch any type of Message-signalled Interrpt (MSI) by fuzzing both the MSI *address* and its *data* components as well as the size of DMA requests. It works by writing data (i.e., MSI data component) to the LAPIC MMIO range using DMA. As PTFuzz is capable of fuzzing each field of an MSI separately, it can be fine-tuned to create both *compatibility* and *remappable* MSIs formats. The operation of PTFuzz can be summarized in a few steps:

1. Prepare a transmission buffer (TXb) in the guest OS, and populate it with the MSI *data* component.
2. Prepare a receiver buffer (RXb) in the guest OS.
3. Change the physical address of the RXb buffer according to the MSI *address* component to point to the memory mapped interrupt space (i.e., MMIO LAPIC).
4. Move the MSI data component via a DMA transaction into the card's internal TX buffer.

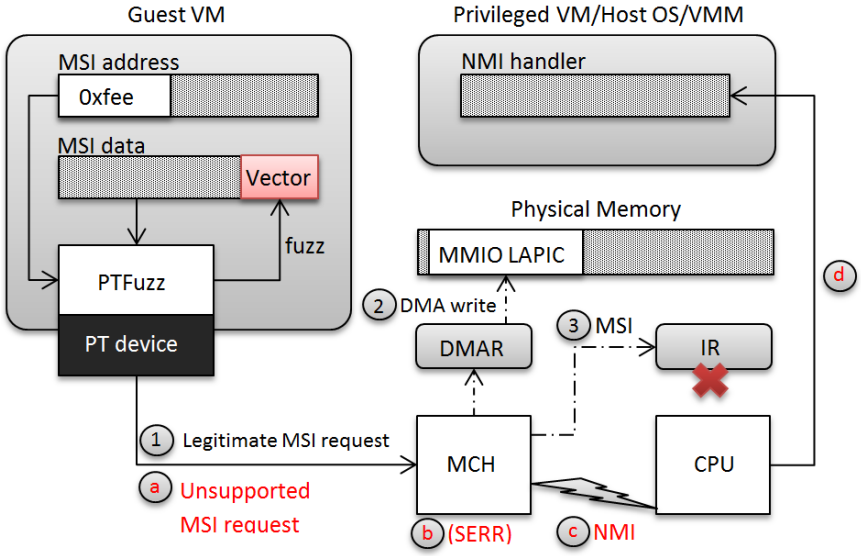

Figure 5: Interrupt generation by PTFuzz. This figure describes two interrupt generation cases indicated by the *numerical* and *alphabetical* paths. On the *numerical* path, PTfuzz requests a legitimate MSI (1) by a DMA write operation to the MMIO LAPIC (2) which is first verified by the DMA remapping engine (DMAR). As a result, a compatibility-format MSI is generated (3) that is blocked by the interrupt remapping engine (IR). The *alphabetical* path, however, shows *my* unsupported MSI request (a), which the platform detects and blocks. However, when System Error Reporting is enabled, the platform sets the SERR status bit on the Memory Controller Hub (MCH) PCI Device (b). As a result, a host-side Non-maskable Interrupt (NMI) is *directly* delivered to the physical CPU (c) executing the privileged VM/host OS/VMM. SERR induced NMIs, however, may cause host software halt or trigger the host-side NMI handler (d) which opens the door for Guest-to-VMM escapes.

5. Send the data in loopback mode into the card's RX buffer.

6. Move the MSI data from the card's internal RX buffer into the corresponding MMIO LAPIC address range specified by the MSI address (0xfeexxxxx) with a given DMA request size.

7. If the MSI data component is fuzzed, then select a new MSI data value and repeat from Step 1.

8. If the MSI address component is fuzzed, then select a new MSI address value and repeat from Step 3.

Fuzzing the entire MSI data and address spaces would require an extensive amount of work to manually verify and validate each result. For this reason, I decided to focus our effort on those MSI fields that were either more interesting from an attacker's point of view, or had clear constraints set by the vendors.

In particular, here I present the results I obtained by fuzzing the *vector* field of a compatibility format MSI *data* component and the *don't care* field of a remappable format MSI *address* component. Whenever I observed an unexpected hardware behavior as a result of our test cases, I instrumented the code of the VMM to collect all the information required to understand the problem in detail.

In the first experiment, I fuzzed the *vector* field of the compatibility format MSI *data* as well as the size of the MSI request. During the tests, I noticed that the VMM/privileged VM

received a *legacy* Non-Maskable Interrupt (NMI) for some values of the *vector*. This happens even when all the existing hardware protections mechanisms were turned on. In addition, I got the same results when the size of the MSI request had not conformed with the required MSI transmission size (i.e., it was not 32-bit long).

My tests indirectly generated a *host-side legacy* NMI to one of the physical CPUs (i.e., Bootstrap Processor - BSP). More precisely, as a result of performing an unsupported MSI request by a DMA transaction to the memory mapped interrupt space, the platform blocks the MSI request and raises a PCI System Error (SERR#) which is delivered as NMI to report hardware errors. In that case, the SERR status bit is set by the platform on the Memory Controller Hub - MCH (BDF 00:00.0) PCI device. Thus, the unchecked host-side NMI is forwarded to the physical CPU executing the privileged VM/host OS/VMM. Depending on privileged VM/host OS/VMM kernel configuration, such an NMI may be handled by the privileged VM/host OS/VMM or can result in a host software halt (*panic*). Figure 5 gives a high-level overview about the attack. When I took a closer look at this issue, I noticed that the NMI was spawned when a compatibility format MSI is requested with vector numbers below 16 or with an invalid request size (i.e., not 32-bit long). The reason for the former lies in the fact that MSI cannot deliver interrupts with vector less than 16 [Int12].

The *main* difference with previous interrupt attacks is that my NMI injection attack works on configurations where interrupt and DMA remapping is *enabled*. In order to be sure that the attack overcomes all the available protection mechanisms, I enabled DMA and interrupt remapping as well as x2APIC mode on the privileged VM/host (this configuration was known to be safe against all known interrupt attacks). I successfully verified my NMI injection attack on both Xen 4.2 and KVM 3.5.0, however, every VMM, which runs on a platform with System Error Reporting enabled, can be affected. The attack successed as the NMI was indirectly spawned by the Memory Controller Hub (and *not* by the passthrough device) which is handled by the host.

More precisely, my interrupt attack (i.e., XSA-59, CVE-2013-3495) is the consequence of a misunderstanding between the hardware and software which allows guest-to-VMM escapes [4] independently from the VMM software itself. While this attack has been first reported to vendors (e.g., Xen, KVM, VMware) during the spring of 2013, no real solution could still be employed. The reason for this lies in the fact that hypervisor vendors have to handle every Intel CPU chipset separately [5] so as not to limit key functionalities with their workarounds. Unfortunately, the list of these chipsets is endless, so the vulnerability cannot be put to rest in future hypervisor releases.

THESES 4: *I propose new techniques to detect malware analysis platforms which build upon hardware assisted virtulization. These techniques are verified against the out-of-the-guest malware analysis platform Ether which is implemented atop Intel CPUs with hardware-assisted virtualization extensions*

Malware analysis can be an efficient way to combat malicious code, however, miscreants are constructing heavily armoured samples in order to stymie the observation of their artefacts. Security practitioners make heavy use of various virtualization techniques to create sandboxing environments that provide a certain level of isolation between the host and the code being analyzed. However, most of these are easy to be detected and evaded. The introduction of hardware assisted virtualization (Intel VT and AMD-V) made the creation of novel, *out-of-the-guest* malware analysis platforms possible. These allow for a high level of transparency by

[4] Theoretically arbitrary code execution is possible, but current system implementations are vulnerable to host system halt (DoS)

[5] `http://www.gossamer-threads.com/lists/xen/devel/360060`

residing completely outside the guest operating system being examined, thus conventional *in-memory* detection scans are ineffective. Furthermore, such analyzers resolve the shortcomings that stem from inaccurate system emulation, in-guest timings, privileged operations and so on.

THESIS 4.1: *I propose a generic technique to detect platforms using hardware-assisted virtualization by verifying the presence of CPU-specific design defects (i.e., errata). I successfully implemented and tested these errata as a part of a Windows kernel driver.*

The detection of system monitors through CPU errata has already been discussed for CPU identification in [RKK07], however, here only the QEMU hardware emulator was tested with this technique. Most of the errata require kernel mode instructions, thus must have been implemented in a driver component. On the other hand, these CPU deficiencies or bugs are strongly bound to CPU models, thus the feature tests can be effective only on certain CPU families (e.g., Intel Core 2 Solo). Since my test environment is built atop a machine with Intel Core 2 Duo E6600 CPU the following erratum exploits a vulnerability of the Core 2 Duo family [Cor10]. In the following, I propose an erratum that I implemented and tested to detect hardware-assisted virtualization.

The AH4 Erratum states that " *VERW/VERR/LSL/LAR Instructions May Unexpectedly Update the Last Exception Record (LER) MSR* " and there is no planned fix for it. The problem is that the `LER MSR` is updated in certain cases for unknown reasons, if the resultant value of the `Zero Flag` (`ZF` flag of `EFLAGS` register) equals zero after the execution of the instructions above. The *Last Exception Record MSR* comprises two registers called `MSR_LER_FROM_LIP` and `MSR_LER_TO_LIP` situating at register addresses `0x1dd` and `0x1de`, respectively. The former is the abbreviation of *Last Exception Record From Linear Instruction Pointer* which points to the last branch instruction (conditional/ unconditional jumps, call, etc.) that had been taken by the processor before the last exception occurred or the last interrupt was handled. The latter refers to *Last Exception Record To Linear Instruction Pointer* which stores the address of the target of the last branch instruction that had been executed by the processor before the last exception occurred or the last interrupt was handled.

The " *Verify a Segment for Reading or Writing*" instructions verify if a code or data segment is readable/writeable from the current privilege level (CPL) according to [Int09]. This erratum requires kernel mode operation as it reads privileged resources (LER MSR), thus it is implemented as a part of our device driver. By providing the cleared `AX` and `CX` registers for `VERR` and `VERW` instructions the resultant `ZF` flag is zero for sure as invalid segment pointers (NULL) are given to the source operands. Last, but not least the value of the *Last Exception Record To Liner IP* MSR is read with the privileged `__readmsr(MSR address)` Visual C++ command at register address `0x1de`.

```
__asm{
  xor eax, eax
  xor ecx, ecx
  verr cx
  verw ax
}
ret = __readmsr(0x1de);
```

A processor erratum is a design fault so its existence is unintended. Consequently, hardware assisted virtualization solutions (e.g., Xen) will not implement them in the exposed virtual CPUs of guests because it would take too much effort and make no sense to mimic unexpected system behaviours, however, a higher level of transparency could be provided.

	Number of updates	
N	*Native*	*Xen*
100	59	0
1000	650	0
10000	4232	0
100000	20870	0

Table 8: The number of updates

Table 8 demonstrates the execution of this feature test. First of all, the CPU erratum was executed 100, 1000, 10000 and 100000 times under the corresponding environments. As the results show, the erratum has occurred only in native environment, thus it is an evident detector for hardware assisted virtualization.

THESIS 4.2: *I propose new detection attacks against the* out-of-the-guest *malware analysis framework* Ether *[DRSL08a]. More specifically, I propose an in-guest timing attack which was supposed to be prevented by Ether, but for various reasons, it actually does not prevent it.*

I propose an *in-guest* timing feature test that can detect Ether by exploiting a practical weakness. To understand how our feature test works I introduce the operation of `RDTSC` instruction in VMX non-root mode. Its value depends on the "RDTSC exiting", "use TSC offsetting" VM-execution control fields and the `TSC_OFFSET` value. VM-execution control fields allow the VMM to control the operation of certain instructions like the aforementioned `RDTSC`, while `TSC_OFFSET` is a special field of the Intel VT architecture that is added to the value of the returned Time-Stamp Counter in defined cases. The `RDTSC` instruction in VMX non-root mode works in one of the following three options. The first option is that if both the "RDTSC exiting" and "use TSC offsetting" control fields are zero, an in-guest `RDTSC` can operate normally, thus it returns the 64-bit Time-Stamp Counter value. If the "RDTSC offsetting" field is zero and the "use TSC offsetting" is 1, then it responds with the sum of the `IA32_TIME_STAMP_COUNTER` MSR and the value of the `TSC_OFFSET` field. The last option is that if the "RDTSC exiting" control field is 1, then it causes a VMExit, which can be intercepted by the VMM to adjust the Time-Stamp Counter value. This is the technique that Ether applies, however, the 64-bit `TSC` value returned to the guest is adjusted in a deterministic manner as the code snippet (pasted from ether_lenny.patch source file) below demonstrates it.

```
+/* maintain a monotonic fake TSC counter */
+u64 ether_get_faketime(struct vcpu *v)
+{
+ return ++v->domain->arch.hvm_domain.
                .ether_controls.faketime;
+}
```

As the above mentioned code snippet is only responsible for increasing the timer monotonically, the authors of Ether proposed to exactly adjust the logical timer by the use of the `TSC_OFFSET` field of the Intel VT architecture. Significant to emphasize that this routine is called only at the time when the guest exited with an `RDTSC`, thus nowhere else the TSC is being incremented. In theory, the task of `TSC_OFFSET` would be to adjust the returned Time-Stamp Counter value conforming to a native system. Contrary to the original paper [DRSL08b], the `TSC_OFFSET` field is not updated by the Ether implementation and does not contain the difference value between the execution time of Ether's exception handler and that of the native handler

routine. The "TSC offsetting" VM-execution control field is being disabled while "RDTSC exiting" is being enabled in the minimum configuration, as the following source code taken from the Ether-patched Xen source file vmcs.c shows.

```
void vmx_init_vmcs_config(void)
{
  ...
    min = (CPU_BASED_HLT_EXITING |
           CPU_BASED_INVDPG_EXITING |
           CPU_BASED_MWAIT_EXITING |
           CPU_BASED_MOV_DR_EXITING |
           CPU_BASED_ACTIVATE_IO_BITMAP |
           //CPU_BASED_USE_TSC_OFFSETING |
           CPU_BASED_RDTSC_EXITING);

    _vmx_cpu_based_exec_control =
    adjust_vmx_controls( min, opt,
    MSR_IA32_VMX_PROCBASED_CTLS_MSR);
 ...
}
```

Due to the fact that `TSC_OFFSET` is neglected and the TSC is increased by one for one in-guest `RDTSC` call, the TSC difference between two `RDTSC` instructions is 1. More precisely, this is the case when the code being executed in Ether is not analysed. According to my observations whenever a code is under analysis (e.g., instruction traced) there are CPU time slots for other guest processes to invoke additional `RDTSC`s. Due to that fact there is a varying TSC difference (∼9-171) value between any two `RDTSC`s of the analysed code we introduce below. This difference is still so tiny that a general timing attack found in various malware samples (measuring the TSC difference before and after the execution of an instruction) will not be successful. However, this approach still does not enable Ether to stay completely transparent as this kind of operation deviates much from the normal operation of `RDTSC`.

In the following, I propose a practical feature test against Ether which a bit different from a conventional *in-guest* timing attack as it builds upon the fact that each instruction of an instruction sequence increments the TSC with a predefined value depending on the CPU family. For example, in case of Core 2 Duo processors the TSC is incremented at a constant rate by either the maximum core-clock to bus-clock ratio of the processor or by the maximum resolved boot-time frequency [Int09]. Supposing that each value is bigger than 1, a program that estimates the extent of increment can be used as a feature test. The code sample below initializes a `NOP` loop that increments the TSC at a constant rate in each iteration, thus the resulting difference between the initial and the final execution of `RDTSC` should be at least equal to the length of the loop (i.e.,2000).

```
      mov ecx, 2000
      cpuid
      rdtsc
      xchg ebx, eax
lb:   nop
      loop lb
      cpuid
      rdtsc
      sub eax, ebx
      cmp eax, 2000
      jbe det
```

As RDTSC is a non-serializing instruction, it may be executed after the subsequent or before the previous instructions if their execution takes a while. That is, a serializing instruction, e.g., CPUID, is put before reading the counter. Furthermore, the loop must contain a non-privileged instruction which does not cause a VMExit so as to prevent its execution from being manipulated in the VMM. Note that in the code sample I examine only the lower 32 bits of RDTSC as the faked TSC has never exceeded this interval for our tests. Since RDTSC has been discussed above the behaviour of it can be used in other two modes. If it works normally, the presence of a debugger can be detected with conventional timing methods. If it returns the sum of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC_OFFSET field the case is the same as with normal operation since the VMM cannot manipulate the TSC value conforming to the length of the loop. Thus, these two other options could not help to adjust the *in-guest* timer. However, in my opinion the implementation of correct faked timing could have been solved with some efforts, thus it is not a theoretical problem.

The related publications are [PBB11, PBB13, PLS+14].

3 Conclusion

3.1 New Methods for Detecting Unknown Malware Infections

Detection of modern, unknown malware is an exceedingly important problem today to solve. This arguement is especially true when targeted samples are under consideration. Due to the increasing demand to pinpoint sophisticated unknown malicious code that show no similarity with known and identified samples, security researchers and companies are about to propose proper detection methods. As a response to this problem, I propose a novel memory forensics solution called Membrane which uses paging event analyis to detect even previously unseen code injection attacks. Membrane is designed to work with memory snapshots as well as a live system to alarm upon detecting diverted paging behaviours in examined processes. Thus, Membrane fits well into comtemporary virtualized IT infrastructures where, for example, the creation of periodic snapshots from monitored systems has already been solved. Additionally, Membrane is able to detect a wide-range of code injection techniques, as it focuses on the derailed paging event cardinalities which code injections cause. More precisely, I reached 86-98% true positive rate when malware injected into `explorer.exe` which is the noisiest system process according to my observations. I also prove that this detection accuracy can be improved significantly for other system processes.

There are, however, corner cases when we work with hardware and software which are operating as a part of live systems (e.g., critical infrastructure). Such systems do not tolerate downtime, thus they cannot be rebooted or halted to migrate them into virtualized infrastructures. For this reason, I propose an on-the-fly installable system monitoring framework by extending the New Blue Pill HVM rootkit to meet the requirements of live systems. I also designed and implemented a novel system call tracing method that promises long-term compatibility with current 64-bit systems as well as allows for configurable granularity and transparency for catching system calls. In contrast to previous methods that mainly used page faults, my approach is based on system call invalidation which offers more acceptable performance.

3.2 New Attacks against Hardware Virtualization

Due to the increasing significance of hardware virtualization in cloud solutions, it is important to clearly understand existing and arising VMM-related threats. For this reason, I design and implement a fuzzer called PTFuzz to find interrupt-related attacks triggerable via passthrough devices being attached to an unprivileged guest environment. This tool successfully detected various unexpected hardware behaviors while running on commodity Virtual Machine Monitors (VMMs). For example, I discovered and implemented an interrupt attack that leverages unexpected hardware behaviour to circumvent all the existing protection mechanisms in commodity VMMs. To the best of my knowledge, this is the first attack that exhibits such behaviour and to date it seems that there is no easy way to prevent it on Intel platforms. As my threat model fits well into contemporary cloud setups (i.e., IaaS clouds), I believe that my work can help cloud operators to better understand the limitations of their current architectures to provide secure hardware virtualization and to prepare for future attacks.

Then, I propose a generic method to detect the fact of hardware-assisted virtualization by pinpointing subtle differences in the execution of given instruction sequences which trigger CPU errors when no virtualization is present. This method clearly shows that even hardware-assisted virtualization (but not the fact of analysis) can be detected with high accuracy. Furthermore, I suggest an in-guest timing attack against the out-of-the-guest malware analysis framework called Ether. My results here show that it is a difficult challenge to guarantee perfect transparency for hardware virtualization-based malware analyzers.

References

[Ali14] AlienVault. Batchwiper: Just another wiping malware. `https://www.alienvault.com/open-threat-exchange/blog/batchwiper-just-another-wiping-malware`, accessed on Nov 13, 2014.

[ANW+10] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 38–49, New York, NY, USA, 2010. ACM.

[BPBF12a] B. Bencsáth, G. Pék, L. Buttyán, and M. Félegyházi. Duqu: Analysis, detection, and lessons learned. In *Proceedings of the ACM European Workshop on System Security (EuroSec)*, 2012.

[BPBF12b] Boldizsár Bencsáth, Gábor Pék, Levente Buttyán, and Mark Felegyhazi. The cousins of Stuxnet: Duqu, Flame, and Gauss. *Future Internet*, 4(4):971–1003, 2012.

[CER14] CERT.PL. More human than human - Flames code injection techniques. `http://www.cert.pl/news/5874/langswitch_lang/en`, accessed on Nov 13, 2014.

[Cor10] Intel Corporation. Intel®Core™2 Duo Processor for Intel®Centrino®Duo Processor Technology Specification Update. `http://download.intel.com/design/mobile/SPECUPDT/31407918.pdf`, September 2010.

[DA07] Loïc Duflot and Laurent Absil. Programmed I/O accesses: a threat to Virtual Machine Monitors? *PacSec*, 2007.

[DRSL08a] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 51–62, New York, NY, USA, 2008. ACM.

[DRSL08b] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 51–62, New York, NY, USA, 2008. ACM.

[Fle08] Sam Fleming. Accessing PCI Express* Configuration Registers Using Intel Chipsets. December 2008.

[G D14] G DATA SecurityLabs. Uroburos, Highly complex espionage software with Russian roots. `https://public.gdatasoftware.com/Web/Content/INT/Blog/2014/02_2014/documents/GData_Uroburos_RedPaper_EN_v1.pdf`, accessed on Nov 1, 2014.

[GAH+12] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. ELI: bare-metal performance for I/O virtualization. *SIGARCH Comput. Archit. News*, 40(1):411–422, March 2012.

[IM07] Nwokedi Idika and Aditya P Mathur. A survey of malware detection techniques. Technical report, Purdue University, 2007.

[INe14] INetSim. `http://www.inetsim.org/`, accessed on Nov 10, 2014.

[Int09] Intel Corporation. Intel®64 and IA-32 Architectures Software Developer's Manual, June 2009.

[Int12] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual - Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C, Aug 2012.

[KSRL10] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. Nohype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 350–361, New York, NY, USA, 2010. ACM.

[LMP+14] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference*, December 2014. To Appear.

[LSLND10] Fernand Lone Sang, Eric Lacombe, Vincent Nicomette, and Yves Deswarte. Exploiting an I/OMMU vulnerability. *MALWARE*, 2010.

[Man13] Mandiant. APT1: Exposing One of China's Cyber Espionage Units. `http://intelreport.mandiant.com/`, 2013.

[Mic12] Microsoft. Windwos Driver Kit. `http://msdn.microsoft.com/en-us/windows/\\hardware/gg487428`, Last accessed, April 02, 2012.

[MLQ+10] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy*, pages 143–158, 2010.

[Pas12] Passmark Software. Passmark Performance Test. `http://www.passmark.com/\\download/pt\_download.htm`, Last accessed, March 26, 2012.

[Pat12] Patrick Stewin and Iurii Bystrov. Understanding DMA Malware. In *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2012.

[PB14] G. Pek and L. Buttyan. Towards the automated detection of unknown malware on live systems. In *IEEE International Conference on Communications (ICC)*, pages 847–852, June 2014.

[PBB11] Gábor Pék, Boldizsár Bencsáth, and Levente Buttyán. nEther: In-guest detection of out-of-the-guest malware analyzers. In *Proceedings of the ACM European Workshop on System Security (EuroSec)*, EUROSEC '11, pages 1–6, 2011.

[PBB13] Gábor Pék, Levente Buttyán, and Boldizsár Bencsáth. A survey of security issues in hardware virtualization. *ACM Comput. Surv.*, 45(3):40:1–40:34, July 2013.

[PLS+14] Gábor Pék, Andrea Lanzi, Abhinav Srivastava, Davide Balzarotti, Aurélien Francillon, and Christoph Neumann. On the feasibility of software attacks on commodity virtual machine monitors via direct device assignment. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 305–316, New York, NY, USA, 2014. ACM.

[PLV+15] G. Pek, Zs. Lazar, Z. Varnagy, M. Felegyhazi, and L. Buttyan. Membrane: Detecting malware code injection by paging event analysis. In *Proceedings of the 24th USENIX Security Symposium (submitted)*, USENIX Security'15, pages 1–16, August 2015.

[PSE11] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Nitro: Hardware-based system call tracing for virtual machines. In *Advances in Information and Computer Security*, volume 7038 of *Lecture Notes in Computer Science*, pages 96–112. Springer, November 2011.

[RDG+12] Christian Rossow, Christian J Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten Van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *2012 IEEE Symposium on Security and Privacy*, pages 65–79. IEEE, 2012.

[Rea14] ReactOS. A free open source operating system based on the best design principles found in the Windows NT architecture. `http://doxygen.reactos.org`, accessed on Nov 8, 2014.

[RKK07] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. Detecting system emulators. In *Information Security Conference (ISC 2007)*, Oct 2007.

[RSI12] Mark Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals, Sixth Edition*. Microsoft Press, 6th edition, 2012.

[RT08] Joanna Rutkowska and Alexander Tereshkin. Bluepilling the Xen Hypervisor - Xen 0wning Trilogy part III. *Black Hat USA*, aug 2008.

[SG10] Abhinav Srivastava and Jonathon T. Giffin. Automatic discovery of parasitic malware. In *RAID*, pages 97–117, 2010.

[SLQP07] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 335–350, New York, NY, USA, 2007. ACM.

[Vol14] Volatility. The Volatility Framework. `https://code.google.com/p/volatility/`, accessed on Nov 13, 2014.

[Wil11] Carsten Willems. Internals of windows memory management (not only) for malware analysis. Technical report, Ruhr Universität Bochum, 2011.

[WJ10] Zhi Wang and Xuxian Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 380–395, Washington, DC, USA, 2010. IEEE Computer Society.

[Woj08a] Rafal Wojtczuk. Preventing and Detecting Xen Hypervisor Subversions - Xen 0wning Trilogy part II. *Black Hat USA*, aug 2008.

[Woj08b] Rafal Wojtczuk. Subverting the Xen Hypervisor - Xen 0wning Trilogy part I. *Black Hat USA*, aug 2008.

[WR11] Rafal Wojtczuk and Joanna Rutkowska. Following the White Rabbit: Software attacks against Intel®VT-d technology, April 2011.

[YBYW08] Ben-Ami Yassour, Muli Ben-Yehuda, and Orit Wasserman. Direct device assignment for untrusted fully-virtualized virtual machines. Technical report, 2008.

4 Publication of New Results

International Journal Papers

[J1] B. Bencsáth, G. Pék, L. Buttyán, and M. Felegyhazi. The cousins of Stuxnet: Duqu, Flame, and Gauss. *Future Internet*, 4(4):971–1003, 2012.

[J2] G. Pék, L. Buttyán, and B. Bencsáth. A survey of security issues in hardware virtualization. *ACM Comput. Surv.*, 45(3):40:1–40:34, July 2013.

International Conference and Workshop Papers

[C1] B. Bencsáth, G. Pék, L. Buttyán, and M. Félegyházi. Duqu: Analysis, detection, and lessons learned. In *Proceedings of the ACM European Workshop on System Security (EuroSec)*, 2012.

[C2] G. Pék, B. Bencsáth, and L. Buttyán. nEther: In-guest detection of out-of-the-guest malware analyzers. In *Proceedings of the ACM European Workshop on System Security (EuroSec)*, EUROSEC '11, pages 1–6, 2011.

[C3] G. Pek and L. Buttyan. Towards the automated detection of unknown malware on live systems. In *IEEE International Conference on Communications (ICC)*, pages 847–852, June 2014.

[C4] G. Pék, A. Lanzi, A. Srivastava, D. Balzarotti, A. Francillon, and C. Neumann. On the feasibility of software attacks on commodity virtual machine monitors via direct device assignment. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 305–316, New York, NY, USA, 2014. ACM.

[C5] G. Pek, Z. Lazar, Z. Varnagy, M. Felegyhazi, and L. Buttyan. Membrane: Detecting malware code injection by paging event analysis. In *Proceedings of the 24th USENIX Security Symposium (submitted)*, USENIX Security'15, pages 1–16, August 2015.

Other

[O1] A. Laszka, A. R. Varkonyi-Koczy, G. Pék, and P. Varlaki. Universal autonomous robot navigation using quasi optimal path generation. In *4th IEEE International Conference on Autonomous Robots and Agents (ICARA)*, February 2009.

[O2] G. Pék, L. Buttyán, and B. Bencsáth. Consistency verification of stateful firewalls is not harder than the stateless case. *Infocommunications Journal*, LXIV(2009/2-3), 2009.

[O3] G. Pék, A. Laszka, and A. R. Varkonyi-Koczy. An improved hybrid navigation method. In *7th International Conference On Global Research and Education in Intelligent Systems (Inter-Akademia)*, September 2008.