Malware Collection and Analysis via Hardware Virtualization

Tamas Kristof Lengyel
University of Connecticut
2015

Abstract

Malware is one of the biggest security threat today and deploying effective defensive solutions requires the collection and rapid analysis of a continuously increasing number of samples. The collection and analysis is greatly complicated by the proliferation of metamorphic malware as the efficacy of signature-based static analysis systems is greatly reduced. While honeypots and dynamic malware analysis has been effectively deployed to combat the problem, significant challenges remain.

The rapidly increasing number of malware samples poses a particular challenge as it greatly inflates the cost of the hardware required to process the influx. As modern malware also deploys anti-debugging and obfuscation techniques, the time it takes to formulate effective solutions is further exacerbated. There is a clear need for effective *scalability* in automated malware collection and analysis.

At the same time, modern malware can both detect the monitoring environment and hide in unmonitored corners of the system. It has also been observed that malware modifies its run-time behavior to lead the analysis system astray when it detects a monitoring environment. Consequently, it is critical to create a *stealthy* environment to hide the presence of the data collection from the external attacker. Such systems

also need to *isolate* critical system components from the executing malware sample while keeping the concurrent collection and analysis sessions separate.

Furthermore, the *fidelity* of the collected data is essential for effective dynamic analysis. As rootkits now employ a variety of techniques to hide their presence on a system, the broader the scope of data collection, the more likely the analysis will reveal useful features.

Over the last decade hardware virtualization has been proposed to develop such tools with promising results. In this dissertation we present a systematic evaluation of hardware virtualization as an underlying technology to construct effective malware collection and analysis systems. The evaluation is realized via the combination of four distinct objectives such systems need to fulfill: scalability, stealth, fidelity and isolation.

Malware Collection and Analysis via Hardware Virtualization

Tamas Kristof Lengyel

B.Sc., University of Connecticut, 2008

M.Sc., University of Connecticut, 2015

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

at the

University of Connecticut

2015

APPROVAL PAGE

Doctor of Philosophy Dissertation

Malware Collection and Analysis via Hardware Virtualization

Presented by

Tamas Kristof Lengyel, B.Sc., M.Sc.

Co-major advisor _____
Dr. Laurent Michel

Co-major advisor _____
Dr. Aggelos Kiayias

Associate advisor _____
Dr. Bing Wang

Associate advisor _____
Dr. Alexander A. Shvartzmann

Associate advisor _____
Dr. Bryan D. Payne

University of Connecticut

2015

Acknowledgments

I would like to express my appreciation and thanks to my advisory committee, Professor Dr. Aggelos Kiayias, Professor Dr. Laurent Michel, Professor Dr. Bing Wang, Professor Dr. Alexander Shwartzmann and Dr. Bryan D. Payne. A special thanks to Professor Dr. Claudia Eckert for her supervision during my visit at the Technische Universität München and to Professor Dr. John Chandy for his continued support of me as a GAANN fellow. I would like to thank you all for encouraging my research and for allowing me to grow as a research scientist. Your advice on both research as well as on my career have been priceless.

A special thanks to my family. Words cannot express how grateful I am to my mother and father for all of the sacrifices that you've made on my behalf. I would also like to thank all of my friends and colleagues who supported me in writing, and inspired me to strive towards my best. I'm grateful to Thomas Kittel, Steve Maresca, Justin Neumann, Jonas Pfoh, Jacob Torrey, George Webster and Sebastian Vogl for all the help and guidance during my research. I would like to also thank the countless open-source developers who have helped and inspired me during this research.

Finally and most importantly I would like to express appreciation to my beloved Bori who was always my support. Being able to share this journey with you has been an incredible blessing.

Credits

I wish to express my thanks for the privilege to co-author several papers over the years which have been instrumental in writing this Thesis. In particular, I would like to thank Jonas Pfoh, Sebastian Vogl, Steve Maresca and Thomas Kittel for their contributions to the un-published paper *Virtual machine introspection: a decade in review*, components of which aided the formation of the Related work section of this Thesis. I would also like to thank Justin Neumann, Steve Maresca, Bryan D. Payne and Aggelos Kiayias for reviewing and editing *Virtual Machine Introspection in a Hybrid Honeypot Architecture* and *Towards Hybrid Honeynets via Virtual Machine Introspection and Cloning*, publications which form the base of the Malware collection section of this thesis. Thanks to Steve Maresca, Bryan D. Payne, George Webster, Sebastian Vogl and Aggelos Kiayias for the discussions, reviewing and editing of *Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System* which has been constructive in the formation of the Malware analysis section of this Thesis. Additional thanks to Jonas Pfoh, Thomas Kittel, George Webster, Jacob Torrey and Claudia Eckert for their insight, reviewing and editing of *Pitfalls of virtual machine introspection on modern hardware*, *Multi-tiered Security Architecture for ARM via the Virtualization and Security Extensions* and *Virtual Machine Introspection with Xen on ARM* which aided in forming the Hardware and Software limitations section of this Thesis. Finally, I would like to thank all other co-authors for the privilege to work together on our publications on topics unrelated to this Thesis.

Contents

List of Figures

List of Tables

1 Introduction

Over the last decade malware has become a central tool used by criminal organizations to conduct crime over the Internet. Based on the financial implications of the widespread occurrence of security breaches, an in-depth understanding of malware internals is critical for designing and deploying effective defensive solutions.

The automation of malware collection with the use of honeypots has been an effective tool in the battle against malware. In the last decade there has been a concentrated effort to push honeypots to virtualized environments to further improve the efficacy of these tools [4, 58, 59]. Virtualized environments provide many benefits for honeypots as they simplify the containment and isolation of infections. Virtualization also provides easy and convenient methods for reverting a compromised honeypot to a clean state after the required data and artifacts have been extracted.

Since the proliferation of poli- and metamorphic malware, dynamic malware analysis has also become an effective approach to understand and categorize malware. Dynamic analysis relies on observing the execution of the malware sample in a quarantined and monitored environment to obtain behavioral information and to extract the unpacked version of the sample [36, 127, 132]. The interaction between the executing malware sample and the host OS allows dynamic malware analysis systems to collect behavioral characteristics that aid in formulating defensive steps by identifying attack vectors and vulnerabilities the malware uses. Consequently, dynamic malware analysis relies on the *breadth* and *fidelity* of the collected data. Dynamic malware analysis can also greatly benefit from virtual environments as it enables real-time monitoring of the execution, disk and memory of the sandbox, providing a direct way to observe infections as they occur and their effects on the compromised systems. Furthermore, virtualization allows such observations to be made from a safe vantage point, providing complete view into the system.

Significant challenges remain to be solved however. The increasing number of mal-

ware samples poses a particular challenge as it greatly inflates the cost of hardware required for effective analysis. In conjunction with anti-debugging and obfuscation techniques, the time it takes to formulate effective solutions is greatly exacerbated [13, 48]. There is a clear need for effective *scalability* in automated malware collection and analysis.

As honeypots and dynamic malware analysis systems have become widely deployed, malware has evolved to detect and evade such systems by either refusing to connect to [61], or to execute in a sandboxed environment [78]. It has also been observed that malware modifies its run-time behavior to lead the analysis system astray [7]. Consequently, it is critical for honeypots and dynamic malware analysis systems to provide a *stealthy* and *isolated* environment to hide the presence of the data collection from the external attacker and protect the data collection system from the executing sample.

Furthermore, the *fidelity* of the collected data is essential for effective dynamic analysis. As rootkits now employ a variety of techniques to hide their presence on a system, the broader the scope of data collection, the more likely the analysis will reveal useful features. While emulation and binary instrumentation based approaches have been shown to provide the greatest depth of information [33], the hardware emulators they rely on incur a prohibitive performance overhead. Furthermore, hardware emulators inherently suffer from incomplete and inaccurate system emulation, which could lead malware to easily detect such environments to thwart analysis.

Hardware virtualization has also been proposed and studied over the last decade as an underlying platform to develop malware collection and analysis tools. In this thesis we present a comprehensive evaluation of the technique's application to malware collection and analysis. As part of this evalation, we formalize the core objectives such systems need to fulfill and present extensive experiments performed to aid us in determining whether hardware virtualization is a viable method to construct such systems.

1.1 Problem statement

This dissertation details the evaluation of hardware virtualization as an underlying platform for automated malware collection and analysis. The evaluation is realized via the combination of four distinct objectives such systems need to fulfill: scalability, stealth, fidelity and isolation. Each of these objectives are reviewed in turn:

(O1) Scalability The ever increasing number of in-the-wild malware requires automated capture and analysis without incurring prohibitive hardware requirements. Unlike emulation based systems, hardware virtualization requires the assignment of real hardware resources, such as disk and RAM, for each virtual machine. Scalability in our context thus means maximizing the number of concurrently active malware collection and analysis sessions as to avoid the linear growth in hardware requirements as the number of sessions is increased.

(O2) Stealth As modern malware effectively deploys anti-debugging and anti-forensic techniques to hide from automated systems, effective stealth is required. Malware should not be able to detect the monitoring environment regardless if it executes within the sandbox, or if it is interacting with the sandbox remotely over a network. Thus stealth in our context means the capability to execute malware samples without the presence of the analysis engine being revealed to the malware.

(O3) Fidelity Formulating effective defense mechanisms requires the in-depth understanding of both the infection process and the run-time behavior of malware. Automated data-collection and artifact extraction thus has to accurately describe and capture the process without the potential of acquiring tainted evidence. As modern malware has burrowed deep into operating systems in the form kernel-mode rootkits, the automated process has to be aware that the guest internal information may have been tampered with. Furthermore, modern malware has increasingly

became resident in memory-only, leaving limited to no traces on the hard-drive of compromised machines. Thus, data-collection and artifact extraction has to take into account these malware tendencies to provide the required fidelity in the collected data.

(O4) Isolation The monitoring component has to be securely isolated from the executing sandbox environment as to minimize the potential of data contamination or the chance of a breakout attack. The monitoring component should also be isolated from critical system components as to minimize the exposure of the trusted computing base in case such a breakout occurs. Furthermore, malware collection and analysis sessions should be isolated from each other as to prevent the cross-contamination of data during concurrent sessions.

1.2 Scope

In this thesis we extensively deal with the study of malware from many perspectives, thus it is important to highlight and properly limit the scope of the investigation. In summary, this thesis limits its scope to the study of *malware collection* and *malware analysis* from the perspective of *hardware virtualization*. Many topics which have been discussed in prior art in relation to these topics are considered out-of-scope for this thesis. We make our best effort to address common questions that arise during the study of malware, and in some cases we provide our own case-by-case analysis, but these discussion are provided solely to aid the reader in obtaining a better perspective on the topic in general. Furthermore, the prototypes herein presented only show limited aspects of how one may develop such tools using virtualization, which we use to determine whether the core objectives can be met with the use of virtualization.

Foremost, a question which we will *not* try to answer is: by what metric do we decide what software is considered malicious (aka. malware)? While the body of work on the topic is significant and has been approached from many angles, in our opinion the best

answer largely remains: *it depends*. It depends for example on who we ask, when we ask or how we ask. Consider for example how a screen-capture software may be deemed benign by the end-user, but be considered malicious by companies who want to protect the video stream transmitted to the users' screen for a one-time view. The very same screen-capture software could be considered malicious by the end-user as well if it is operated by a third-party without their permission. Furthermore, should we consider this software in its entirety malicious, even when it does not actively perform its screen capture function, just has the potential to do so? The answer to these question remains highly subjective. In this thesis when we discuss specific pieces of malware, those have been labeled as such by a third-party, such as anti-virus companies. Whether their classification is justified remains out-of-scope for this thesis.

Another important topic we consider out-of-scope for this thesis is the *reproducibly of experiments*. While reproducibility of experiments is a cornerstone of the scientific method, it is the unfortunate case that many security research results remain irreproducible. The reason for this is partially caused by researcher never releasing their source code. However, there is another fundamental reason why - even when the tools are released - we are unable to repeat the experiments exactly: many of the vectors determining the results of the experiments are beyond our control. Experiments performed with live malware samples that communicate with external systems are notoriously problematic. In fact, an entire body of research has been dedicated to the study of malware that aims to record as much of the environment as possible, so that the malware analysis sessions can be replayed. In our opinion, such systems can still only capture a narrow scope of data and thus provide incomplete replay capability. For example, it is impossible to record data for which we have no sensors for. The most famous recent incident for such a case would be the RowHammer hardware bug presented by Google. Nevertheless, amicable attempts are being made to develop such tools regardless of their limitations. During the research leading up to this thesis we diligently worked on open-sourcing all tools and

techniques so that fellow researchers may use them to perform their own experiments. However, exact replication of live malware experiments remains out-of-scope for this thesis.

In this thesis we also consider the detection of virtualized environments distinct from the detection of malware collection and analysis systems. As hardware virtualization is already ubiquitous in computing environments, modern malware requires separate mechanisms to distinguish between monitored and unmonitored virtual environments. It is nevertheless the case today that many malware solely rely on the detection of virtualized environments to determine whether to continue operation or not. In this thesis we consider countering such malware to be out-of-scope as their prevalence and effectiveness will foreseeably continue to decline.

Similarly, deciding what is the optimal time-frame for collection and analysis, and whether we have collected enough data is also considered to be out-of-scope. Given that deciding whether and when a given software will exhibit a specific type of behavior is reducible to the *halting problem*, we do not attempt to determine an optimal time or optimal volume of data to be collected. Rather, we observe what generic time-out periods have been used in prior research and use similar values in our experiments. Furthermore, for our malware collection system, we additionally limit the time and scope of data collection in case we observ malware propagation attempts made to systems outside our test environment as a security precaution, which can also be considered standard practice.

Finally, we present only limited prototypes of hardware virtualization based malware collection and analysis systems. As our goal is evaluate the use of hardware virtualization itself, we consider it out-of-scope to develop comprehensive tools for the purpose of this dissertation. Our goal is simply to determine whether it is possible to develop such tools and to determine what constrains may apply and what limitations need to be kept in mind.

1.3 Methodology and Limitations

In our systematic evaluation we limit our scientific endeavor to the study of the core hypothesis: *hardware virtualization is an effective technique to perform both malware collection and malware analysis such that our Objectives are met*. The methodology by which we performed the evaluation differs depending on the Objective. For example, while *Scalability* can be largely evaluated empirically, *Stealth* and *Fidelity* remains an inexact study. The reason for this is rooted in the evolving nature of malware. Without a universal model of what malware is and what behavior is malicious, the best we can do is study the current state-of-the-art for what is commonly considered to be malware and present a case-by-case evaluation whether are Objectives are met based on that study. As a consequence, our methodology in such cases will inadvertantly only provide results that are indicative, rather than definitive.

- Our methodology to measure Scalability has been two fold. First, for malware collection systems we perform measurements by calculating the maximum number of *concurrent network sessions* that can be maintained with limited hardware resources, while matching all Objectives. Second, for malware analysis we measure the *throughput* of the system by calculating the maximum number of malware analysis sessions that can be performed in parallel on limited hardware resources, while matching all Objectives. Based on these measurements we extrapolate the expected scalability of the system.

- Our methodology to evaluate Stealth has been based on the study of modern malware behaviors found in literature. We researched state-of-the-art malware techniques used to detect analysis and capture environments and actively countered these techniques, which have been shown to be effective both in prior art and in our own experiments. Nevertheless, our method offers only limited guarantees that the Objective will be met once malware evolves new techniques.

- Our methodology to measure Fidelity has also been based on the study of modern malware behaviors found in literature. We evaluated techniques employed by modern malware that allow them to hide in previously unmonitored corners of the systems and developed methods which allow us to retain visibility into the behavior of such evasive malware as well. The reasoning behind our methodology has been based on the assumption that the more verbose and inclusive the data collection is, it would be reasonable to assert that the chances of observing behavior that is useful for the understanding said malware will be higher. Thus, our methodology focuses on evaluating the scope and method of data collection to show how it maximizes the visibility. We do not attempt to calculate or evaluate how useful the collected data is, as in our opinion that is a highly subjective metric unfit for scientific evaluation. However, we present numerous cases to highlight how we have been able to observe behavior that thus far evaded other analysis systems, and even human analysts.

- Our methodology to evaluate Isolation has been based on the study of the attack surface of modern hypervisors systems, presented in prior art and based on our own research. Based on this study, we created a heavily disaggregated and hardened environment to minimize the exposure of the known attack surface and a design is presented that highlights how the architecture maximizes the adherence to the Objective. Our method to evaluate this Isolation is based on the analysis of the attack surface. This analysis is provided with the understanding that complete isolation is impossible to achieve on modern hardware and thus the Objective is expected to be only partially met as future malware is likely to discover new methods to circumvent it.

1.4 Publications

Most of the material presented in this dissertation has appeared previously in the following publications, in chronological order:

- **T.K. Lengyel**, J. Neumann, S. Maresca, B.D. Payne, A. Kiayias; (August 2012). "Virtual Machine Introspection in a Hybrid Honeypot Architecture." *Proceedings of the 5th Workshop on Cyber Security Experimentation and Test*, Bellevue WA, USA. Acceptance rate: 12/25 = 48%.

- **T.K. Lengyel**, J. Neumann, S. Maresca, A. Kiayias; (June 2013). "Towards Hybrid Honeynets via Virtual Machine Introspection and Cloning." *Proceedings of the 7th International Conference on Network and System Security*, Madrid, Spain. Acceptance rate: 41/169 = 24%.

- **T.K. Lengyel**, T. Kittel, J. Pfoh, C. Eckert; (September 2014). "Multi-tiered Security Architecture for ARM via the Virtualization and Security Extensions." *Proceedings of the 1st Workshop on Security in highly connected IT systems*, Munich, Germany.

- **T.K. Lengyel**, T. Kittel, G. Webster, J. Torrey, C. Eckert; (December 2014). "Pitfalls of virtual machine introspection on modern hardware." *Proceedings of the 1st Workshop on Malware Memory Forensics*, New Orleans LA, USA.

- T. Kittel, S. Vogl, **T.K. Lengyel**, J. Pfoh, C. Eckert; (December 2014). "Code Validation for Modern OS Kernels." *Proceedings of the 1st Workshop on Malware Memory Forensics*, New Orleans LA, USA.

- **T.K. Lengyel**, S. Maresca, B.D. Payne, G. Webster, S. Vogl, A. Kiayias; (December 2014). "Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System." *Proceedings of the 30th Annual Computer Security Applications Conference*, New Orleans LA, USA. Acceptance rate: 47/236 = 19.9%.

- **T.K. Lengyel**, T. Kittel, C. Eckert; (September 2015). ”Virtual Machine Introspection with Xen on ARM” *Proceedings of the 2nd Workshop on Security in highly connected IT systems*, Vienna, Austria.

- A. Fischer, T. Kittel, B. Kolosnjaji, **T.K. Lengyel**, W. Mandarawi, H. de Meer, T. Mller, M. Protsenko, H.P. Reiser, B. Taubmann, E. Weishup; (October 2015). ”CloudIDEA: A Malware Defense Architecture for Cloud Data Centers.” *Proceedings of the 5th International Symposium on Cloud Computing, Trusted Computing and Secure Virtual Infrastructures*, Rhodes, Greece.

Specifically, the first two papers are about our malware collection system and are the base of Chapter 3. The fourth paper forms the basis for Chapter 5 and the other papers are the base for Chapter 4 where we discuss dynamic malware analysis.

1.5 Outline

The rest of this document is organized as follows. Chapter 2 reviews the literature of hardware virtualization based malware collection and analysis. Chapter 3 discusses the design and implementation of our malware collection system and presents the evaluation of our core objectives via extensive experiments. We describe in Chapter 4 the design and implementation of our dynamic malware analysis system, and with the aid of extensive tests and experiments we evaluate our objectives. In Chapter 5 we take a look at various hardware and software limitations that may impact similar hardware virtualization based security systems, including the prototypes presented in this thesis. Finally, in Chapter 6 we provide a summary overview of our work, highlight the contributions made, discuss future directions and present our concluding thoughts.

2 Related work

In the following we provide an overview of select publications that formed the base of our inquiry. This chapter is organized into three subsections: first, we review prior work in the development of malware collection systems; second, we turn our attention of malware analysis; third, we discuss systems which proposed the use of hardware virtualization for the development of security applications in general.

2.1 Malware collection

Honeypots over the last decade have seen a significant role in the battle against malware. Honeypots have been developed with the purpose of capturing malware binaries and have traditionally been categorized based on the underlying mechanisms they are implemented with: emulation of vulnerable services or full-system exposure. The former has been commonly referred to as Low-Interaction Honeypots (LIHs); the latter as HIHs.

One of the most commonly used open-source low-interaction honeypot is Dionaea. Dionaea can be configured with a set of emulated services to mimic the behavior of known vulnerabilities to capture malware binaries. However, in order to successfully capture binaries, the interaction provided by the emulated services has to match with real-world systems, which is a challenging task when we consider unknown (0-day) bugs malware may seek to exploit.

Recent Honeypot developments have more closely focused on developing honey-clients, such as PhoneyC [83]. Unlike regular honeypots - which passively wait for an attacker to connect - honey-clients pro-actively connect to potentially malicious services. The methods by which these systems have been developed closely resemble LIHs: emulating vulnerable client-side programs, such as browsers, to trigger the malicious behavior and capture of the exploit from the server.

High-interaction honeypots however have not seen wide-scale deployment, arguably

because of the high cost of maintenance. Furthermore, each high-interaction honeypot has the potential to be turned into an attack vector itself, thus it can be potentially hazardous to host such systems. HIH tools developed have also been shown to be vulnerable to tampering. For example Sebek [54], a kernel module designed to provide stealthy monitoring of a Windows operating system has been shown to be detectable shortly after release [34]. Later efforts, such as Qebek [53] and VMscope [58] moved the monitoring component into the emulation layer provided by systems such as QEMU.

A core limitation with HIHs is related to the need of assigning considerable hardware resources to each HIH session. To address this problem, Honeybrid [11] was developed as a transparent network proxy designed to reduce the load on HIHs by utilizing LIHs as filters on the incoming attack traffic. By providing fine-grained control of the network flows, the active connections can be switched transparently between the available honeypots. Furthermore, the system was designed to allow custom extensions to easily integrate into the decision engine of the system.

To address the excessive resource allocation problem when using hardware virtualization based HIHs has been prominently addressed by Vrable *et al.* [120] who implemented a highly scalable honeynet system named Potemkin. Potemkin focused on improving the *scalability* of HIHs by observing that the high number of network connections to the honeypot system have quickly exhausted the available hardware resources. Potemkin solved the scaling issue associated with running a large number of nearly identical Virtual Machines (VMs) by introducing memory sharing that de-duplicates identical memory pages present across VMs. The technique was built upon solutions introduced to the live VM migration problem: optimizing the transfer time of a live VM's memory from one physical host to another. The insight into the problem was that a VM that had to be transferred did not have to be paused for the entire migration time, as the majority of the pages remain unchanged during the transfer period. By tracking the pages that do change during the first phase of the live migration period, the downtime of the VM can be limited to the

period of transferring the modified pages only. The same technique that allows tracking of page modifications opened the door for memory de-duplication using a CoW approach implemented by Potemkin. However, Potemkin lacked monitoring and artifact extraction mechanisms which had to be performed manually.

Later systems, such as the one developed by Asrigo *et al.* [4] attempted to use the hardware virtualization for in-depth monitoring of honeypots. However, due to the lack of proper hardware support at the time, components of the monitor still had to be placed within the guest. These in-guest components thus shared a similar attack surface as, for example, the Sebek honeypot did. Despite these short-coming, the hybrid model of in-band delivery of interesting events to external monitoring applications has also been the base for subsequent systems, which we further discuss in Section 2.3.1.

2.2 Malware analysis

Today, anti-virus systems heavily make use of *binary signatures*. Traditionally, malware binaries obtained from honeypots and other sources are first fingerprinted using hashing and static analysis methods [36] as to avoid repeating the analysis of already known binaries. However, as a counter-measure, malware has started to restructure the binaries between infections as to avoid being easily identified, known as the *packer problem* [48]. Consequently, manual reverse engineering efforts can no longer keep up with the influx of binaries. As a solution, dynamic malware analysis was proposed to automatically execute and extract the relevant information and artifacts from the executing binary. The information obtained can thus aid in identifying malware to speed up the development of protective solutions. The observed behavior of the malware sample can itself be turned into a *behavioral signature* [90] to be used by end-point security solutions. Furthermore, the behavioral information can also reveal the infrastructure the malware relies on (command and control channels, domain names, etc.) which can then be taken offline by authorities [84].

CWSandbox [127] was one the of first dynamic malware analysis systems to utilize a sandbox environment for monitoring the interaction between the OS and the malware. CWSandbox operates by loading a kernel driver into Windows that hooks all exported APIs to intercept the system calls performed by user-space programs. Subsequent systems, such as Anubis [8], were developed similarly. However, as has been shown for the Sebek [34, 54] honeypot, in-guest kernel modules and user-space agents are vulnerable to detection and tampering.

Cuckoo [14] is currently one of the most popular open-source dynamic malware analysis systems which uses the same approach as CWSandbox. Cuckoo supports a wide-range of virtual machine monitors aka. hypervisors (VMMs); however, it doesn't take advantage of the VMM beyond isolation, as it relies on an in-guest agent to perform monitoring. As no special protection is provided to the in-guest agent from the hypervisor, stealth and tamper resistance cannot be guaranteed, thus potentially leading to incorrect or incomplete analysis results.

Ether [24] made use of hardware virtualization extensions to monitor the execution of malware within a virtual machine. Built on a modified version of Xen, Ether made use of page-faults configured to trigger VMEXITs on specific code locations. Ether used this to effectively trace system calls in the observed VM. At the time Ether was built, this feature was used to trap all page-faults to the hypervisor where the address translation could be performed in the shadow page tables. While Ether has made significant efforts to hide these modifications, its effective stealth in practice has been called into question by Pek *et al.* [89] who pointed out implementation issues that may still reveal the presence of Ether to the guest OS.

A common problem facing the above mentioned approaches is the limitation of using only system-call monitoring. While the technique has been shown to be applicable for intrusion detection [51], the approach loses its effectiveness when we consider how modern malware increasingly resides directly in the kernel [52]. As kernel-mode malware (aka.

rootkits) only use system-calls in a limited form or not at all, these monitoring techniques will be unable to accurately trace the execution of the malware after the initial infestation. While alternative monitoring techniques have been proposed, such as tracking the heap allocations of the guest operating system [95], these methods have not been implemented for hardware virtualization in prior art.

Alternative monitoring techniques not based on emulation or hardware virtualization have also been proposed, such as BareCloud [62]. Similarly to Ether, BareCloud proposes to avoid the usage of in-guest agents to perform monitoring. Unlike Ether however, BareCloud attempts to do the monitoring of the malware via hard-drive activity. While BareCloud does avoid issues with malware that refuses to execute on virtualization hardware, it relies on a networked hard-drive, which is detectable to the malware executing in the system. As such a setup is highly unusual, it is reasonable to argue that it is an easy sign of the monitoring environment malware could look for. Furthermore, with the increasingly memory-only nature of modern malware [45], capturing only the hard-drive activity severely limits the scope of data-collection.

Another alternative technique has been proposed by Zhang *et al.*, named SPECTRE [134]. Similar to virtualization, SPECTRE proposes to move the monitoring component outside the guest system. Instead of utilizing virtualization, SPECTRE proposes to use the System Management Mode (SMM) to perform the monitoring from. While effective, the SMM normally restricted to OEMs only, thus gaining access to this operation mode in practice is problematic. We will further discuss the SMM in Section 5.4.

2.3 Monitoring via Hardware Virtualization

As we can see, both malware collection and malware analysis systems have experimented with the use hardware virtualization as an underlying platform [24,120]. In parallel to these systems, virtualization has been identified as a method to be used for various other types of security systems as well [20]. As such, a significant body of work deals

with the problems inherent in using virtualization for security applications [42].

First, the problem with in-guest agents is their inherent exposure to the same execution context they are to observe [34]. While the virtualization layer still provides some level of isolation of the VM, the monitoring component is inherently at risk of tampering. External monitoring approaches however face the *semantic gap problem* [43]. The problem can be summarized as follows: while in-guest agents have direct access to the system Application Binary Interface (ABI) in the form of system calls, external monitoring agents need to reconstruct high-level state information of the guest OS by observing low-level hardware information. For example, on modern systems the hypervisor interacts minimally with the memory of the guest OS. For most VMMs, a VM is a region of physical memory containing opaque data, rather than memory with virtual addresses or other high-level data structures. It becomes the job of the external application to either reconstruct the state of the high-level software system itself or leverage the guest OS to gain a high-level, high-fidelity view. This is referred to as Virtual Machine Introspection (VMI).

In the following we examine and discuss related work that approach the core problem from different angles: hardening in-guest agents vs. pure external monitoring.

2.3.1 In-band delivery

As in-guest agents have access to the native ABI, many standard information gathering steps can be performed by utilizing the guest OS itself. However, this method is vulnerable to tampering: the guest OS may have been modified to provide false information. Nevertheless, the VMM could be leveraged to provide tamper resistance to the in-guest monitor and/or critical kernel components. In the following we will discuss notable systems exploring such approaches.

To isolate the security application within the guest system, SIM [104] proposes to create an additional address space. As the hypervisor controls the physical memory, the new address space can be setup such that only the security application can access the guest's

entire memory. With this the hypervisor can enforce access to the security application's address space to be restricted to the security application itself. Enforcing such address space isolation is achieved by ensuring that the memory regions of the security application aren't mapped into the address space of the guest. An attempt therefore to access the security application will lead to a page fault as the hardware will be unable to resolve the virtual addresses belonging to the security application. To avoid virtual address conflicts, SIM marks the virtual address space that the security application occupies as used within the guest's address space. For the security application to be able to access the guest's memory, the guest's memory regions is also mapped into the address space of the security application.

Process implanting [47] proposed hijacking a process running within the guest from the hypervisor. To provide stealth, the hypervisor does not create a new process within the guest, but rather substitutes the image of an existing process with the image of the program that is injected, a technique also known as *process hollowing* used by rootkits [126]. As a consequence, whenever the victim process is scheduled, the guest system will actually execute the injected program instead of the original process. The injected process can then access guest information using system calls and transfer the obtained information to the hypervisor with the help of hypercalls.

The security of the approach is based on the assumption that the guest OS system is trusted. To protect the implanted process against other malicious processes, its rights are elevated to root and it is protected from kill commands. In addition, the hypervisor creates a new physical memory region for the injected process at run-time. Since other process' are unaware of this memory region and due to the fact that the implanted process is injected into a randomly selected guest process, malicious process' on the system neither know which physical memory range the implanted process uses nor which process was substituted. While this does not directly protect the injected process, it makes it more difficult for an attacker to detect the implanted process.

The most significant drawback of process implanting is the restriction that the approach is only secure if the guest OS has not been compromised. One of the main security reasons to make use of virtualization is to be able to protect security applications in spite of the fact that the OS kernel is compromised. In fact, if the OS is trusted, there is - from a security standpoint - no reason to resort to virtualization in the first place. Thus the technique, is not particularly well-suited for the implementation of hypervisor-based security applications.

X-TIER [117] tries to solve the security problems of process implanting by providing hypervisor-based security applications the possibility to inject device drivers into a running VM. In contrast to an implanted process, the injected drivers are executed within the guest's kernel space. To protect the injected drivers during their execution, X-TIER isolates the drivers within the guest system by employing two separate techniques. First, to isolate a driver during its normal execution, X-TIER disables all interrupts within the VM. As a result, the driver is executed atomically and cannot be accessed by other code contained within the guest system. While this approach can protect the injected driver during normal execution, the isolation will be broken if the driver invokes an external function. Thus, as the second technique, to circumvent this restriction, X-TIER intercepts all external function calls that are conducted by the injected driver and temporarily removes the driver from the guest system. Once invoking the external function on behalf of the driver finishes, the driver is reinjected. Using this mechanism the injected driver is able to call arbitrary kernel functions and use their results, while itself cannot be accessed by the invoked functions. Since X-TIER provides its own loader code for kernel drivers, it is capable of injecting any driver that was compiled against the target kernel, even if the driver loading functions are hooked within the guest.

Due to its isolation mechanisms, X-TIER provides strong security guarantees. Compared to process implanting the added security, however, also reduces the performance of the mechanism. This is especially true if the injected driver makes use of many ex-

ternal function calls, since each external function call leads to at least two VM exists. In addition, while X-TIER is able to isolate the injected component during external function calls, it does not check the integrity of external functions before their execution. Thus, the data gathered with X-TIER may not be trustworthy.

SYRINGE [19] tries to solve this problem by verifying the integrity of in-guest functions before and during their execution. In particular, SYRINGE enables the hypervisor-based security application to inject function calls into a guest system. Before an in-guest function is invoked from the hypervisor, SYRINGE verifies the integrity of the function by comparing the hash of the code page containing the function's entry point against a white-list. During execution, SYRINGE will repeat this process whenever the function leaves the current page and starts executing from a different page. In addition, the system ensures run-time control-flow integrity by verifying the targets of all `call`, `ret`, and indirect branch instructions according to a control-flow policy. To provide protection against run-time modifications of stack and heap data, SYRINGE executes the injected function call atomically (without interrupts) within the guest.

Since SYRINGE only injects function calls into a guest system, security applications are isolated by the hypervisor and must not directly be protected by SYRINGE. Stealth is thereby achieved by executing the injected function atomically within the guest system. However, while SYRINGE ensures the integrity of the executed code, it cannot ensure the integrity of the data that is used by the injected functions. Therefore an attacker can still provide false information to the monitoring application, by performing for example DKOM [17] attacks, which alter the state reported by the guest kernel without affecting system stability. We further discuss the details of DKOM in Section 4.5.1.

2.3.2 Out-of-band delivery

In the following we will discuss systems which leverage purely out-of-band data collection methods. As this method has formed the bases of our own prototypes, we provide an

extensive discussion on the topic to aid the reader in better understanding the design and development challanges of such systems. Afterwards, we highlight the most significant prior work in this direction.

2.3.2.1 Discussion

The out-of-band delivery approach uses semantic information that is obtained before the data collection takes place, referred to as *introspection*. The type of semantic information that is required to be delivered out-of-band requires at least information about the virtual hardware of the guest. Constructing such hardware semantic information is relatively simple because of the requirement of compatibility exposed upon the virtual hardware. Nevertheless, differences between the physical and virtual hardware do exist, having a direct impact upon *scalability* and *stealth* that have to be taken into consideration [41].

As with in-band delivery, the use of untrusted guest data-structures have been shown to be problematic due to various attacks, such as DKOM and Direct Kernel Structure Manipulation (DKSM) [6]. The core problem with both of these attacks is that the VMI tool relies on in-guest information to bridge the semantic gap, without that data being enforced by the hardware. An in-guest attacker is free to manipulate the use and layout of kernel structures without the risk of causing the guest OS to crash. This problem has recently been named the *strong semantic gap problem* [57]: given the potential of poisoned in-guest data, accurately reconstruct high-level state information from an external perspective.

While hardware semantic information is important in almost any VMI approach, there are some systems that try to *exclusively* make use of it. These have the advantage that the architecture of the hardware will not change at run-time, leading to stronger tamper-resistance. When relying on software semantics, this software can change at run-time, even for malicious means. For example, a VMI component may be inspecting the system call table, while a malicious entity creates a second system call table with malicious hooks

and patches the code to make use of this malicious table. If the VMI component only inspects the system call table at the position known from software semantic information, it will not catch such a malicious change. While this is a simple example that can easily be remedied, it serves to illustrate that the software architecture can easily change at runtime, while the hardware architecture cannot. If a VMI component inspects the interrupt descriptor table register (IDTR) to determine where the interrupt descriptor table (IDT) is located, it can be sure that the IDT is in that location as there is no way for a malicious entity to change this behavior without crashing the system. This fact leads to increased tamper resistance for hardware-based solution.

Additionally, such approaches benefit from being guest OS agnostic. That is, if a component relies completely on the hardware semantic information, it can be sure that its information is relevant regardless of the guest OS that is running in the VM. Any software running in the guest must adhere to the hardware architecture, therefore the information that is retrieved is relevant to all guest software that runs on the hardware.

On the other hand, approaches that exclusively make use of hardware semantic information are limited in the scope of what they can inspect and understand. There are simply some components within the guest OS that are independent of the hardware semantics. For example, the process ID of a process is difficult to retrieve without software semantic knowledge as a process ID is a construct of the OS and is completely independent of the hardware.

Thus, when building out-of-band delivery VMI applications to inspect software systems, the semantic gap is a significant barrier. Out-of-band VMI systems require in-depth knowledge of the inspected software system's data structures and algorithms in order to reliably reconstruct their state. The problem is further exacerbated by the fact that the software system may be closed-source, thus require significant reverse-engineering effort to properly inspect. Even when such reverse engineering effort has been performed, keeping the semantic information up-to-date and valid can be challenging.

2.3.2.2 Prominent prior work

ReVirt was one of the earliest VMM based security applications which has implemented such an approach [35]. ReVirt's primarily focus was on capturing hardware events to be able to replay them later to reconstruct the state of the VM. ReVirt however had no understanding of the *context* of these hardware events and the task was left to a human analyst to inspect the state of the VM.

AntFarm [60] attempted to deduce contextual knowledge about the operating system by fusing hardware events with hardware semantic information to track the execution of running processes. This context can be deduced by the common use of the CR3 register on the x86 architecture to implement protected memory mode: the CR3 register is used by the operating system to hold the physical address to a process' directory table base (DTB) used for translating virtual memory addresses to physical addresses. As such, the CR3 value uniquely identifies a process on the system. AntFarm has thus been successful in tracking running processes; however, extracting higher level software semantic information was left unresolved.

Srivastava *et al.* implemented a VMI-based firewall called VMwall [109]. VMwall captured network flows and, using the XenAccess library, correlated them with processes running within a VM. VMwall accomplishes the correlation by extracting information from data-structures of the guest's Linux kernel. Dolan-Gavitt *et al.* [27] later noted that the same functionality can be achieved by utilizing forensics tools (Volatility), to inspect the guest kernel data-structures in conjunction with live introspection tools (XenAccess, LibVMI). As both approaches rely on VM kernel data-structures, they are vulnerable to the same kernel subversion attacks we discussed earlier.

Nitro [91] is another system that uses virtualization extensions to monitor system calls within the VM. Nitro, similarly to Ether which we described earlier, also manipulates the machine specific registers: by changing the SYSENTER_EIP_MSR and MSR_SYSENTER_CS used during SYSENTER to cause an invalid value of 0 to be loaded into CS register. This

in effect causes a general protection fault leading to a VMEXIT. Similar techniques were implemented for both interrupt-based and for SYSCALL based system-calls. The benefit of Nitro's approach over Ether is that the faults it triggers are significantly fewer then Ether's page faults, leading to better performance.

InSight bridges the semantic gap by deriving information about the kernel's data structures from the source code and its debugging symbols. By parsing the source code, InSight can identify how different data types are used. For example, when `void*` is used in the debug symbols to describe a pointer, the true type of the pointer can be deduced from how it is used in the source code, which may even be variable through casts and the like. The framework uses this information to resolve symbols and build a graph of data types to aid in understanding kernel memory. The graph is generated by using the public symbols delivered with the binary code as the graph root. As there are still data types where such "used-as" approach is not enough, InSight also provides a rule-engine to record expert knowledge manually to enable the system to walk data structures like linked lists and trees. For example, the linked-list data structure is usually embedded within other structures; however, the `prev` and `next` pointers do not point to beginning of the structure they are embedded into and instead reference the next structure's `list->next` pointer. With the use of the rule-engine, the framework can automatically calculate the proper offsets to enable the user to directly navigate through the kernels graph of data structures.

Automatic construction of expert knowledge for systems without source-code is of entirely different nature. In such cases, knowledge have to be derived from additional sources, like debug data, documentation and reverse engineering. For example for Windows, access is only available to the documentation and debug data provided by Microsoft. When we look at the data structures exported by the debug data, many of the structures lack public documentation as Microsoft considers them opaque. Even more perplexing are the structures that are opaque and take on different meaning based on context (such as unions) or are flat out obfuscated (like pointing to structures with void point-

ers and having non-descriptive names) to protect against reverse engineering [12, 32]. While such data structures and types are present in the Linux kernel as well, their use can be derived from the source-code - a method unavailable for Windows.

Volatility [119], an open-source forensics memory analysis (FMA) framework is a collection of such state-reconstruction algorithms obtained primarily by reverse engineering. While designed for manual investigations of physical memory dumps, in combination with LibVMI [76], a hypervisor agnostic abstraction library to access VM memory, it can be used to inspect the memory contents of live VMs as well [27]. Among functions to enumerate a wide aspects of Windows and Linux internals, Volatility also showcases a set of routines directly aimed at the detection of API hooking and DKOM attacks. Its Application Program Interface (API) hook detection capability includes detecting hooks in the Import Address table (IAT) and Export Address table (EAT) of processes by verifying that the symbol contained in the tables point to memory locations of valid modules. Inline hooks (aka. trampoline hooks) are detected by disassembling the symbols and checking for `PUSH/RET` instruction sequences. It is worth noting, however, that inline hooks may well be more complex and evade this detection scheme.

The detection of DKOM attacks and anti-forensics tricks [6] [50] still pose a significant challenge however. Even if the algorithms used to gather the information are re-implemented on the VMM-layer, the data they operate on may still be unreliable. For example both Linux and Windows keep the list of running processes maintained in a circular linked-list. The head of the list is exported through a kernel symbol and process enumeration is performed by simply looping through the list [102]. As this data structure is a non-critical element of the kernel, not used for example in the scheduling of the processes, it can be safely modified. If an entry is removed from the list, it will continue to run, but will be invisible to tools that rely only on the linked-list [39].

Volatility's detection of such hidden processes and threads uses the *cross-view validation* method. Cross-view validation can - in certain scenarios - enable the detection

of DKOM techniques [18, 124]. The technique requires multiple views of the same information to detect discrepancies, such as a process being present in one list but not in another. Cross-view validation may still prove unreliable when there are no discrepancies to detect [102].

As an attempt to side-step this issue, signature based scanners were proposed to directly locate data structures in memory, without the need to traverse potentially compromised links. An example of such an approach is Volatility's pooltag scanner implementation for Windows, which looks for strings (like "Proc" or "KDBG") in the pooltag header, a header that is automatically attached to structures when when they are allocated. However, the reliability of a signature directly depends on the signature being a critical component of the structure itself. Pooltag signatures for example are non-critical and therefore can be modified without affecting the functioning of the process using the structure they describe. Finding signatures that are both critical components of the structure and also uniquely identify the structure are hard to come by and remain an open issue [31] [73].

CXPInspector [128] builds exclusively on the use of EPT for execution tracing. As EPT adds an additional layer of paging where translation between guest physical and machine physical address takes place, any access violation within this layer will trigger a VMEXIT, thus remaining transparent to the guest. By marking certain pages non-executable, CXPinspector is able to monitor the execution of the VM. However, the use of EPT for execution tracing has been known to still add significant overhead to the execution of the VM, mainly as a result of the granularity EPT can be set to trigger violations on.

SPIDER [23] evaluated the use of injecting debugging breakpoints to enable stealthy debugging of processes within virtual machines using the KVM hypervisor. However, starting a malware sample with SPIDER still requires either manual action or an in-guest agent, which either prohibits scalability or hinders the stealth capability of the system. Similarly, determining where to place breakpoints with SPIDER is a manual process, as would normally be done during debugging.

3 Malware collection

Automated malware collection systems, aka. honeypots, have been a central tool over the last decade to provide security analyst with malware binaries, as discussed in Section 2.1. Honeypots have been traditionally divided into two categories: low-interaction, emulated services (LIH); and high-interaction, full systems (HIH). For the evaluation of hardware virtualization, our target honeypot type is the HIH.

The overarching goal of any type honeypot is the *capture of binaries*, which can be considered the *primary requirement*. Thus, in this section our goal is to evaluate how hardware virtualization can aid in constructing a system that achievies the *primary requirement* while satisfying our *four core objectives*.

In this section we first list the challenges facing in achieving each of our core objectives. We follow by presenting the design and implementation of our first prototype that achieves the primary requirement while meeting the core objectives. We follow by performing extensive experiments to highlight how the system operates under realistic conditions. Based on these experiments we identify where our system can be improved to better satisfy the core requirements and the challenges that one faces when constructing such improved systems. We provide the design and implementation of our improved prototype that solves these challenges, and we perform additional experiments under realistic conditions. We conclude with our summary evaluation on how hardware virtualization can aid in constructing such systems while meeting the core objectives.

3.1 Challenges

There are a number of challenges one has to consider when constructing virtualization based honeypots. Foremost, the *four core requirements* have to be met, which has thus far not been shown in practice. While in our discussion of prior work in Section 2.1 we highlighted systems that achieved some of the core objectives, thus far no system has

been developed that satisfies all four. The reasons for this are summarized in the following.

(O1) Scalability: While running multiple virtual machines on the same hardware directly offers improved scalability as compared to bare metal systems, there still exists a set of hardware requirement that needs to be met for each virtual machine. The main challenge in this regard are *disk space* and *memory*, for which the requirements grow linearly as we increase the number of virtual machines. As the primary goal of the system is to obtain malware binaries, the system has to take into consideration the situation when the hardware requirements cannot be met to use hardware virtualization.

(O2) Stealth: The use of in-guest data-collection techniques have been shown to be vulnerable to detection, even when such collection is implemented in the kernel. Thus, the primary challenge to meet this objective is developing a data-collection method that operates exclusively out-of-band. Additionally, in prior work such out-of-band approaches required significant changes to be made to the hypervisor, which may introduce discrepancies in the behavior of the system which may reveal the analysis environment. For effective stealth the underlying hypervisor should not have to be modified such that it may present differences as compared to when it is deployed under normal circumstances.

(03) Fidelity: Modern malware notoriously hides aspects of its operations by modifying the state that the operating system reports to standard tools. Thus, the out-of-band collection method has to obtain state information such that the fidelity of the data is preserved in light of such malicious modifications.

(O4) Isolation: The malware collection system has to isolate the incoming attack traffic from each other so that data attribution can be performed more accurately. While

virtualization and out-of-band monitoring provides a base-line of isolation, the virtual honeypots also have to be kept separate on the network as well.

Other challenges facing the development and operations of honeypots include deciding the network placement, choosing and configuring vulnerable services, avoiding the attacker using the honeypot to infect others and providing auxiliary information on the captured malware. While these challenges are important aspects of honeypots, we consider these mostly out-of-scope.

3.2 System Design

In our prototype we opted to for a combined honeypot archicture in which both low- and high-interaction honeypots are used, commonly referred to as hybrid honeypots. This design choice has been warranted by our primary goal to obtain binaries, even when hardware resources are insufficient for the use of hardware virtualization. In such cases the LIH can be used as a fall-back mechanism to provide at least some interaction to the attacker. An overview of the system can be seen in Figure 1.

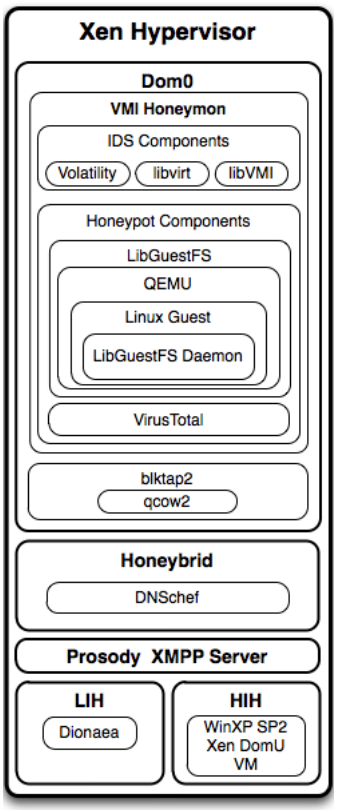

Figure 1: Malware collection system implemented on Xen

The incoming malware connections are routed using Honeybrid [11], which decides which type of honeypot should handle it to based on the current utilization of the system. The LIH has been a standard emulation based system called Dionaea. The choice of Dionaea has been made based on the variety of services Dionaea is able to emulate, enabling the capture of a high variety of malware. The HIH has been developed with the use

of hardware virtualization and its implementation details are discussed in the following.

3.2.1 Hardware virtualization based subsystem

During our implementation process we experimented with two open-source hypervisors, Xen [55] and KVM [68]. While our system was operational on both platforms, Xen was chosen as no custom patching of the hypervisor was required. Xen is a bare-metal hypervisor running at the lowest and most privileged mode of the CPU. In such a system, Xen presides over multiple operating systems, known as domains, with the Xen management domain (dom0) granted privileged access.

In order to gain access to the guest memory, we used the open-source library LibVMI [76], which is an introspection library written in C that builds upon low-level functionality provided by the hypervisor itself. Through this API we gain visibility into many aspects of a virtual machine, including CPU state and guest memory. Running in userspace within dom0, a program utilizing LibVMI is isolated from the monitored guest and therefore increasingly protected from tampering by the software inside the VM. Our prototype system passively monitors the sandbox, thus it further minimizes the potential of detection.

Command	Description	Is scanner?
ssdt	Print the Native and GDI System Service Descriptor Tables.	No
ldrmodules	Cross-reference memory mapped files with the 3 PEB DLL lists.	No
apihooks	Detect IAT, EAT, and Inline hooks in process or kernel memory.	No
idt	Dump the Interrupt Descriptor Table and check for inline API hooks.	No
gdt	Dump the Global Descriptor Table.	No
callbacks	Print kernel callbacks of various types.	No
psscan	Scan memory for EPROCESS objects.	Yes
modscan	Scan memory for LDR_DATA_TABLE_ENTRY objects.	Yes
driverscan	Scan memory for DRIVER_OBJECT objects.	Yes
filescan	Scan memory for FILE_OBJECT objects.	Yes
mutantscan	Scan memory for KMUTANT objects.	Yes
thrdscan	Scan memory for ETHREAD objects.	Yes
sockscan	Scan memory for socket objects.	Yes
svcscan	Scan the Service Control Manager for information on Windows services.	Yes

Table 1: Volatility tests utilized.

With access to the low-level virtual hardware state, Volatility is used for high-level state reconstruction. Volatility is the most popular open source memory forensics tool,

incorporating templates for Windows and Linux, and several plugins such as seen in Table 1. While some of the Volatility plugins rely upon the guest kernel entry-points (such as the linked list of running processes) to perform state-reconstruction, a set of *Scanner* plugins are also available that bypass the standard entry-points entirely. This analysis method allows detection of hidden or otherwise disguised kernel data.

The *Scanner* plugins operate by fingerprinting each byte in the inspected VM's memory as a candidate for the target datastructure, thus require a continuous sweep of the entire memory of the guest. The fingerprint builds on the observation that on Windows datastructures allocated on the kernel heap are appended a so-collaed *pool header*, shown in Listing 1. This header allows Windows to keep track of structures with a size less then 4096 bytes, the minimum size that can be addressed using memory paging. Windows *pools* such small structures onto as few pages as possible for efficient memory management. This pool header consequently identifies the type and size of the structure that follows. While the type description is not binding, it provides an alternate method to obtain a view into the kernel heap, as opposed to following standard kernel datastructures known to be modified by some rootkits.

The sandbox VM was running Windows XP SP2 with 128MB RAM as a VM. Paging within Windows was turned off to maximize the access to the state of the VM without having to take into consideration paged-out memory. Automated system processes were also disabled, such as automatic updates, automatic defragmentation and screensavers. We installed no additional software in the HIH and therefore only the default TCP ports were open: 135, 139 and 445. Since LibVMI and Volatility supports all versions of Windows, our prototype could also operate on a wide-range of HIHs. So far we have only experimented with Windows XP SP2 as it had been the standard platform for other introspection based Honeypots [24, 30].

The memory scans were initiated both when a time-out was reached which we set at 10 minutes, or in case the compromised sandbox attempted to initiate a network connec-

tion to a third-party IP. As we considered unblocked propagation of malware unacceptable, all such attempts immediately resulted in the sandbox being suspended for analyses and then reverted. We were able to judge the connection to be malicious by ensuring that the sandbox does not initiate any network connections when left alone and idle, thus the only scenario for it initiating connections is if it is caused by malware.

3.2.2 Network setup and fall-back system

One of our core objectives is to provide a separation between intrusions on the HIH so that the memory footprint we obtain can be assigned to a single intrusion session. This isolation requirement has been extended to control the outgoing connection attempts initiated by HIH as well to minimize the security risk an infection poses to other entities. Figure 2 shows the layout of our network setup. In our prototype we based our network setup on the open-source Honeybrid [11] system, which is a honeynet coordinator designed to reduce the load on HIHs by utilizing low-interaction honeypots (LIH) as filters on the incoming traffic. Honeybrid also provides fine-grained control over outgoing connections from the honeynet, which can be easily extended by creating additional decision modules for Honeybrid, a feature we took full advantage of.

The Dionaea low-interaction honeypot [25] has been used as our back-up system and filter on incoming traffic. A custom Honeybrid module has been created to monitor Dionaea's actions via chat messages to prevent repeated captures from the same IP. Also, utilizing the chat messages Honeybrid can filter out IPs that have previously dropped a payload on the LIH. Furthermore, network connections are only transferred to the HIH if the connection passed the TCP handshake stage, providing additional filtering in case an attacker is sending a *syn-flood*.

The control module created for Honeybrid redirects outgoing DNS queries to DNSchef [26] for logging purposes and only allows outgoing connections back to the attacker. All other connection attempts result in Honeybrid pausing the HIH, initiating a Volatility scan

of the VM and reverting it to the origin state. When Honeybrid observes no traffic to the HIH from the attacker for two minutes, the same actions are taken: initiating Volatility scans and reverting the VM.

To avoid a single attacker taking over our HIH for an extended period of time, we placed an additional timer on the attack sessions which sets an absolute allowed time-frame of 10 minutes. After 10 minutes the attacker is redirected to the LIH and the HIH is scanned for artifacts and reverted to the clean state. We chose 10 minutes as a maximum limit because in our observation with Dionaea infections occurred within a minute of the connection being established while others reported an average of three minutes [66]. However, malware may choose to wait longer as a detection avoidance technique.

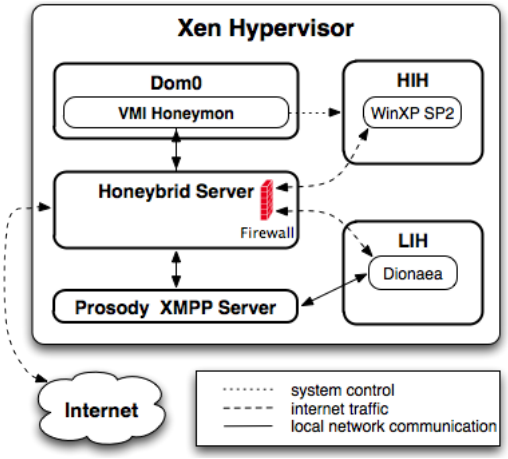

Figure 2: Overview of the different communication paths within the honeypot. Only one connection is allowed to the HIH at any given time.

3.2.3 Fidelity and the path to Scalability

To perform anomaly detection in the output of Volatility's various plugins, we created *VMI-Honeymon* (*honey*pot *mon*itor), to execute Volatility and parse the results in parallel. In our setup, *VMI-Honeymon* resides in a separate domain from Honeybrid, listening to commands through a TCP socket. The anomaly detector compares the Volatility results obtained from the live HIH to results that were obtained from the clean state of the HIH.

By performing a differential analysis akin to running `diff`, *VMI-Honeymon* can effectively flag all deviations observed. This method has been effective in detecting a wide range of changes in the HIH during infections.

One of the scan plugins of Volatility was of particular interest to us in the context of malware collection, as it performs a scan for `_FILE_OBJECT`s on the kernel heap. Using the results obtained from this scan, *VMI-Honeymon* effectively obtains a list of files that are likely candidates for malware binaries so that we don't have to scan the entire filesystem for changed files later. To extract these files, we utilize Libguestfs [74], an open source library developed by RedHat intended for analysis and manipulation of guest filesystems. Libguestfs operates by launching a small Linux appliance inside Qemu and attaches to the filesystem of the running VM. In our setup Libguestfs attaches to the guest filesystem in read-only mode to avoid creating discrepancies in the filesystem. As *filescan* provides a very rich set of information, some of the files that were not actually changed but have simply been opened for reading by processes. To filter out these false-positives, VMI-Honeymon compares the checksum of the candidate files in the running VM to the origin state and only extracts new files or files with changed checksums.

The extracted files are further checked to determine if the file is a binary using the `file` command. When the file created has a proper PE header it also gets submitted to VirusTotal (VT) [115] for further analysis. VirusTotal is an online resource which provides an API for automatically scanning samples with a set of popular antivirus software. We used VirusTotal to estimate the number of binaries containing malware. As VirusTotal uses many antivirus products to scan the submitted samples, we can effectively determine whether the sample is classified or unknown.

To automate back-up and restore procedures, *VMI-Honeymon* takes advantage of functionality provided by LibVirt [75] to automatically save the entire memory of the HIH and revert it when required. By saving the entire memory and filesystem of the VM in an initial ”snapshot” operation, *VMI-Honeymon* can quickly revert the HIH, without requiring a

complete reboot of the VM. To stream-line the process of filesystem restoration, we utilize the qcow2 (Qemu copy-on-write) format through the Xen blktap2 driver. Qcow2 works by utilizing a base-image of the filesystem and a copy-on-write image that keeps only the changes that were made. Since no changes are written to the base-image during operations on the qcow2 image, the two main advantages are its small storage requirement and the ability to keep infections in a contained environment. Furthermore, this technique also opens the door for effective scalability, as multiple copy-on-write disks could utilize the same backend disk simultaneously.

3.3 Initial Experiments

Once the system has been implemented with the above outlined design, our goal was to determine whether the system is capable of capturing binaries, which is the primary goal of the system. For this, the system has been exposed on the public Internet without firewall or IDS systems filtering the incoming traffic. Furthermore, we were interested in determining how the system behaves and the types of auxiliary information it generates during regular operations.

3.3.1 Performance

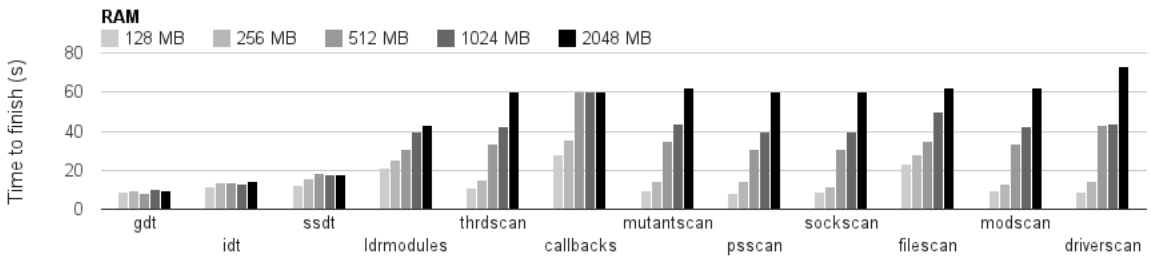

Figure 3: Benchmarks of Volatility scans

During regular operation of *VMI-Honeymon*, Volatility scans and baseline comparison took an average of 30 seconds; reverting the VM to a clean state took an average of 25 seconds. We performed further benchmarks of our system with varying VM memory

sizes for which the results can be seen in Figure 4. The hardware specs of our system were as follows: second generation Intel i7-2600 quad-core CPU with Intel VT-x and VT-d, Intel DQ67SW motherboard, 16GB DDR3 1333Mhz RAM and two 1.5TB SATA 6.0Gb/s 5900RPM hard-drives using Intel Rapid Storage Technology in RAID 1.

The *snapshot* operation consists of creating a memory backup, a filesystem backup and running initial Volatility scans on the VM. Libguestfs checksum creation of the entire filesystem is a separate step in the snapshot operation not affected by the memory size of the VM and which takes an additional 5-6 minutes to finish (not shown on Figure 4). From our benchmarks it is clear that the memory scanning plugins of Volatility take longer with growing memory size, seen in Figure 3. Nevertheless, even with 2GB of RAM the majority of the scans finished in about a minute.

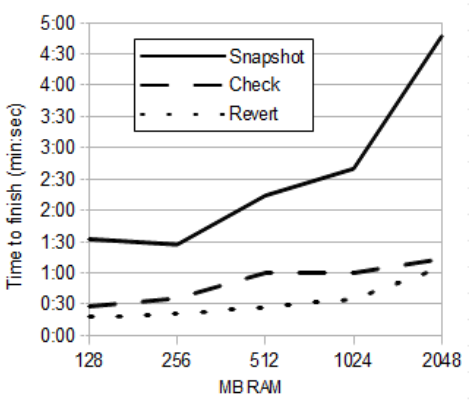

Figure 4: Benchmarks showing the setup times of VMI-Honeymon. Snapshot operations are performed only once, while Check and Revert are performed at the end of each session.

3.3.2 Testing with Metasploit

To verify that our system is able to detect intrusions, we exploited the HIH using Metasploit's *ms08_067_netapi* [93] remote code execution exploit for the SMB stack, an exploit also used by the Conficker malware [81]. This exploit was chosen as Conficker infections had been observed on our LIH with Dionaea on multiple occasions. The anomaly detection did indeed pick up the changes in the memory, particularly the presence of our

remote shell, cmd.exe and the presence of a new TCP socket. This intrusion resulted in a total of 127 changes in memory, predominantly new threads spawning according to *thrdscan*, various dll's being loaded into memory according to *ldrmodules* and those dll's being opened for reading according to *filescan*.

We conducted an additional test where we used Metasploit's auxiliary *smb_version* scan to fingerprint the HIH and ran our anomaly detection tool to see if a simple fingerprint operation results in detectable changes. Indeed, running *smb_version* resulted in a set of changes in *filescan* and *ldrmodules* in relation to the kernel modules "spools" and "browser". While the number of changes was small, six changes in the output of *filescan* and a single change in *ldrmodules*, this test illustrates the sensitivity of our anomaly detector.

3.3.3 Rootkits

To further test our system we obtained recent samples from contagioexchange [86] of the "Sinowal Mebroot Torpig" family and manually infected the HIH with the trojan [1]. This malware was chosen as it is known for disabling itself when it detects a virtualized environment [77]. We observed significant changes all around the memory footprint of the VM, most notably, changes in the Interrupt Descriptor Table (IDT) and the Global Descriptor Table (GDT). A total number of 78 changes were detected, including changes with *psscan*, *mutantscan*, *filescan* and *ldrmodules*. The only filesystem change occurred during execution of the malicious binary we placed in the HIH. No outgoing connection attempts were detected during this infection. Table 2 shows a snippet of the memory changes detected by *VMI-Honeymon*.

Similar results were obtained while infecting the HIH with a TDL4 sample (md5sum 1ca0ca80bf70ca999be809edc2606ac0). This malware was also chosen for the known behaviour of disabling itself when a virtualized environment is detected [38]. The infection triggered detection by modifying the IDT and GDT kernel tables and by changing the

output of *mutantscan*, *filescan* and *ldrmodules*.

Scan	Result
gdt values	”Sel Base Limit Type DPL Gr Pr”
gdt	”0x38 0x0 0xfff Data RW Ac 3 By P” is missing/changed!
gdt	”0x40 0x400 0xffff Data RW 3 By P” is missing/changed!
gdt	”0x48 0x0 0x0 Reserved 0 By Np” is missing/changed!
gdt	New element: ”0x38 0x7ffdf000 0xfff Data RW Ac 3 By P”
gdt	New element: ”0x40 0x400 0xffff Data RW Ac 3 By P”
gdt	New element: ”0x48 0x80d44000 0x177 LDT 0 By P”
idt values	”Index Selector Function Value [Details]”
idt	”21 8 - 0x0” is missing/changed!
idt	New element: ”21 C7 - 0x0”

Table 2: Sinowal Mebroot Torpig detection (snippet).

3.3.4 Live sessions

To expose our system to malicious traffic, it was placed on a University network with the University firewall configured to allow all incoming and outgoing connections. In a period of two weeks we recorded 1,158,477 TCP and 467,173 UDP connections to our honeypot setup.

The HIH was exposed to 6,335 connections during this time in 1,980 sessions. *VMI-Honeymon* extracted a total of 886 unique binaries out of which 236 were verified as being malicious by VirusTotal. Dionaea in the same time captured 1,411 binaries out of which 431 were verified by VirusTotal.

One of the worms that has triggered the vast majority of Dionaea captures on our LIH is Conficker. Similarly, we have observed several Conficker infections on our HIH, constituting 14% of the total number of unique captures and 53% of all captures that had a detection with VirusTotal. We observed several variants of Conficker trying to connect to the default gateway on port 445, possibly trying to propagate on the local network (LAN), and also trying to initiate connections to unknown web-servers. A detection and capture snippet of Conficker can be seen in Table 3. Another subset of sam-

ples, an infection we have not observed previously with Dionaea, triggered detection only with McAfee-GW-Edition according to VirusTotal with a detection named ”Heuristic.BehavesLike.Exploit.CodeExec.L”. Figure 5 shows details of unique binary captures with VirusTotal detection, both from the LIH and HIH.

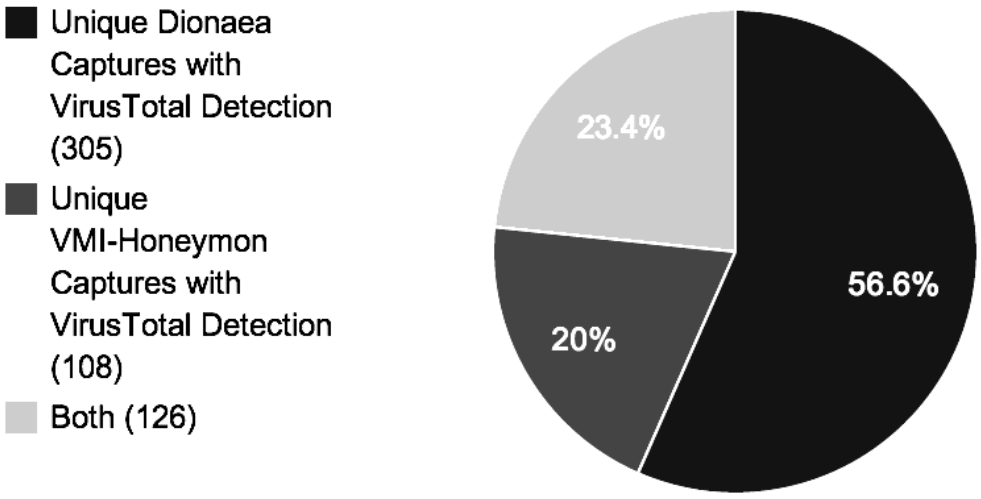

Figure 5: Unique binary captures with VirusTotal detection

In Figure 6 a breakdown of samples attempting to establish a connection to an IP other than the attacker can be seen. This data show us that sessions which resulted in the HIH trying to connect out had a higher number of binary captures with 98% of the captures being verified by VirusTotal. On the other hand, the majority of the sessions with no outgoing connection attempts resulted in no modification to the filesystem. Furthermore, only 26% of the files extracted during these sessions were verified by VirusTotal. As such, it is reasonable to assert that an outgoing connection attempt is a better indicator of a successful exploit but it should not be relied upon as the sole indicator.

The results were further broken down to show how many of these sessions resulted in a binary capture attempt and how many of these binaries were verified by VirusTotal. While the total number of sessions that resulted in the HIH trying to connect outside was only 15%, 44% of unique binaries that had at least one VirusTotal detection were extracted or observed during these sessions. Nevertheless, 42% of the time the captures

extracted during these sessions had no VirusTotal detection. It is reasonable to assert that the unclassified captures are malware binaries as yet unidentified by the 42 anti-virus vendors.

An estimated 45% of the sessions were scans or other benign connections as they resulted in fewer than 20 changes in the memory analysis results. While the sheer number of changes is not a reliable metric, we based our estimate on the observations from the manual scan with Metasploit and the sessions for which we observed either an outgoing connection and/or extracted a binary with a VirusTotal detection. During our tests with Metasploit, only 7 changes were observed, suggesting a relatively low number of changes for such interactions. Verified exploit sessions had a minimum of 62 changes, a maximum of 962 and an average of 339 with a standard deviation of 313, thus we find this a reasonable metric to identify such connections.

Action	**Result**
sockscan values	”Offset PID Port Proto Address Create Time”
sockscan	New element: ”0x00a6e7a0 984 1039 6 TCP 0.0.0.0 2012-04-10 04:10:37”
sockscan	New element: ”0x01085e98 984 1032 17 UDP 127.0.0.1 2012-04-10 04:09:48”
sockscan	New element: ”0x0109ae98 984 7054 6 TCP 0.0.0.0 2012-04-10 04:10:20”
sockscan	New element: ”0x010c1b20 984 1038 6 TCP 0.0.0.0 2012-04-10 04:10:34”
sockscan	New element: ”0x010d8480 1056 1037 17 UDP 0.0.0.0 2012-04-10 04:10:34”
sockscan	New element: ”0x010f2780 984 1036 6 TCP 0.0.0.0 2012-04-10 04:10:34”
sockscan	New element: ”0x04ca5a68 984 1041 6 TCP 0.0.0.0 2012-04-10 04:10:41”
filescan values	”Phys.Addr. Obj Type #Ptr #Hnd Access Name”
filescan	New element: ”0x010c7120 0x80e94ad0 1 0 R–r-d /WINDOWS/system32/fmrmj.dll”
CAPTURE	”/WINDOWS/system32/fmrmj.dll”

Table 3: Conficker infection (snippet).

3.4 Improving Scalability

With our initial experiments successful, we turned our attention to the four core objectives. Foremost, in our initial experiments only a single HIH was utilized either though the system has hardware resources available to run multiple. Thus, our next prototype focused on improving scalability by enabling the concurrent use of multiple HIHs.

As we have described in our initial experiments, copy-on-write disk enabled us to

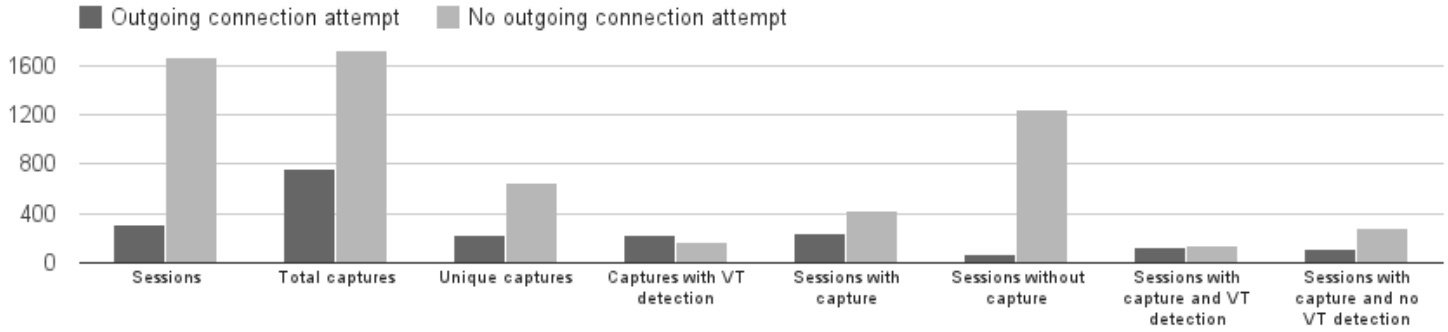

Figure 6: Honeypot statistics. Note: A *Connection Attempt* is defined as any attempt made by the HIH to connect to an IP other than the attacker's IP. A *Session* is defined as all interactions with an attacker.

rapidly revert the HIH environment. This technique also enables us to deploy multiple clone VMs in parallel; nevertheless, to truly maximizing the number of instances that can run in parallel the memory footprint of these VMs need to be taken into account as well. To explore this aspect of hardware virtualization based systems, we enhanced our prototype system to reduce the hardware cost of running nearly identical VMs. By effectively being able to deploy memory sharing between virtual machines, our system further improved the *scalability* of malware collection using the high-fidelity HIHs. As we developed this prototype, we discovered additional challenges such a setup needs to address: transparently perform Network Address Translation (NAT) on incoming and outgoing network traffic using virtual machines that share MAC and IP addresses.

3.4.1 Memory sharing

To achieve dense HIH deployment while avoiding the linear memory requirements we take advantage of Xen's native memory sharing subsystem. Memory sharing enables the creation of nearly identical clones that transparently share the memory pages that have not changed during HIH execution. The system is designed so that the origin (parent) VM is paused, and all clones created initially point to the parent's static memory. When a clone writes to memory, Xen performs a copy-on-write (CoW) routine and duplicates the memory page for the clone, providing an optimized use of the overall memory of the physical host.

Recent developments of the memory sharing subsystem enables the creation of nearly identical clones without requiring modifications to Xen itself. While the subsystem is capable of carrying out Potemkin-style cloning of VMs, much of the toolstack for performing this has been kept proprietary and thus unavailable to the general user. Fortunately, cloning can also be achieved by performing the standard snapshot-and-restore routine with the XenLight library and de-duplicating the memory pages of the clone afterwards. While the approach is slower then the flash-cloning would likely be, it is sufficient for evaluating the

nature of several HIHs in a memory sharing setup.

A key aspect in performing the XenLight (XL) snapshot-restore routine is that the snapshot operation is performed only once when a VM is being designated as a honeypot origin. This snapshot operation also encompasses the scanning of the VM's memory with Volatility and performing a full filesystem fingerprinting with LibGuestFS, so that later infections can be correlated to a known clean-state of the honeypot. The domain configuration is dynamically updated during cloning to change some metainformation about the VMs configuration, such as disk path and domain name, which allows the standard Xen toolstack to create the clone domains.

3.4.2 Clone-routing

As the combination of CoW RAM and filesystem enables the creation of identical clones, from a networking perspective the identical clones pose a new challenge: the configuration of each network interface will also remain identical. This effectively means that the clones will share both the MAC and IP address of the original VM, thus placing these clones on the same network bridge leads to MAC and IP collisions. This collision represents a roadblock to the standard Linux IP routing stack as the connections could get mangled and rejected. Both Potemkin and SnowFlock solve the problem of clone-routing by performing post-cloning IP reconfiguration of the VMs. However, at this we had to avoid performing such reconfiguration as it would have required an in-guest. Furthermore, the reconfiguration inadvertently would change the initial state of the memory in the clone, leading to noisy analysis results when comparing memory states between the clone and its origin. This 'noise' is caused by the process of unpausing the VM to alter settings, which allows all processes to resume execution, subsequently causing potentially substantial deviation from the original memory state.

To ensure the creation of truly identical clones and enable pure comparison between a clone and the origin, we retain the MAC and IP of the original VM for each clone. To

do so, each clone is placed upon a separate network bridge to provide isolation for the MAC of the clone and avoid a collision. As seen in Figure 7, the clone's bridge is also attached to the VM that runs Honeybrid. This solution enables us to avoid collisions on the bridge, but requires custom routing to be setup on the Honeybrid VM that can identify clones based on the network interface they are attached to instead of their (identical) IP addresses.

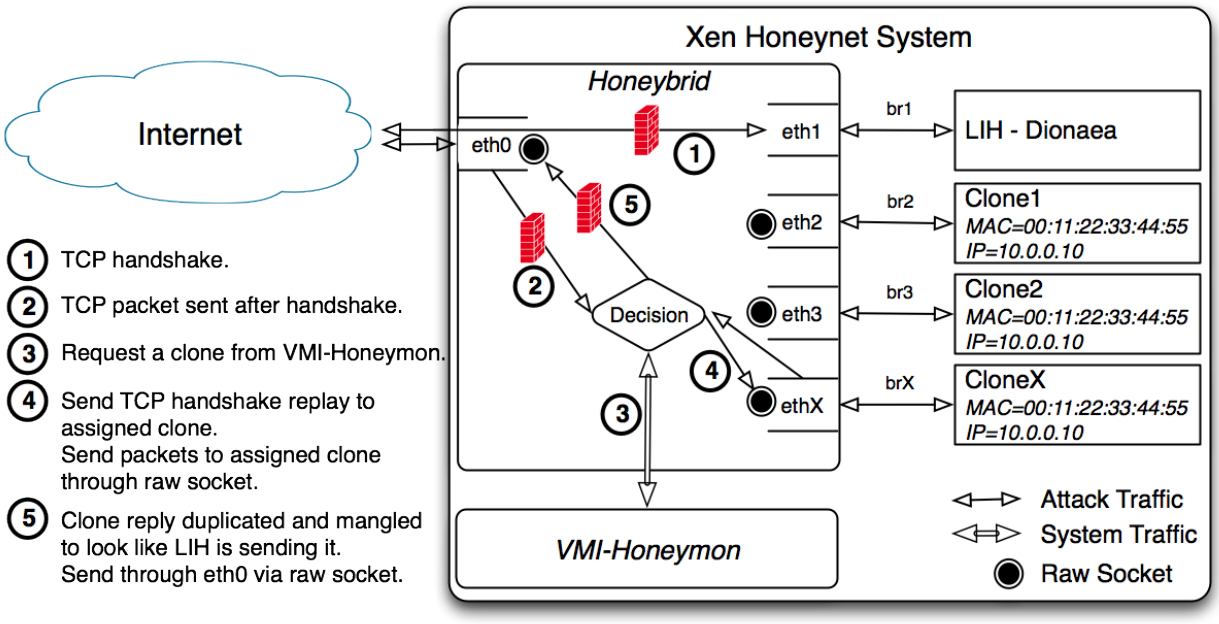

Figure 7: Clone routing layout - externally initiated.

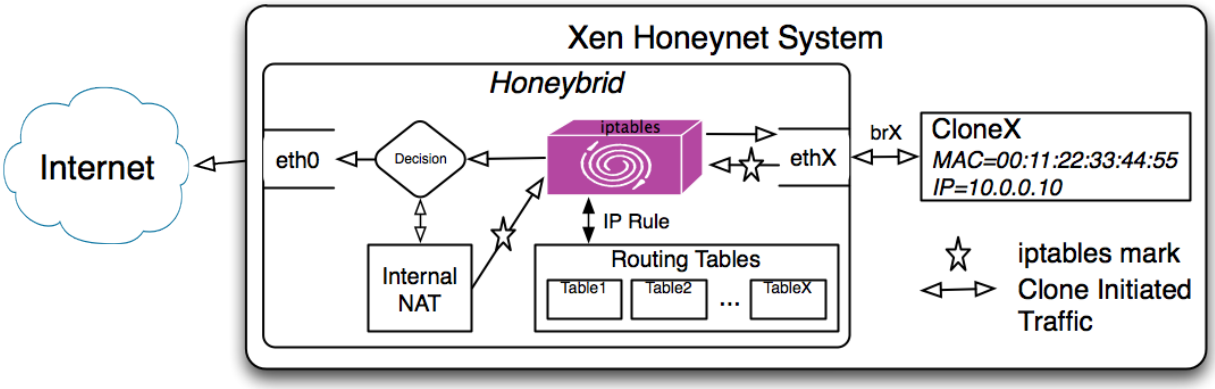

Figure 8: Clone routing layout - internally initiated.

In order to provide transparent connection switching between the LIH and an HIH, Honeybrid acts as a man-in-the-middle. Using iptables, each incoming connection in the Honeybrid VM is DNATed to the LIH and then queued to be processed by Honeybrid. Each TCP connection performs the TCP handshake with the LIH, and if the connection sends any additional packets, Honeybrid evaluates if the connection should be switched to an HIH. The evaluation is performed in conjunction with VMI-Honeymon where Honeybrid asks for a random available clone from VMI-Honeymon through an SSH tunnel. When there is one available, VMI-Honeymon responds with the clone's name and Honeybrid looks up the clone's interface from the pre-defined configuration file. If VMI-Honeymon reports that all HIHs are taken, the attacker's IP is pinned to be able to only interact with Dionaea. When the connection is switched to an HIH, Honeybrid replays the TCP handshake with the HIH. The incoming packets bound to the LIH thereafter are copied and updated to be directed to the clone and transmitted through a raw socket fixed to the clone's network interface. The use of raw sockets forces the incoming packets to egress on the proper bridge. Packets in transit back from the HIH are also copied updated on-the-fly to contain the IP of the LIH, thus allowing the standard Linux routing tools to take the packet form here and route it normally upstream.

For connections that are initiated from a clone, which happens for example when an exploit performs a reverse TCP connection, Honeybrid must be aware to route incoming packets for that connection back to the clone that initiated the connection. We use additional routing tables to specify which interface each clone is bound to and by using iptables marks and ip rules we can direct incoming reply packets to specific routing tables which in effect lead to specific clones, shown in Figure 8. We utilize Honeybrid to set the iptables mark on the reply packets by looking up Honeybrid's internal NAT table to identify the original source of the connection.

3.5 Final Experiments

With the improved scalability available to us by utilizing CoW-memory as well as CoW-disk, we further evaluated our system's performance and effectiveness. Several tests have been conducted which focused on the scalability of the memory sharing subsystem when used with Windows XP SP2 x86, Windows XP SP3 x86, and Windows 7 SP1 x86 clones. The experiments were conducted on a single server with the following hardware specs: second generation Intel i7-2600 quad-core CPU, Intel DQ67SW motherboard and 16GB DDR3 1333Mhz RAM. In our tests the Windows systems were running with the minimum recommended memory, which is 128MB RAM for Windows XP x86, and 1GB RAM for Windows 7 SP1 x86.

3.5.1 Idle clones

An important aspect of our intrusion detection approach and of effective memory sharing is to limit the memory changes that are not related to an incoming attack. While in Windows XP the number of background processes that generate unrelated memory changes are limited to a handful of services (automatic updates, NTP, background disk defragmentation and background auto-layout), in Windows 7 the number of such services have increased significantly. The effect of these background services on Windows 7 is significant as even within two minutes the amount of shared memory decreases below 25%, effectively requiring over 750MB RAM to be allocated to the clone. At the same time, the clone itself reported only using 26% of its available memory, therefore the allocated CoW memory pages had only short-lived purposes.

By disabling background services which required the allocation of unnecessary resources and polluted our Volatility scans, we were able the minimize the resources allocated to idle clones. The disabled Windows 7 services include prefetch, superfetch, BITS, RAC, indexing, offline files and font cache. Figure 9 shows the resulting memory sharing

state of the clones when idle, in terms of shared memory and Figure 10 in terms of additional RAM allocated to the Windows 7 SP1 clones. It is important to note that disabling services in the HIH will inevitably impact the attack surface of the HIH, as these services may contain vulnerabilities that could be looked for and/or exploited by the attacker. Since the services we disabled were not listening for incoming connections on the network we deemed their absence to be a reasonable trade-off from a network intrusion perspective. In another scenarios one may opt to leave all services running and trade the benefits of memory saving achieved via CoW to improved fidelity.

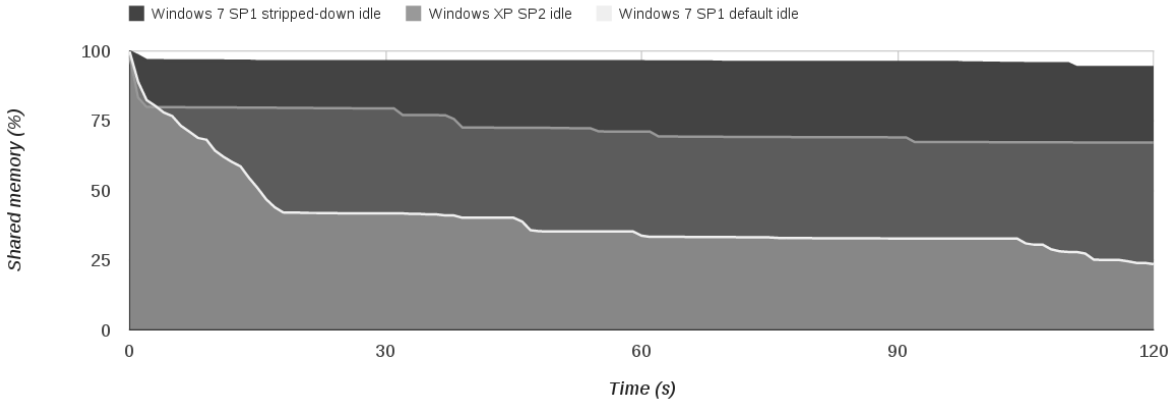

Figure 9: Clone shared memory when system is idle.

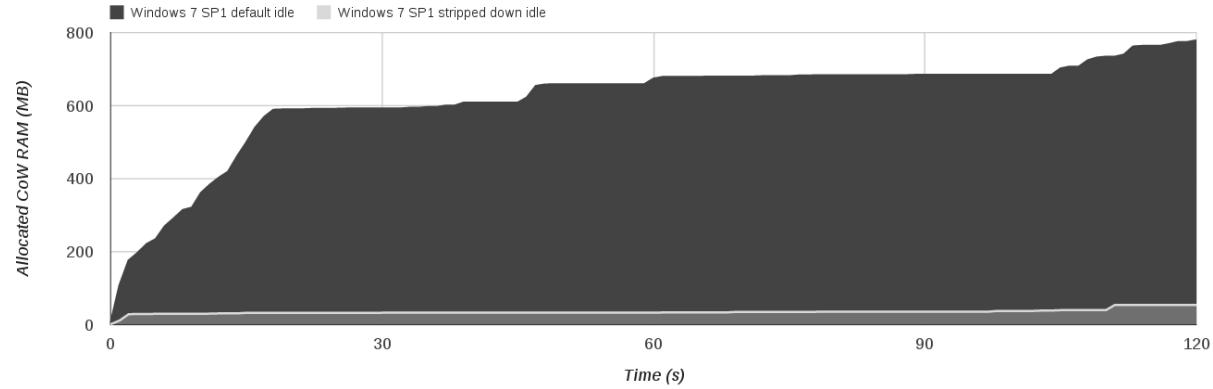

Figure 10: CoW RAM allocated when system is idle.

3.5.2 SMB and RDP

The following tests were targeting open services in our clones, namely the standard SMB and Remote Desktop services as both of these services have known vulnerabilities. The RDP sessions had significant impact on the RAM allocated to both the Windows XP and Windows 7 clones, reducing the amount of shared memory to 25% in case of the Windows XP clone and 50% for the Windows 7 clone, as seen in Figure 11. In terms of actual RAM allocation, the Windows 7 clone's 50% memory allocation translates to allocating 500MB RAM for the clone, while the Windows XP clone at 75% required 96MB RAM.

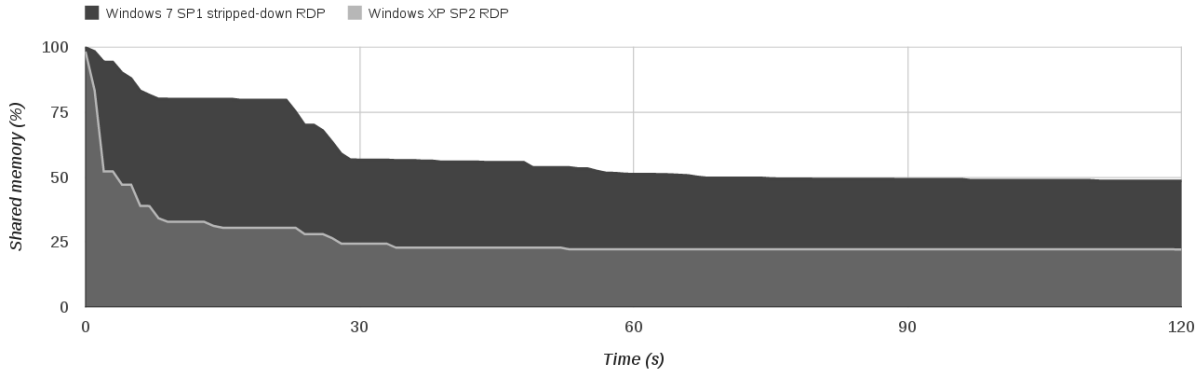

Figure 11: Clone shared memory after RDP connection.

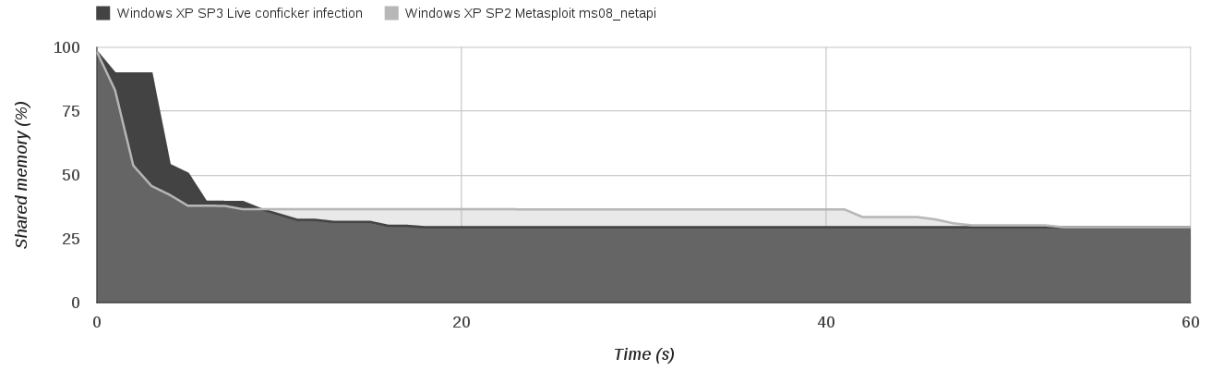

Figure 12: Clone shared memory after SMB exploitation.

The SMB tests were only conducted on Windows XP clones as Windows 7 SP1's SMB stack has no publicly available exploit. We used a manual Metasploit exploit session (ms08_067_netapi) to benchmark the effect on the Windows XP clone when used

with a meterpreter payload that performs a reverse TCP callback. This exploit was chosen because Conficker uses the same vulnerabilities, which has been observed many times during our live tests. Figure 12 shows the result of the benchmark compared to a live Conficker infection. Only the first 60 seconds were benchmarked since the Conficker infection performed a connection attempt to a third party at that point, triggering our pause-scan-revert operation with VMI-Honeymon. The Windows XP clones retained 25% of their memory in a shared state, which translates to saving 32MB of RAM.

3.5.3 Live sessions

Our experiments were conducted using multiple HIH back-ends drawn from a pool of clones consisting of five Windows XP SP3 x86 and five Windows 7 SP1 x86 VMs. Each Windows VM was configured with the firewall, automatic updates, time synchronization and memory paging turned off and remote desktop enabled. Windows 7 had additional adjustments as described previously in Section 3.5.1.

For the live captures we utilized a single IP on a university network with all firewall ports open. Over two weeks of activity, we recorded a total of 52761 connections out of which 6207 were forwarded to an HIH. Currently we forward any incoming connection that passes the TCP handshake to an HIH (if one is available), regardless of whether the HIH is actually listening on the port the attack is targeting. In this way, 1466 forwarded connections never actually established real communication with the HIHs, because these connections targeted ports that were closed (MsSQL, MySQL, SSH, HTTP, VNC).

For the live sessions, one aspect we were interested in was the concurrency of active clones and the amount of memory savings achieved due to CoW RAM. Figure 13 and Figure 14 shows the breakdown of the concurrency that occurred in our system. Figure 15 and 16 show the distribution of the memory remaining shared at the end of the clones' life cycle.

While VMI-Honeymon is configurable to scan the clones periodically during their life-

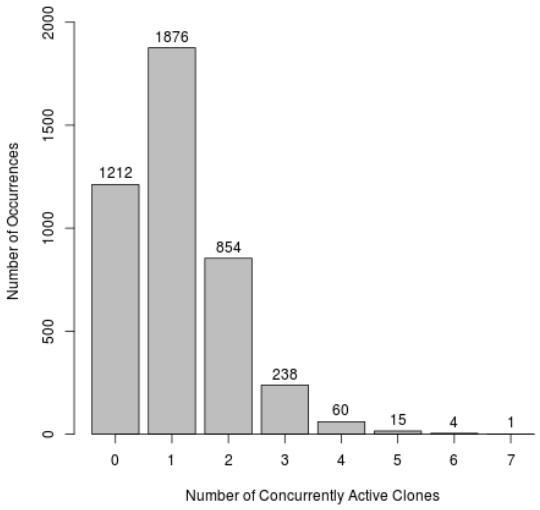

Figure 13: Clone activity by number of occurrences.

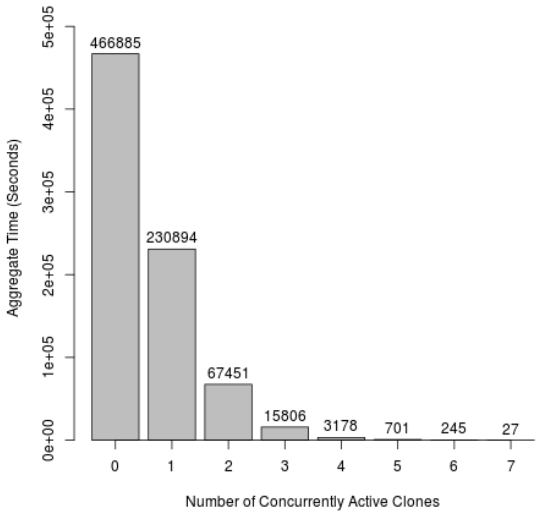

Figure 14: Clone activity by time spent in each state.

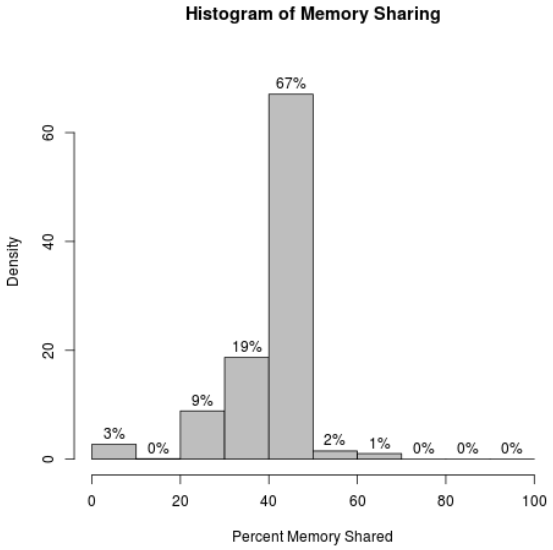

Figure 15: Shared memory distribution of Windows XP SP3.

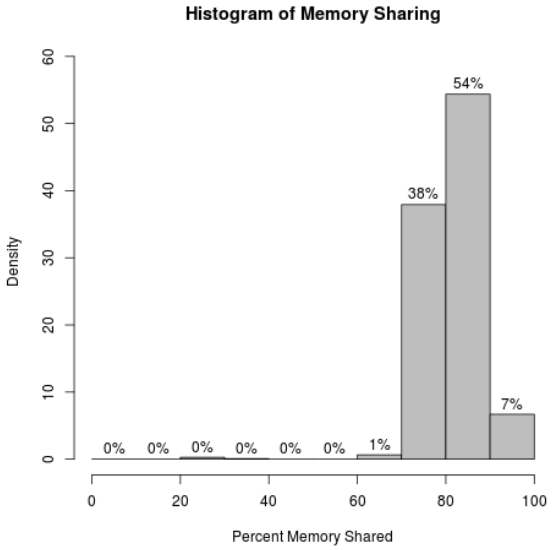

Figure 16: Shared memory distribution of Windows 7 SP1.

span, we decided to limit such scans to a single instance which happens when the clone reaches its maximum allowed life-span or when a network event is detected. The maximum life-span was set at two minutes, which is cut short if the clone initiates a connection to an IP other than the attacker's. The highest concurrency of active clones was observed as seven, therefore our pool of ten clones was never depleted. The HIHs were actively handling incoming attack traffic 41% of the time during our experiment.

By using the information gathered during these sessions we calculate the projected memory savings when running multiple clones concurrently, shown in Figure 15 and Figure 16. From these projections it is clear that the savings are more significant when the base memory of the HIH is large, as in the case of Windows 7 SP1, allowing for a larger percentage of the overall memory to remain shared. We estimate that we would be able to run 40 Windows 7 SP1 clones concurrently and not run out of memory even if all forty clones were three standard deviations above the observed average memory allocation with our 16GB RAM limitation (this would still use only 13.34GB RAM out of the available 16GB). Similarly, we would be able to run 140 Windows XP SP3 clones concurrently and not run out of memory which would allocate 14.6GB RAM assuming all clones are three standard deviations above the observed average.

The malware samples we obtained were all Conficker variants verified by VirusTotal and all of the samples were extracted from the Windows XP HIHs. Nevertheless, we have observed several intrusions in our Windows 7 HIHs as well, which resulted in the clones trying to perform DNS queries. The service exploited during these attack sessions were against the SMB server running on port 445. To allow these intrusions to further interact with the HIH to potentially drop a payload we will be refining our firewall policy to allow some DNS queries and to allow connections to the IP's mapped in the DNS response (with certain rate-limiting applied as to avoid potential malware propagation from within the honeynet).

While the combination of CoW RAM and filesystem enables the rapid creation of iden-

tical VM clones from a networking perspective the identical clones pose a new challenge: the network interface in each clone will also remain identical, sharing both the MAC and IP address of the original VM. Placing these clones on the same network bridge leads to MAC and IP collisions that prevents proper routing. Prior systems employing similar techniques, such as Potemkin [120] and SnowFlock [71] solved the problem of clone-routing by performing in-guest IP reconfiguration of the VMs. As our system aims to achieve *stealth*, such in-guest network reconfiguration could leave identifiable artifacts within the sandbox and should be avoided.

Thus, to be able to retain the MAC and IP of the original VM, each clone is placed upon a separate network bridge to provide isolation for the MAC of the clone and avoid a collision. As seen in Figure 7, the clone's bridge is also attached to the VM that runs Honeybrid. This solution enables us to avoid collisions on the bridge, but requires custom routing to be setup on the Honeybrid VM that can identify clones based on the network interface they are attached to.

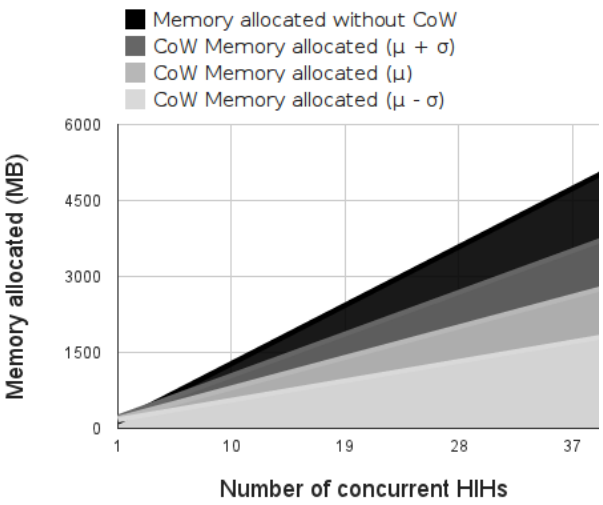

Figure 17: Projected memory savings of Windows XP SP3. μ=75.52MB σ=10.1MB

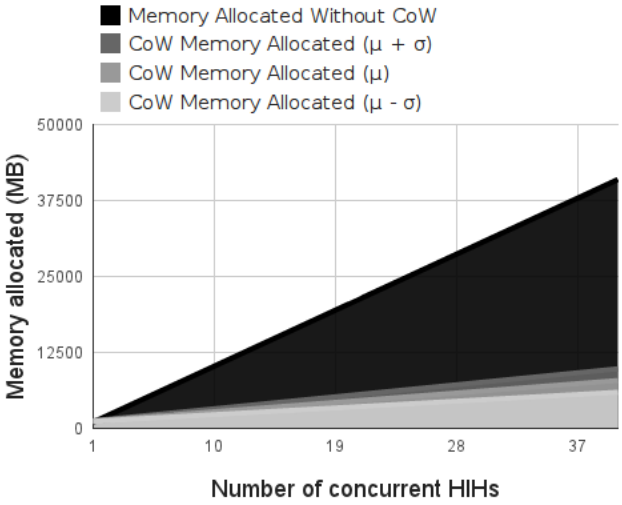

Figure 18: Projected memory savings of Windows 7 SP1. μ=170.94MB σ=48.3MB

In our experiments we evaluated the effective memory savings gained by the utilization of memory sharing. As seen in Figure 17 and 18, the memory sharing approach has a

great potential to scale the honeypot operation, especially when the base honeypot image requires a high amount of RAM.

3.6 Summary

As these experiments prominently show, hardware virtualization offers a variety of properties which can be used to great effect when developing malware collection tools. The tools we developed have highligthed during live experiments that using hardware virtualization based honeypots effectively expands the types of malware that can be captured. We also determined limitations that need to be taken into account and should be addressed in future work. To reflect on our core objectives, we summarize our findings in the following.

(O1) Scalability: The use of copy-on-write disk and memory has shown us that hardware virtualization is an effective method to create malware collection systems. In our prototypes we were the first to deploy memory-shared fully-virtualized honeypots. Our experimental data has highlighted that copy-on-write techniques are key in preserving hardware resources. However, the benefits rapidly decrease the longer the collection sessions are active, thus great care must be taken into minimizing the effects of background processes on the memory footprint of the honeypots. Future work should explore the possibility of performing continuous memory deduplication to preserve the benefits achieved herein.

(O2) Stealth: As our system design is focused on performing monitoring with an out-of-band, passive memory scanner, our system minimizes the impact which could be used for detection. As no in-guest agents or hooks are added, it is reasonable to argue that malware has no direct way to determine whether the virtualized system is monitored or not. Furthermore, as the honeypot systems can be configured with great liberty as to what services are running, it would be increasingly difficult for malware to create a static environmental fingerprint of the honeypot. However, once

malware is allowed to communicate with external systems, those external systems may be able to observ the irregular network connectivity patterns of the honeypot and use these to externally flag the network location.

(O3) Fidelity: The passive memory scanners utilized offer a direct way to improve the fidelity of the data collection. As malware is known to modify standard kernel structures used for in-guest monitoring, our scans by-pass these structures entirely. Rather, our tool scans the entire physical memory of the honeypot to find structures allocated on the kernel heap to perform state reconstructions. Thus, at the time the system was prototypes it offered the best visibility into the honeypot. As this technique has now been widely used by forensics tools, new types of malware emerged which deploy methods to counter this technique and these have to be addressed in future work.

(O4) Isolation: While the use of virtualization and purely out-of-band monitoring offers a great base-line of isolation, significant challenges were faced on providing such isolation on the network. The techniques used to maximize scalability presented unique problems for isolation, which have been effectively countered by our custom network setup. It is important to note however, that the rapid development of virtual switches has greatly reduced the effort required to build such segregated networks, which we will also utilize in our next prototypes.

The experimental data collected shows the effectiveness of hardware virtualization as a platform to develop highly scalable, stealthy and tamper resistant malware collection systems. This data clearly indicates that hardware virtualization is an effective platform for the development of malware collection tools satisfying our core objectives. The tools developed during the research have been open-sourced to allow fellow researchers to conduct their own experiments, some of which have already appeared at peer-reviewed confereces [9].

4 Malware analysis

Since the proliferation of metamorphic malware, dynamic malware analysis has been an effective approach to understand and categorize malware by observing the execution of malware samples in a quarantined environment [36, 127]. The interaction between the executing malware sample and the host OS allows dynamic malware analysis systems to collect behavioral characteristics that aid in formulating defensive steps. Thus, in the context of malware analysis the *primary requirement* is to collect information that a malware analyst deems valuable.

In contrast with malware collection, where the *primary requirement* was well defined and limited, for malware analysis the goal as we can see is generic. Evidently, determining what information is deemed valuable is subjective and may differ depending on the end-goal of the analyst. Some analyst may be interested only in the network traffic of the executing malware sample, while for others the trace of system-calls is what contains the most value. Yet another approach may be to simply extract all new and modified files in the hopes of capturing the unpacked version of the malware which could then be looked at with static analysis tools. Thus, in the following we aim to evaluate hardware virtualization based techniques to accomplish only common aspects of malware analysis, with the understanding that it is ultimately up to the analyst to decide what information is deemed valuable.

We first begin with a discussion of the challenges in accomplishing our core objectives in hardware virtualization based malware analysis systems. Afterwards, we present a brief overview of the theory of virtualization and the core virtualization extensions available on modern Intel processors to aid the reader in better understanding the limitations of these extensions that have to be taken into consideration in the system design. We follow with our system's design and implementation details. To evaluate the system under realistic conditions we present extensive tests on a wide variety of malware samples and

provide a brief overview of the type of information we collected and how it may aid analyst in gaining a better understanding of modern malware. We conclude the chapter with our summary evaluation.

4.1 Challenges

There are a number of challenges one has to consider when constructing virtualization based analysis systems. Foremost, the *four core requirements* have to be met, which has thus far not been shown in practice. While in our discussion of prior work in Section 2.3 we highlighted systems that achieved some of the core objectives, thus far no system has been developed that satisfies all four. The main challenges we focus on are summarized in the following.

(O1) Scalability: Malware analysis systems also face linearly increasing *disk* and *memory* requirements as the concurrent analysis sessions are increased. As we have successfully tackled this challenge in our prior prototype malware collection system, the challenge we face is adapting the system to the new use-case.

(O2) Stealth: As with our malware collection system, in-band data-collection techniques are deemed vulnerable, thus our focus is on using purely ouf-of-band collection techniques. However, without an in-guest agent to start the execution of the malware sample the *primary requirement* is at risk. Thus, the main challenge we consider in this thesis is using only out-of-band tools to both monitor and start the execution of the malware sample without leaving an identifiable trace in the sandbox.

(O3) Fidelity: As the *primary goal* is to collect as wide an array of information as possible to allow a high degree of flexibility in tuning the analysis, the main challenge we face is performing the monitoring *actively* for both user- and kernel-mode malware. While in our prior malware collection prototype data collection was performed passively at the end of the collection session, for malware analysis the full execution

trace may contain information that otherwise would be lost. Thus, data collection has to happen live to allow the analysis system to contain the information it deems necessariy.

(O4) Isolation: Our disaggregated system design used during malware collection has been shown to be effective in isolating executing malware samples from each other, as well as minimizing the exposure of the critical system components. However, as the malware analysis system will interact with live malware samples directly, extra precaution needs to be taken into performing this analysis from a de-privileged domain. Furthermore, the recent evolution of virtual switching technology allows us to simplify our network requirements and use standardized VLAN technology to isolate systems on the network.

It is important to note that there is an inevitable trade-off between *Stealth* and *Fidelity*: the more intrusive the data-collection, the more likely the system will introduce time-skews that may be used to detect the monitoring environment [110]. Prior research has extensively delt with providing manipulated time-sources to executing malware samples in an effort to thwart its detection routines [24]. However, in our opinion such efforts are at best incomplete *if* the malware is allowed to communicate with external systems over the network. While innovative approaches have been introduced in recent years to avoid the introduction of such time-skews by performing an additional layer of copy-on-write memory for the duration of introspection [64], such approaches in our opinion only shift the trade-off to be between *Scalability* and *Fidelity*. We argue that currently the trade-off between *Stealth* and *Fidelity* is more desirable as it is still extremely rare for malware to attempt detection with the use of time-skews, while the introduction of extra memory requirement would apply to all analysis sessions. However, this may easily change in the future at which point our approach may need to be adjusted as well.

4.2 Overview of Hardware Virtualization Extensions

In the following we discuss the hardware virtualization extensions that are required for effective run-time malware analysis. While the methods we deployed in Section 3 relied on virtualization to access the memory of the Operating System (OS), by which we achieved *isolation*, modern virtualization extensions offer a plethora of options to achieve effective *interposition* into the execution of the live virtual machine. This ability to perform interposition is critical in our context as that is what allows us to trigger introspection at pre-configured points in the execution of the virtual machine, thus performing live analysis of the executing malware samples.

In their seminal paper "Formal Requirements for Virtualizable Third Generation Architectures" [92], Popek and Goldberg formalized the requirements for achieving full hardware virtualization on modern processors. Among these requirements was the definition for restricting access to resource control properties of the system, putting the VMM in exclusive control of the real hardware. To achieve this, they classified instructions based on *privileged instructions* which are required to *trap* - that is, to transfer control to the VMM instead of actually executing. Such traps are critical for our malware analysis system as they bypass the guest operating system entirely: with the right configuration of traps we can monitor the execution of the sandbox *without the sandbox being aware of this monitoring*.

On modern x86 Intel CPUs there are a handful of such instructions which unconditionally trap to the VMM according to the Intel SDM [56], such as `CPUID`, `GETSEC`, `INVD`, `XSETBV`, `INVEPT`, `INVVPID`, `VMCALL`, `VMFUNC`, `VMCLEAR`, `VMLAUNCH`, `VMPTRLD`, `VMPTRST`, `VMRE-SUME`, `VMXOFF`, `VMXON`. These instructions are mainly used for managing virtual machines and other security features. While these instructions define the absolute minimum of instructions that are required to be trapped, a handful of other instructions can be further configured to also trap to the VMM. These optional configuration

settings are stored in the VM's Virtual Machine Control Segment (VMCS) by the VMM, which is read by the hardware each time the VM is scheduled to execute.

While on today's CPUs the set of optional trapping instructions is quite extensive, it has been a challenge over the years to find the combination of trapping instructions which correspond to high-level system behavior, such as system-calls [91]. The problem is largely due to the fact that the instructions used in these situations can't themselves be configured to trap to the VMM, thus alternative settings need to be configured to work around the hardware limitations. This however often leads to trapping more than what is required. For example, Dinaburg et al. [24], proposed a set of mechanism that relied on changing paging permissions found in the *shadow page-tables* to trigger violations which can be trapped to the VMM. However, due to the granularity of page permissions, this often leads to violations being triggered by events that were not the target.

4.2.1 VM Scheduling

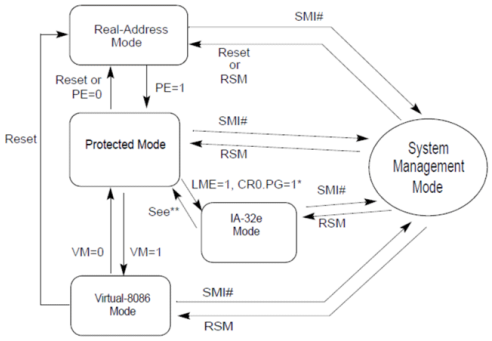

Figure 19: CPU modes available on modern x86 Intel CPUs as descirbed by the Intel SDM [56]

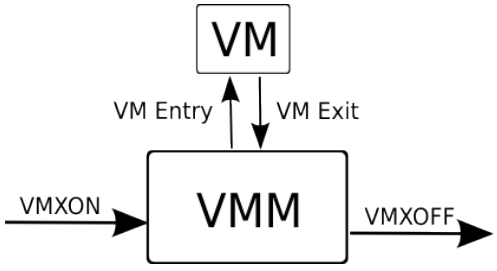

Figure 20: Summary of VMX operation on Intel CPUs

On modern x86 Intel CPUs there are a variety of operational modes, as seen on Figure 19. Virtualization is normally used in Protected mode (32-bit hypervisor) and in IA-32e mode (64-bit hypervisor) with VMX-root privileges. When virtualization is used, the guest operating system can decide for itself which of the available operation modes it needs (except SMM) as it will run in VMX-non-root mode. Virtualization is also available

in System Management Mode, which we will discuss further in Section 5.

As seen in Figure 20, the hypervisor mode is not enabled by default as the `VMXON` instruction needs to be executed first to signal to the processor to turn on the virtualization extension. After the VMM is activated, scheduling of the VM is performed via VM entry operations, such as `VMLAUNCH` and `VMRESUME`. VM exits are triggered by the guest any time a privileged instruction is executed, via the VMX-preemption timer or optionally if guest inactivity is detected when `HLT`, `MWAIT` or `PAUSE` is executed. As of today the Intel manual defines 64 different reason that could cause a VM exit. During these operations, information about the guest VM's state is passed via a memory in the VMCS.

The VMCS is a special control structure established for each virtual CPU, which can be only modified through designated instructions after it is initialized. Each CPU can only execute one vCPU at a time, and a vCPU can only execute on one CPU at any given time. The VMCS stores the guest execution state in the *guest state area* so that the VM can be resumed, while the *host state area* stores the host state to be restored and the pointer to the function to be executed when a VM exit is performed. The VMCS is also responsible to define the set of optional configuration settings for *trapping* additional instructions.

4.2.2 Optional Traps

Among the optional traps that can be configured for Intel CPUs, a handful are of particular interest for VMI. On x86 machines, the registers `CR0` and `CR4` of particular interest, as these hold the configuration of options the guest operating system is currently using, such as paging mode, security extensions and cache management. These configurations are essential to be tracked during the run-time execution of the guest OS, as any modification to these registers significantly alters the behavior of the guest OS.

Similarly, the register `CR3` is of particular interest, as when paging is enabled (as defined by `CR0` bit 31), this register holds the physical address of the page-table for the currently executing process. This is essential for the hardware to be able to perform the

virtual-to-physical (V2P) memory translation on behalf of the process. Thus, by being able to monitor changes of the `CR3` register, we immediately gain valuable information about the scheduling that takes places inside the guest OS. Monitoring of this register was employed exactly for this reason by AntFarm [60].

Additionally informative traps are the occurrence of an *exception*. Exceptions include a variety of events, including *faults, traps* and *aborts* performed by the CPU while in VM mode. In order to avoid trapping all exceptions, Intel provides a mechanism via a bitmap in the VMCS to select which exception causes a VM exit. If the bit corresponding to the exception is 0, the exception is delivered normally via the guest Interrupt Descriptor Table (IDT). For our purposes of particular importance is the treatment of the exceptions caused by the `INT3` instruction, which can also be configured via this mechanism to cause VM exits. Exceptions which may be additionally defined for trapping include *debug exceptions*, *page faults* and *general protection faults*.

4.2.3 Two-stage paging

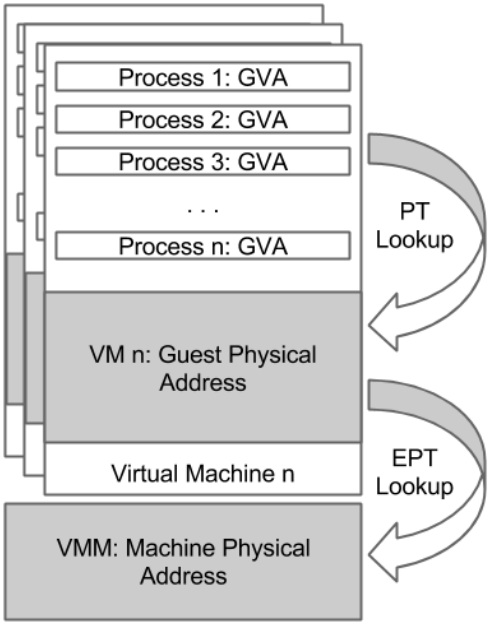

Figure 21: EPT Overview

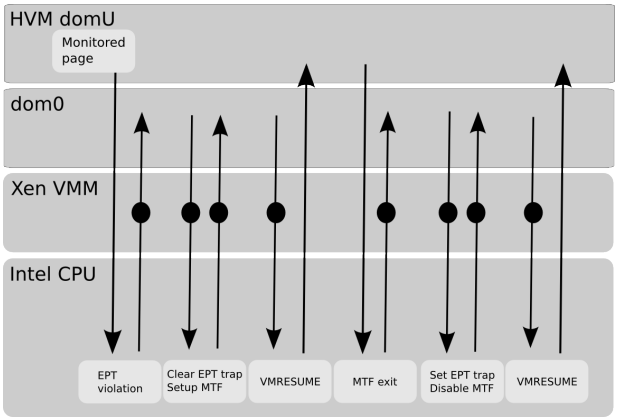

Figure 22: Handling an EPT violation on Xen.

Historically, a VM was required to perform a software based address translation from Guest Virtual Address (GVA) to Guest Physical Address (GPA). The hypervisor performed this action through the shadow page table and it had a significant performance cost. In response, Intel introduced EPT, a new virtualization extension. This extension rectified the situation by providing hardware assisted address translations at both stages, as shown in Figure 21. With EPT, the VM no longer needed to invoke the hypervisor to perform page table operations. This provided a boost in performance by alleviating the necessity to perform a VMEXIT when doing an address translation and by freeing the hypervisor from having to maintain the shadow page table.

Security software running outside of the VM has a long history of using the second stage translation to trigger traps for *active* monitoring. This technique was first used by Ether [24] to trace system calls through modifying the access permissions in the shadow page tables. In newer systems, such as CXPinspector [128], the EPT itself has been used for this purpose because violations in the second stage translation traps into the hypervisor. Combined with other CPU extensions, such as the eXecute-Never (NX) bit, EPT allows for tracing arbitrary memory R/W/X operations. Furthermore, this tracing remains invisible to the guest, as the second-stage pagetables are managed by the hypervisor and the violations are delivered to the hypervisor directly by the hardware.

During an EPT violation, the VMCS further describes the location of the violation, both as a GPA and as a GVA. Additionally, the VMCS also describes if the violation occurred during a first-stage GVA translation (violation during the CPU's first-stage page-table lookup) or with the final GPA obtained from the translation.

While tracing memory accesses with EPT is stealthy, the performance overhead is considerable. Each violation on a monitored page needs to be first cleared and then reset to allow the VM to continue the execution but to still catch all subsequent events. On Figure 22 we show the common handling of EPT violations on Xen. In case only a certain section of a page is of interest, the tracer also needs to filter unrelated violations, further

adding to the performance overhead and complexity of handling EPT violations.

4.3 System design

In our prototype system, named DRAKVUF, we implemented a set of data collection mechanisms for 32-bit and 64-bit versions of Windows 7 SP1. The system expanded upon our prior malware collection prototype to maximize the number of sessions that can concurrently run on the hardware via copy-on-write disk and memory. However, the monitoring system has been completely replaced by a VMM-event based tracking, thus eliminating the need for scanning the memory for data-structures of interest.

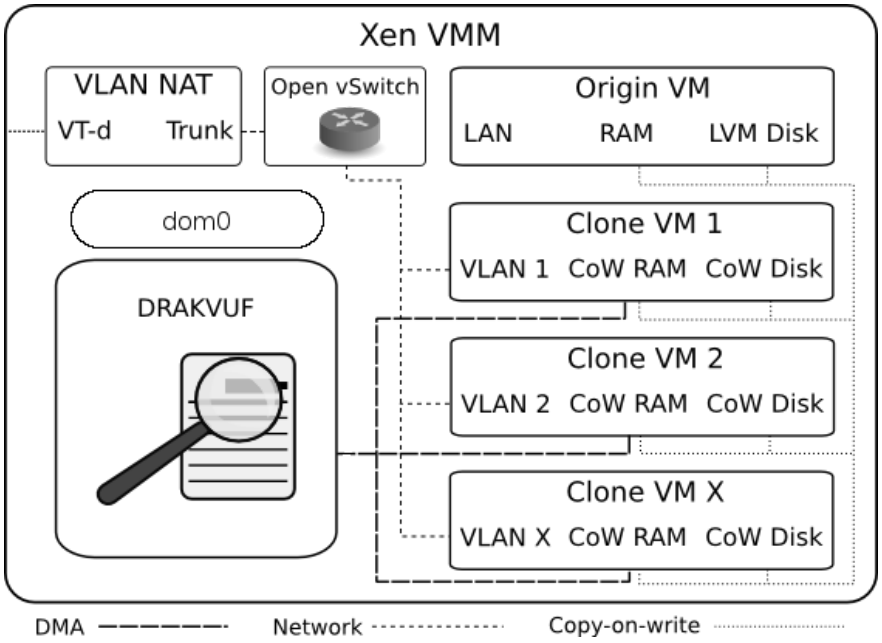

Figure 23: System overview of DRAKVUF

Our prototype named *DRAKVUF* is built on the open-source Xen Project Hypervisor. The high-level organization of system components is illustrated in Figure 23. To allow fast deployment of analysis VMs, *DRAKVUF* creates full VM clones via Xen's native copy-on-write (CoW) memory interface and the Linux logical volume manager's (LVM) copy-on-write (CoW) disk capability. While the static components of the analysis VMs' memory and disk are shared, the use of copy-on-write prevents clone VMs from interacting with

each other as they don't have access to exclusive resources allocated to other clones.

DRAKVUF runs in a secondary control domain and makes use of direct memory access (DMA) through the LibVMI library. Within this secondary control domain *DRAKVUF* also has access to hypervisor features to control virtualization extensions provided by the CPU, such as the Extended Page Tables (EPT). However, the hypervisor restricts to ability of the *DRAKVUF* domain to interact and affect domains specifically labeled as VMs assigned for analysis using the native Xen Security Modules (XSM) [21]. In order to facilitate access to events associated with the execution of the analysis VMs, *DRAKVUF* uses a combination of techniques to trigger a transfer of control to the hypervisor (VMEXIT) when required.

The core technique we employ is the use of *breakpoint injection* in which a #BP instruction (`INT3`, instruction opcode 0xCC) is written into the VM's memory at code locations deemed of interest. The breakpoints are further protected by EPT permissions so their presence cannot be discovered by code running within the guest. By configuring the CPU to issue a VMEXIT when breakpoints are executed and configuring Xen to forward such events to the control domain, *DRAKVUF* is capable of trapping the execution of any code within the analysis VM. In *DRAKVUF* we are the first to apply the technique for automatic execution tracing of the entire OS and demonstrate how it is a key component in enabling *active* VMI. With this technique *DRAKVUF* gains deep insight into both kernel and user-land code execution. As the system's integrity is assumed to be valid before a malware sample is executed, we take advantage of using standard in-guest data-structures to map out the system's internal layout, thus bridging the *weak semantic gap problem*. Once the malware sample is started, we tackle the *strong semantic-gap problem* by dynamically trapping kernel heap allocations and the kernel's internal functions.

In *DRAKVUF* we address the previously overlooked problem of starting the execution of a malware sample without leaving a trace: thus far the task had to be either performed manually, which hinders scalability [23, 24]; or with the use of in-guest agents that can

be detected [14]. *DRAKVUF* addresses this shortcoming by enabling the automatic execution of a sample without the use of in-guest agents, as the presence of such agents could be used to detect the monitoring environment. Instead *DRAKVUF* uses *active* VMI via #BP injection to hijack an arbitrary process within the VM to initiate the start of the malware sample, a technique further described in Section 4.4. By using existing processes running within the VM, *DRAKVUF* does not introduce new code or artifacts into the analysis VM, thus greatly improving stealthiness.

As malware is known to use external input and resources to function, providing network access to the analysis VMs is also required. In order to maintain isolation between *DRAKVUF* and the analysis VMs, network traffic passes through a domain running Open vSwitch and exits through a VLAN NAT domain containing a physical network card passed through using Intel VT-d. The clones are placed on separate VLANs, therefore the only network access they have is through the NAT engine which actively prevents the analysis VMs from discovering each other over the local network. This setup is aimed at minimizing the number of components within dom0 that expose an interface to the infected clones, as we consider emulated device backends to be a more likely attack surface as compared to the minimal interface exposed by the VMM.

To further minimize the exposure of critical system resources to malware which may hypothetically use *DRAKVUF* itself to stage a break-out attack, we further isolated *DRAKVUF* to a deprivileged secondary control domain. Using XSM a security policy has been created and enforced which limits the access of the *DRAKVUF* domain to interact only with domains used for the analysis sessions, effectively removing *DRAKVUF* from the TCB of the system. The only communication channel *DRAKVUF* retains with the TCB is a single-command interface with dom0 through XenStore to request the deployment of a new analysis VM. This step further strengthens the system's isolation .

4.4 Stealth

The stealth of virtualization based analysis systems has been commonly considered in the context of the detection of the monitoring environment itself and not the detection of virtualization in general [24]. The argument for this division is that the already wide-scale deployment of virtualization in commodity systems creates an economic incentive for malware not to exclude virtualized systems. With these assumptions in place, *DRAKVUF* expands upon prior research that showed that the combined use of breakpoint injection with EPT protection is an effective technique to hide from even the most advanced anti-debugging techniques [23]. Time skew introduced by trapping the guest into the hypervisor has also been effectively countered in prior research by altering the guest's access to hardware time sources such as the `TSC`, although external time sources could still be available to the malware if network access is allowed.

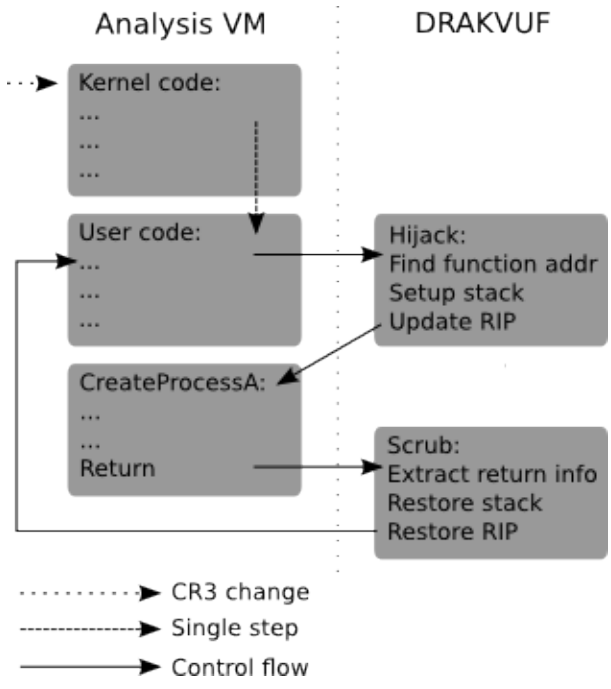

Figure 24: Initiating the execution of malware samples without in-guest agents

In *DRAKVUF*, we have turned our attention to a critical step so far overlooked in automated dynamic malware analysis: we start the execution of the malware sample without

leaving an identifiable trace of the monitoring environment. With systems where an in-guest component is used, the execution can be initiated by the monitoring agent itself, but the same in-guest component could be potentially used to detect monitoring, even if it is only an auto-start script. On the other hand, when no in-guest agent is present, the sample has to be started manually. Therefore, in order to avoid creating any artifacts within the analysis VMs but to allow automated execution, we implemented an injection mechanism that hijacks an actively running but arbitrary process within the VM to initiate the start of the sample on our behalf.

```
BOOL  WINAPI      CreateProcessA(
        _In_opt_  LPCTSTR                  lpApplicationName,
        _Inout_opt_ LPTSTR                 lpCommandLine,
        _In_opt_  LPSECURITY_ATTRIBUTES    lpProcessAttributes,
        _In_opt_  LPSECURITY_ATTRIBUTES    lpThreadAttributes,
        _In_      BOOL                     bInheritHandles,
        _In_      DWORD                    dwCreationFlags,
        _In_opt_  LPVOID                   lpEnvironment,
        _In_opt_  LPCTSTR                  lpCurrentDirectory,
        _In_      LPSTARTUPINFO            lpStartupInfo,
        _Out_     LPPROCESS_INFORMATION    lpProcessInformation
);
```

Table 4: Function prototype of the CreateProcessA function

On Windows, a new process can be created by any user-space application via the CreateProcessA function (shown in Table 4), which is part of the standard Windows API exposed by the kernel32.dll library [82]. While not every application on Windows has kernel32.dll loaded, generally only a view system processes are the exception, thus in practice *DRAKVUF* can hijack any normal application.

The injection mechanism relies on a set of events, shown in Figure 24, to successfully hijack a process without causing system instability or altering the state of the machine in a way that would reveal the monitoring environment. As the first step after the clone analysis VM is created, *DRAKVUF* traps write events that happen to the control register CR3 to catch when a process context switch occurs. When an event is caught, we

examine what libraries are loaded in the address space of the now running process by walking the list of loaded modules within the process. If the process has kernel32.dll loaded into its address space, the execution of the VM is switched into single-step mode until the process starts executing user-level code (`CPL3`). This is required since after the context-switch the process is still executing in kernel-mode and calling any API mapped into the user-space (paged memory) of the process would cause of "Bug Check 0xA: IRQL_NOT_LESS_OR_EQUAL" error.

While the singlestep approach has been sufficient in most cases, in later revisions of the prototype we have evaluated additional mechanisms to detect when execution returns to user-mode. This further exploration was prompted by the overhead that singlestepping presented, as it often lead to the process never reaching user-mode before being scheduled out again by OS scheduler. Instead, a more reliable method has been by finding the *trap frame* on the executing user stack. The *trap frame* is a critical system structure that the operating system prepares when a process is being scheduled out, which allows the OS to resume the process at a later time. Most importantly, the *trap frame* contains the address of the first instruction that should be executed once the process is switched to user-mode. This presents an ideal method to detect when the process has returned to `CPL3`.

Another possibility for trapping return to usermode could also be accomplished by the injection of Asynchronous Procedure Call (APC) [105] into the target process when it is scheduled out. An APC on Windows is a linked-list that contains elements describing the necessary functions the process should call before returning to `CPL3`. The APC could define both kernel-mode calls as well as user-mode calls. The APC calls can only be of type `APCProc`, thus we can't inject arbitrary calls via this mechanism. For example, the CreateProcess functions take multiple inputs (as shown in Table 4, while `APCProc` functions take a single pointer to their input, thus we can't use this method for process injection directly. Nevertheless, it is still sufficient for *dll injection*, as the LoadLibrary

functions share the same prototype. We can also inject a call this way to an arbitrary location that we have already breakpointed, thus detecting when the process' APC queue is being processed in user-mode from the hypervisor level.

The hijack mechanism takes over the execution at the first instruction executed in CPL3, and locates the CreateProcessA routine in kernel32.dll's export table. First, to locate kernel32.dll, the process' loaded library list has to be walked to find the desired library. This is done by walking the linked list that starts at the location defined within the Process Environment Block (PEB) of the process. The PEB points to a structure, _PEB_LDR_DATA, which holds the linked list of loaded libraries. In fact, there are three copies of the list, each ordered differently for optimized searches: in load order, in memory order, and in initialization order. For our purposes the load order list is preferred, as kernel32.dll is a critical system library usually being loaded very early during each process' initialization. The elements in the list are of type LDR_DATA, with the library names stored in Unicode format. During enumeration we convert the names to UTF-8 first, and once the required library - kernel32.dll - is found, we obtain the library's base address from the the field DllBase.

Once the library base address is found, we continue by locating the *export table*. The export table is part of the Portable Executable (PE) header, at an offset defined in the data directory array at index 0, corresponding to the IMAGE_DIRECTORY_ENTRY_EXPORT definition. The export table on Windows is used to list all function entry points that the library makes available. The export table structure itself begins with a reserved 32-bit field which must always be set to zero. On Windows 7 the export table is occasionally mapped on page-boundaries, such that the first reserved field is not accessible, as it is mapped onto a page that is not in the process' virtual memory. The table lists available functions both by name and by *ordinal* to speed up function lookup. When a process is loaded, the loader looks up all functions it uses from a library based on either the name or the ordinal and maps the address of the function into the process' *import table* to avoid further

lookups during run-time. However, using the import table is optional and a program can use any function that is exported by a loaded library.

Once the function's virtual address is determined by parsing the export table of kernel32.dll, we are tasked with imitating the results of what an actual call to the library function would look like. Calling conventions differ depending on the compiler, and further differ depending on the underlying architecture. To better understand this process, we need to look at the assembly instructions of a simple binary that executes this function (shown in Listing 2) to determine the calling convention used by the Microsoft compiler (assembly for x86 is shown in Listing 3 and for x86-64 in Listing 4). As can be seen, on x86 all arguments as passed via the stack, in opposite order. That is, the pointer to the `PROCESS_INFORMATION` structure is pushed first on the stack. Furthermore, no differences between the type definitions are present between 0, `NULL` and `FALSE`. It is also important to note the 4-byte alignment of the variables on the stack, irrespective of the underlying type. This alignment is particularly important for pushing the command line string argument on the stack, which needs to be 4-byte aligned as well. In case the string's length is not 4-byte aligned (with the NULL-termination included), it is necessary to pad the remaining space to make sure the string's start is aligned.

On x86-64 we can see a shift in that only a subset of the function input parameters are passed via the stack, the first four arguments are passed via the `RCX`, `RDX`, `R8` and `R9` registers. However, the first input parameter passed on the stack is located +20h bytes above the stack pointer (`RSP`). The space, corresponding to the first four arguments, is still reserved - even if the function takes less then four parameters - and is called a *parameter homing space*. This space is used "if either the function accesses the parameters by address instead of by value or if the function is compiled with the `/homeparams` flag" [98]. The `/homeparams` flag is only available on checked builds and results in all values passed via registers also being saved on the stack.

In our hijack mechanism, based on the information gathered from the disassembled

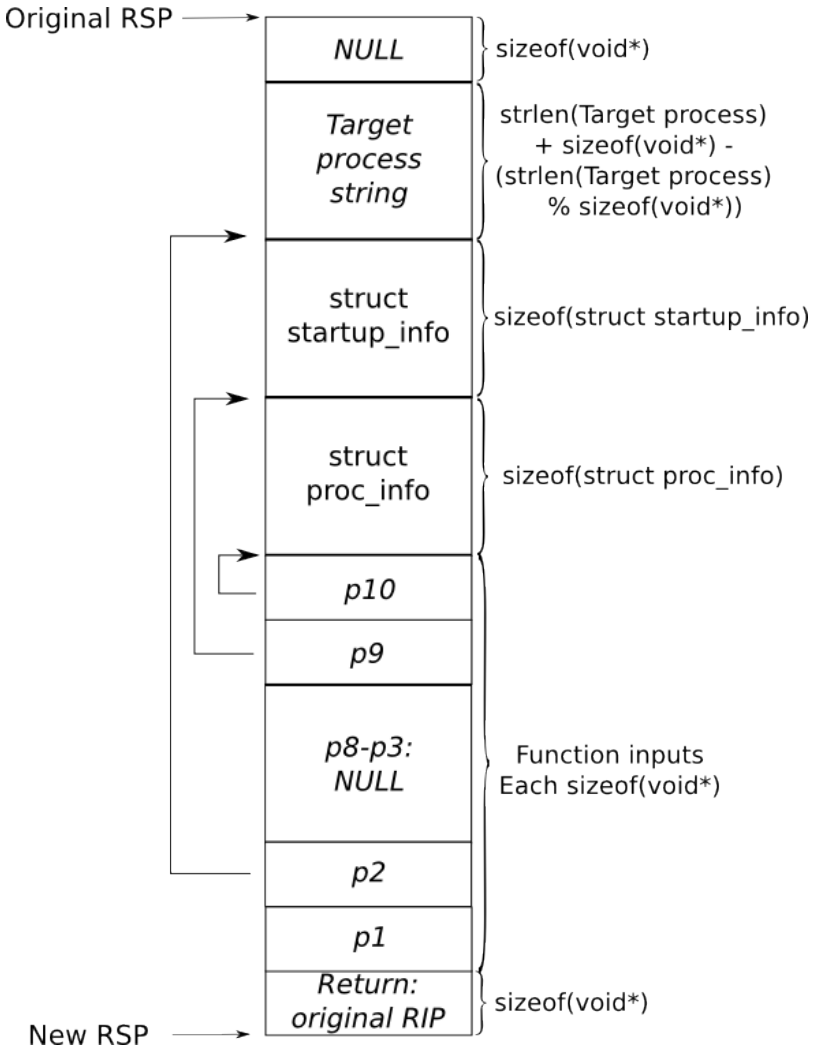

Figure 25: Setup of the stack for function call injection of kernel32.dll!CreateProcessA on a 32-bit Windows 7. The setup is similar for 64-bit Windows, where p1-p4 are passed on registers instead of the stack.

binaries, we can sketch out how the stack needs to look like, shown on Figure 25. While the disassembled binaries all end with a `call` instruction, we further need to highlight what effect this instruction has on the stack. In fact, a call instruction can be also replaced by a pair of `PUSH address_after_call; JMP operand` instructions [133]. The `JMP operand` instruction can be further thought of as `mov RIP, operand`. Thus, after the input parameters are placed on the stack, the final value placed on the stack is the instruction where the execution should return after the callee finishes. Here, we place the address of the original instruction that should have executed (the current address in `RIP`),

and we further write a breakpoint into this location.

When the return breakpoint is hit, we can determine if the process has been successfully created by examining the RAX register, and if it was successful, we also obtain the PID and the handle information of the process that will be used by the executing malware sample. As a context switch could occur while CreateProcessA is executing, the return trap checks if the process at the return trap is the one that was hijacked. Before resuming the original execution of the hijacked process, the stack and vCPU registers are restored, thus seamlessly resuming the execution of the process.

In our implementation we use this mechanism to start malware samples in clean virtual machines. By using this hijacking routine, no artifacts are left on the system that could be detected as a fingerprint of the monitoring environment. This property may not hold if a malware sample is started in an already infected machine, a use-case that we considered out-of-scope for our implementation.

Furthermore, any process running within the system can be hijacked to initiate the execution of the malware sample as long as kernel32.dll is loaded into its memory space, which in practice constitutes the majority of processes on Windows. However, the sample of the malware has to be present on the filesystem of the analysis VM for the injection to work. As the clone VM inherits the execution state of the origin VM, simply placing the sample on the analysis VM's CoW disk is insufficient, as the in-memory filesystem information does not reflect the presence of the new file. In our prototype system the samples have to be placed on the disk of the origin VM before clones are created as to later allow the clone VMs to access them. In the future the samples could be loaded into memory directly by either employing process hollowing techniques through VMI [72] or by hijacking the control flow of the OS when the file would be loaded from the filesystem, further improving the usability of the system.

4.5 Execution tracing

A key feature of existing dynamic malware analysis systems is the ability to trace the execution of processes by monitoring system calls. However, monitoring *only* system calls limits the execution trace to the interaction between user-space programs and the kernel, thus excluding the execution of kernel-mode rootkits. To overcome this issue, in *DRAKVUF* we took an alternative approach by directly trapping *internal* kernel functions with a technique known as *breakpoint injection*.

On modern processors the breakpoint instruction (hex 0xCC) can be configured to trap to the hypervisor directly. Thus, by writing this instrusion into select code location we can induce a trap directly at points in the execution we are interested in. With direct trapping *DRAKVUF* is able to monitor malicious drivers, as well as rootkits, which was previously not possible with just system call interception. Since these instructions are placed into the guests' memory, extra precaution needs to be taken to prevent the guest from discovering these breakpoints. This is fortunately also achiavable by further marking the pages where these breakpoints are placed execute only in the EPT. This way, any attempt by the guest to read the code where the breakpoint is placed is first trapped to the hypervisor and thus the memory can be reverted as to present the original content of the memory location.

The location of the kernel functions are determined by extracting information from the debug data provided for the kernel. The use of debugging information has been an established method in the forensics community and it is the most convenient avenue to gain insight into the state of the operating system. In *DRAKVUF* we make use of the Rekall forensics tool [94] to parse the debug data provided by Microsoft to establish a map of internal kernel functions.

At run-time, *DRAKVUF* locates the kernel automatically in memory without having to perform signature based scans, which improves resiliency as compared to existing forensics tools [85,119]. To automatically locate the kernel in memory we make the observation

that Windows 7 uses the FS and GS registers to store a kernel virtual address pointing to the _KPCR structure, which is always loaded into a fixed relative virtual address (RVA) within the kernel, identified by the KiInitialPCR symbol. As we have obtained the RVA for all kernel symbols, including KiInitialPCR, we only have to subtract the known RVA of the symbol from the address found in the vCPU register to obtain the kernel base address.

Once the kernel base address is established, *DRAKVUF* can trap all kernel functions via #BP injection. With internal kernel functions being trapped, the logs thus generated provide a full trace of the execution of the OS from the moment the malware sample is executed.

4.5.1 Tackling DKOM attacks

Rootkits notoriously modify internal kernel structures to hide their presence on a system, commonly referred to as DKOM [17]. Standard DKOM attacks are performed by unhooking structures from kernel linked-lists (like the running process' list), which effectively prevents tools that use these lists to enumerate the structures from discovering the additional elements. Forensics tools have long discovered that objects within the Windows kernel heap are created with an additional header attached (_POOL_HEADER). This header contains a four-character description of the structure that can be used to detect unhooked structures by simply performing a brute-force string search for these tags in physical memory [16, 102, 121].

As pooltag scanning became a standard approach in forensics, malware is known to attempt to overwrite the header to prevent scanning tools from later finding these structures [31]. Other rootkit techniques hide the structures by disconnecting the allocated object from its header by changing the requested object size to be greater than 4096 KB, as such allocation requests result in the object being placed into the *big page pool* where no such header is attached to the object. This technique effectively prevents basic pool

tag scanning routines from fingerprinting the object.

However, having access to these internal kernel structures is critical in understanding the runtime state of the system, therefore in *DRAKVUF* we took a new approach to tackle DKOM attacks. As we see from the description of the attack methods, the root cause of the problem with DKOM attacks is that the location of the structures becomes unknown within the kernel's heap. If the location can be accurately determined, DKOM is effectively defeated [95]. With *DRAKVUF* we track the kernel heap allocations directly with #BP injection at internal Windows kernel functions responsible for allocating memory for structures used by Windows: `ExAllocatePoolWithTag` and `ObCreateObject`. *DRAKVUF* dynamically extracts the return address from the stack of the calling thread at function entry and traps it to catch the event when the allocation routine returns. Monitoring heap allocations allows us to detect the location of all kernel structures without malware being able to tamper with our view into the system.

In the current implementation *DRAKVUF* tracks the allocation of all objects on the kernel heap, and generates logs based on the associated tag of the structure. If the tag of the structure is one of the already known 2,254 tags, the log contains further details about the type of the object to aid the analyst in identifying allocations that may be of further interest. To detect for example a hidden process, an analyst can now apply a cross-view check to determine if the allocated structures are also accessible via standard lists [59]. *DRAKVUF* further traps the routines responsible for freeing these structures, thus providing a full-view into the life-cycle of the structures. In the next section we further illustrate how this approach enables us to track the active usage of `_FILE_OBJECT`s.

4.5.2 Monitoring filesystem accesses with memory events

Monitoring filesystem accesses is one of the core feature of any dynamic malware analysis system, however, prior agentless VMI approaches have attempted to monitor filesystem accesses by modifying the disk emulator to intercept events [88]. While such an

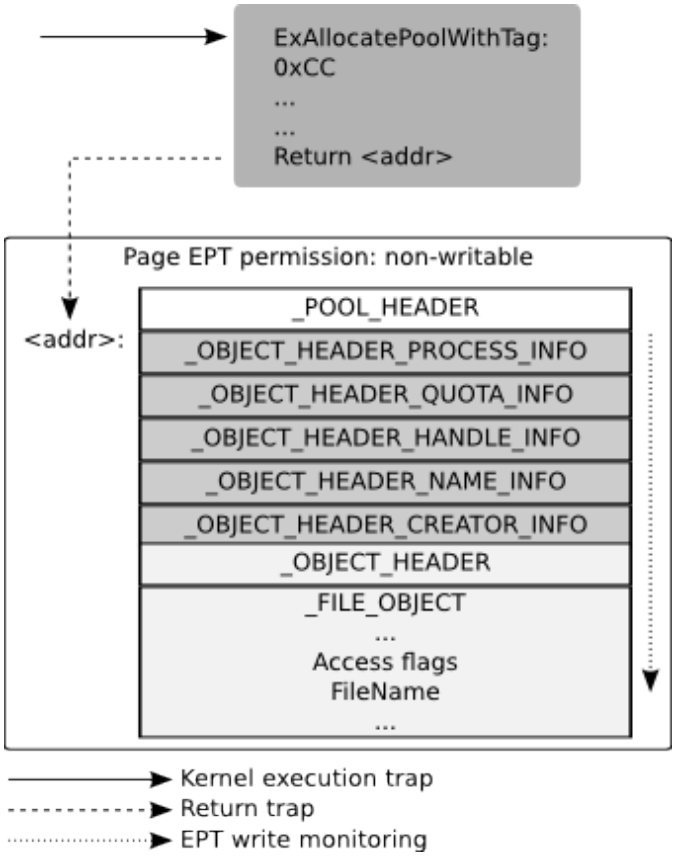

Figure 26: Tracking file accesses by monitoring the allocation of `_FILE_OBJECT`s in the Windows kernel heap.

approach is effective, reconstructing high-level file-system accesses (like path and permissions) from the low-level disk-emulation perspective is in itself a form of the semantic gap problem and requires extensive knowledge of file-system internals. However, the internal kernel structures that *DRAKVUF* tracks reveal highly valuable information about the execution state of the system, such as the complete set of running processes, kernel modules, threads, and even objects allocated for filesystem accesses by the OS.

The process by which we catch filesystem accesses is shown in Figure 26. When a file is accessed, either by the OS or by a user-land process, a `_FILE_OBJECT` is allocated within the kernel heap with the accompanying tag, `"Fil\xe5"`. When the allocation address is caught, we mark the page on which the structure is allocated as non-writable in the EPT. As the `_FILE_OBJECT` is preceded by a set of optional object headers (shown

with a gray background), we derive the exact location of the access flags and file name by subtracting the known size of the _FILE_OBJECT from the end of the heap allocation. This allows us to determine the full path of the file as well as the access privilege with which the file is accessed, such as read, write and/or delete permission, without the need to have any deeper understanding of the filesystem itself.

As the pool tracking mechanism traps the call to the entry of the allocation routine, in order to determine where the space is allocated we inject an additional trap into the return address of the function that is pushed on the stack. When the trap at the return address is hit, the kernel virtual address where the object has been allocated can be read from the vCPU register RAX. However, it is possible that a context switch happens before the allocation routine returns, which in turn can also call the allocation routine, potentially resulting in a trap at the same return address having multiple objects pending allocation. Therefore *DRAKVUF* keeps track of the allocations based on the CR3 value, which identifies the process, and the RSP, which identifies the thread of the caller, to avoid confusion between pending object allocations.

4.5.3 Carving deleted files from memory

A common feature of malware droppers is the rapid creation and deletion of temporary files used during the infection process [10]. These temporary files can potentially contain the unpacked malware binary before it is loaded into memory, or other forensically relevant information. However, malware authors are well aware of this fact and it is standard procedure to clean up the temporary files after the dropper finishes installing the malware.

Existing malware analysis systems implemented with the use of in-guest agents can simply retrieve these files when file-deletion is initiated. From a VMI perspective, retrieving these temporary files is complicated by the fact that Windows defaults to write caching being enabled on all hard-drives. When files are created and destroyed rapidly, as is often the case when malware is being dropped on a system, the files are never written to

disk. As a result simply mounting the analysis VM's disk in the control domain would not give access to these files and the only possible scenario is to carve the files directly from memory.

In *DRAKVUF*, the carving of deleted files is implemented by intercepting specific internal kernel calls that are responsible for file deletion, such as the `NtSetInformationFile` and `ZwSetInformationFile` routines. Once the functions are intercepted, the file is identified by examining the arguments passed, among them the file handle information. This handle does not point directly to the file object, it is only a reference number to an entry in the handle table of the owning process. By parsing the handle table of the process we can locate the corresponding `_FILE_OBJECT` and automatically carve it from memory with Volatility [119]. This enabled us to capture transient files during the execution of the malware sample, resulting in the capture of previously unknown binaries with virtually no AntiVirus detections, which we will further discuss in Chapter 4.6.

4.6 Experimental results

In the following, we discuss an extensive set of experiments performed to establish performance metrics and to evaluate the effectiveness and throughput of the systems. The experiments were performed on an Intel Second Generation i7-2600 CPU running at 3.4GHz.

Unless specified otherwise, the samples were executed on a Windows 7 SP1 x64 analysis platform with a run-time of 60 seconds. During these tests *DRAKVUF* was set to monitor the execution of each internal kernel function that starts with Nt or Zw. The functions starting with Nt are the functions available to user-space applications through regular system calls as listed in the System Service Dispatch Table (SSDT) and monitoring the Zw version of these functions reveals the execution of kernel-level code. We also trap two additional kernel internal-functions, `ExAllocatePoolWithTag` and `ObCreateObject`, which are responsible for kernel heap allocations. While monitoring all internal functions

would have been possible during these tests, we reduced the scope of tracing as to reduce the verbosity of the collected data without hindering the insight into the execution of the system.

4.6.1 Rootkits

The first sample we tested was `TDL4` [2]. This sample was chosen since an in-depth technical write-up has been already created by an antivirus company after reverse engineering the sample [49], which provides a contrast to our automatic analysis. The dropper itself had a 45/46 detection ratio on VirusTotal (VT) [115]. After executing the sample in a Windows 7 SP1 x64 analysis VM, we obtained two additional temporary files created by the dropper in the Windows' System32 folder: cryptbase.dll [3] and syssetup.dll [4].

These temporary files were carved from memory as they were created by the dropper and never flushed to disk before deletion. After submitting the files to VT, the detection ratios were reported as 19/50 and 22/50 respectively. Further investigation into the nature of these temporary files, unmentioned in the original analysis report, revealed them as being part of a known method to elevate privileges by circumventing user access control (UAC) on Windows 7 and 8 [63]. After the exploit installed its payload, a system shutdown was initiated, at which point 1.1GB of memory out of the 2GB assigned to the VM remained shared.

We also obtained a sample of the `SpyEye2` banking trojan [5], recently released as part of a report made by Fox-IT Security [40]. The sample had a detection ratio of 26/50 on VirusTotal and after 60 seconds of execution 1.6GB of memory remained shared on the analysis VM. No file deletion requests were caught during our analysis and no malicious file manipulations were made by the injected process. However, after closer examination we can see that immediately after the initial binary executes, *svchost.exe* attempts to access files within the same folder where the initial binary is placed at. Most suspicious of these files is "\Users\MrX\Desktop\spyeye \pc\I1\COLCUBE.BIN", but provided that

the file wasn't actually present on the filesystem, the access fails, and no other modifications of the system are observed. This file path only shows up as being part of a Russian torrent advertising itself as "Test Drive 4 full version (ENG)" on Google, uploaded on 09/16/2012, which fits into the malware's reported activity period of 2012-2013, and therefore presumably was used to propagate the malware sample.

The next sample we analyzed was a recent sample of `Zeus` [6] with a VT detection ratio of 44/50. In our analysis no files were deleted. However, in our logs we see the sample interacting with files in a temporary folder, GoogleUpdate.exe having a VT detection ratio of 43/50, and FlashPlayer.exe which had no VT detection and was identified as a real Adobe installer. A DLL was also located in the temp folder, msimg32.dll[7], with a VT detection ratio 25/47 that has never been submitted before. The DLL was later revealed by our logs to be dropped into "\Windows\System32" by the executing flash player installer. The sample's installation behavior very closely follows that of ZeroAccess [131], also analyzed by HP [97]. After executing the sample for 60 seconds, the analysis VM had 1.4GB memory in shared state.

Another recent sample we analyzed was `CryptoLocker` [8]. The original sample had a detection ratio of 35/48. After executing the sample and checking for file accesses, the unpacked executable is identified: "\Users\MrX\AppData\Roaming\B8B838D254.exe". No deletion was requested on any file during our analysis, therefore after 60 seconds of execution we carved the file from memory with Volatility. The file obtained [9] had a detection ratio of 8/49, but after repeating the experiment with a time-out of 3 minutes, the unpacked executable was also obtained from disk [10] which had a detection ratio of 35/48. After 60 seconds of execution 1.3GB of memory remained shared, and after 3 minutes, 955Mbyte did.

Out of the sample set, one deleted executable had no AV detection[11]. The executable is only 4.5Kbyte[12], which appears to be consistent with reports on the malware family the original sample was classified as, albeit still being smaller than the reported size of

20Kbyte [67]. From our execution logs we were able to determine that first the sample loads a set of DLL's from Windows, then tries to locate "C\WINDOWS\System32-\TENSAFE.SYS" for deletion. However, the path is invalid: note the missing semi-colon after C and the fact that Windows keeps track of devices separate from the file path string in the `_OBJECT_HEADER_NAME_INFO` structure. No other outstanding events were observed however during the execution of the sample.

As in the first run only deleted files were extracted from the analysis VM, we repeated the execution of the sample such that all files get extracted when their handle is being closed, however, no AV detection was made on any of the files either. Comparing the checksum of the files in the CoW partition to the origin showed no discrepancies.

No AV detections were made on any of the files accessed during the analysis session, except one by AVG on RpcRtRemote.dll: *Win32/Heur*. However, when we repeated the experiment RpcRtRemote.dll had no detection, therefore it is reasonable to assume to be a false positive in the heuristics of AVG. We further verified file integrity on disk after shutting the analysis VM down and comparing the checksums of the CoW partition to the original, and found no discrepancies.

4.6.2 Anti-VM malware samples

In order to test *DRAKVUF* at scale, we obtained 1000 recent malware samples from ShadowServer [103]. The samples were selected with the AlienVault YARA signature that implies the use of anti-virtualization techniques [1]. During these tests paging and the UAC were disabled. The malware samples were placed on the VMs origin disk, thus the malware startup were simply initiated with a direct CreateProcessA injection, as described in Section 4.4.

Out of the 1000 samples, 241 failed to execute via CreateProcessA injection. These executions failed not because the malware shut itself down but because Windows failed to execute the samples. On average, 159,222 breakpoints had been hit in 60 seconds

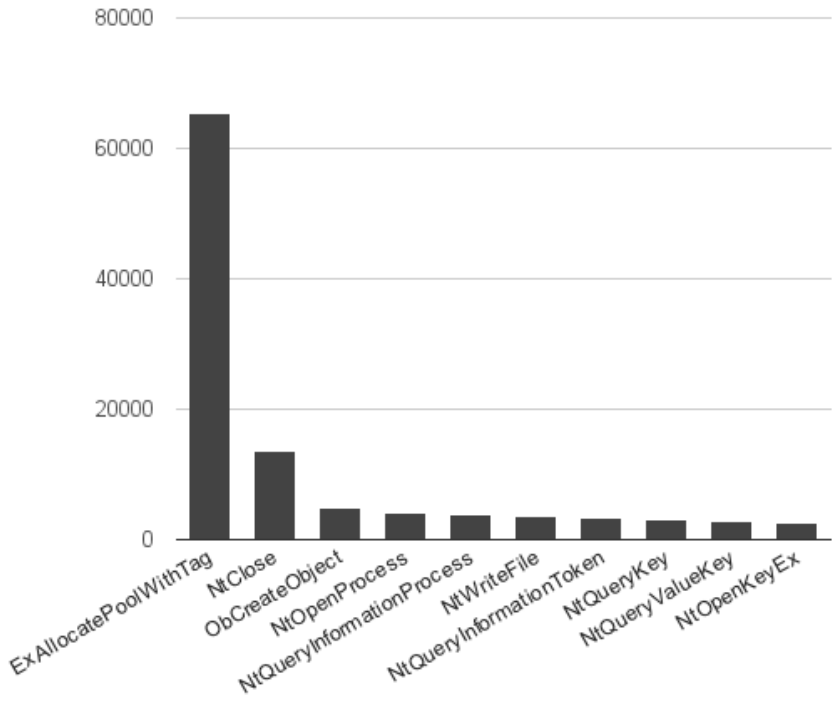

Figure 27: Top 10 monitored kernel functions in terms of average number of observed executions.

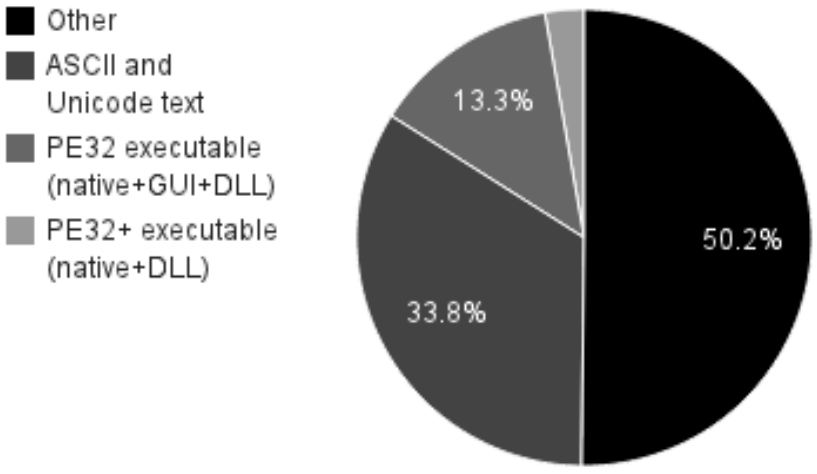

Figure 28: Breakdown of 8797 intercepted file deletion requests by type in recent malware samples.

Figure 29: Number of Read EPT violations versus breakpoints hit within 60 seconds when trapping all internal kernel functions.

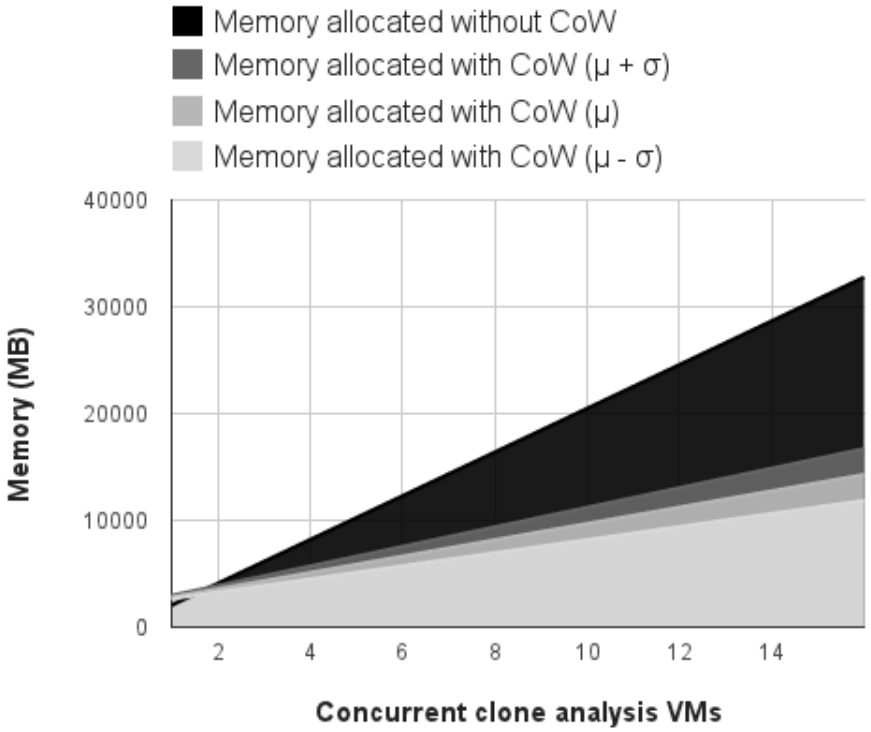

Figure 30: Projected CoW memory allocations (μ = 764MB, σ = 151MB).

of execution, and on average 67,950 were pool allocation requests. Figure 27 illustrates the top 10 API calls that were hit across all samples with the internal functions used for heap allocations among the top three. From this figure we can see that some internal kernel functions are executed significantly more often, thus allowing the analyst to pick and choose those functions that are required for the analysis goal, as we did for these experiments. Only read violations were observed during these tests, an average of 12,182 per session. Our file access tracer recorder an average of 2386 files being accessed, and an average of 12 file deletion requests.

A total of 8797 unique files were extracted from the analysis VMs by carving them from memory before deletion. To better understand the nature of the deleted files in the analysis VMs, we categorized the files by their type, shown in Figure 28. The 1,412 PE files were also submitted to VT. 561 were new submissions (39%), and nearly all of the files had at least one AV detection. On average, only 20.4% of the anti-viruses (AVs) categorized these files as malware. Out of these files, only 2 had no detection with any AV engine. One of these files was not actually malicious but a DLL from the CPU-Z freeware software package, which enables checking the results of CPUID on Windows, presumably used by the malware to detect virtualization.

$:hash:procexp.exe
$:hash:procmon.exe
$:hash:processmonitor.exe
$:hash:wireshark.exe
$:hash:fiddler.exe
$:hash:vmware.exe
$:hash:vmware-authd.exe
$:hash:vmware-hostd.exe
$:hash:vmware-tray.exe
$:hash:vmware-vmx.exe
$:hash:vmnetdhcp.exe
$:hash:vpxclient.exe
$:hash:devenv.exe
$:hash:windbg.exe
$:hash:ollydbg.exe
$:hash:winhex.exe
$:hash:processhacker.exe
$:hash:hiew32.exe
$:hash:vboxtray.exe
$:hash:vboxservice.exe
$:hash:vmwaretray.exe
$:hash:vmwareuser.exe

Table 5: Strings embedded in the temporary files of MultiPlug hint at anti-debugging techniques employed.

A significant portion of the samples in the ShadowServer dump were various versions of the MultiPlug adware. Via *DRAKVUF* a variety of temporary files were extracted for each version before the dropper deleted them, including JavaScript, HTML, JSON and executable

files. To illustrate we take a closer look at one of the samples [13]. This sample created and automatically deleted two DLL's and an executable. While the original sample had a detection ratio of 33/51, the executable [14] had only 5/49 detections, and the 64-bit DLL [15] had 10/49. Interestingly, the 32-bit version of the DLL [16] had a detection ratio of 20/47 and was already submitted to VirusTotal. Checking for strings in the executable immediately hint at the anti-reverse engineering technique of the sample (snippet shown in Table 5), and reveals the developer environment: "C:\Development\extension-setup_2013\bin\Win32\Release\crxdrop_exe.pdb".

Another group of malware in the sample set was variants of Sisbot.A. Here we briefly examine one particular sample of this family [17]. The sample had been only identified as a generic dropper by various AV engines with a detection ratio of 25/50, and after executing the sample with *DRAKVUF*, the unpacked version was obtained from memory before it was deleted [18]. Checking VirusTotal, the unpacked executable had a detection ratio of 21/50.

The dropper also creates a shell script and a Base64 obfuscated Visual Basic script in the temp folder, which were chain executed and then deleted. Further checking file access we can see the malware installing itself into the Windows Startup folder as "svchost..exe" and also attempting to delete a file from the Startup folder, "svchost._backup.exe", which wasn't actually present on the system. These characteristics suggest the sample to be related to the *Troj/MSIL-KW* malware family, analyzed by Sophos [106].

4.6.3 100k+ samples

For the second large-scale experiment we have obtained 114,319 recent malware samples captured in the time-period between January and May, 2015. In order to facilitate automatic analysis we have altered our setup so that the malware samples were not required to be placed on the origin VMs disk before the clones were created. Furthermore, we opted to simplify the setup by removing the VLAN NAT engine. Instead, after the VM

clones were instantiated, as the first step, a configuration command is injected to reconfigure the VMs IP. For this, UAC had to be disabled to allow the command injected via a non-admin process to succeed and avoid the UAC prompt. This has been a temporary measure that can be addressed by injection of user-inputs, as have been successfully shown by Dolan-Gavitt [29].

Once the IP command finished, a 5 second sleep command was performed by pinging localhost, to enable the guest operating system to finalize the network settings. Afterwards, the malware sample to be executed is obtained via TFTP (alternatively, the sample could be obtained via any other mechanism, such as NFS, Windows share, etc). The second step has been identical to our previous experiment, a direct CreateProcessA call is injected through a hijacked task manager process, and once the new process is scheduled to execute, monitoring is started on all kernel internal functions. The MAC address of each clone in this setup remained the same, as the VLAN isolation and IP reconfiguration has been sufficient to avoid collisions.

Out of 114,319 samples, 22,485 failed to execute via direct CreateProcessA injection. This in effect means 80.33% success rate on the execution start. The raw output generated by the system has totaled 775 GB of data. As a portion of that execution log contains all processes executing on the system, we further filtered this data to only lines tagged with the executing malware's unique `CR3` value.

A subset of the samples that started showed no traces of execution with the injected process' `CR3`. We have observed 1,889 samples such samples. A closer examination of this group revealed that 1,294 of these samples started ntvdm.exe, which is the Windows 16-bit Virtual Machine for legacy binaries. No execution has been observed with the process' CR3 however, thus these samples were omitted from our other statistics. The remaining 593 samples have also been abnormal in that the injected process' name didn't match that of the filename of the malware sample on disk, which has been the most commonly observed behavior. These names included innocuous sounding

0q3skeaw.exe	1042585675.jpg	15416841687687	413.exe
757.exe	837.exe	Acadview.exe	Adobe
Acrobat.	ana.exe	bsIjON5.exe	calc.exe
cc.exe	Chome.exe	chorm.exe	chrome.exe
chrom.exe	Claen2.exe	conhost.exe	crypts.exe
dllhost.exe	dw20.exe	dwm.exe	Explore.exe
explorer.exe	flashwr.exe	GNU.exe	google.exe
h.exe	ieplorer.exe	iexplore.exe	IFinst27.exe
IMETIP.EXE	irsetup.exe	lyEptR0.exe	LocalzbVGjJmk_
mmuema.exe	mnbalajs.exe	MRX-PC.exe	mstools.exe
netsh.exe	njw0rm.exe	notepad.exe	Notepad.exe
nsissetup.exe	Pluguin.exe	resort.exe	Resoucpack.exe
Roccketoo.exe	rt.exe	rundll32.exe	RUN.exe
server.exe	Server.exe	shell32.exe	stkGsk9.exe
STMG.exe	svchost.exe	svhost.exe	system32.exe
system.exe	System.exe	Tempserver.exe	tgzdgk.exe
toolbar.exe	Trojan.exe	vbc.exe	vvvvv.exe
WerFault.exe	windows.exe	Windows.exe	

Table 6: Process names with no observed execution of syscalls or heap allocations

names such as `System.exe`, `svchost.exe` and `notepad.exe`. Other process names were more suspicious, such as `Windows.exe` or `google.exe`. We can also see attempts at obfuscation which ultimately fail because spelling is hard, like `wiindows.exe`, `Resoucpack.exe`, `chrom.exe` or `Pluguin.exe`. The honesty in others should also be appreciated, as a process running as `Trojan.exe` or `njw0rm.exe`, would probably raise eyebrows. In total, we have observed 70 distinct process names, with the top three being `netsh.exe`, `svchost.exe` and `server.exe`. A complete list of observed process names for this group can be seen in Table 6.

In the samples that executed normally we have observed an average of 6,288.36 system calls being issued, with standard deviation of 16,809.09 and a skewness of 8.76. On average, 69.17 APIs were used, with a standard deviation of 25.04 and skewness of -0.75. This in effect means that the majority of samples used only about 17% of the trapped system calls. In fact, we have only observed a total of 218 system calls used, thus 46% of the available system calls were never used. This data highlights the diverse

nature of system call utilization employed by modern malware, which may aid the analyst in identifying malware families in the future.

A further breakdown of the data, seen in Figure 31, shows that 17.8% of the observed system calls were each utilized by more then 90% of the samples. Afterwards, the system call utilization sharply declines and we can further observe a sharp drop at 59%. Nearly half of each observed system calls (45.8%) were observed in less then 10% of the samples, with 35% of system calls each used by fewer then 1% of the samples.

Heap allocation requests were performed on average 1899.88 times, with a standard deviation of 4,898.97, skewness of 44.48 and kurtosis 3,310.70. The high standard deviation and skewness to the right indicates a very long tail in the distribution. An overview of the entire distributed can be seen in Figure 32. On average, 87.79 tags were observed per execution, with a standard deviation of 33.89 and skewness of -0.635. In total we have observed 586 unique tags being used. The utilization of the heaptags have shown a more diverse utilization compared to syscall utilization. Only 6.48% of the heaptags have each been used by more then 90% of the samples, while 87.03% of the heaptags were used by less then 50% of the samples. If we further follow the tail of distribution we observe that 72.01% of the heaptags were used by less then 10% of the samples, and 57.5% of all heaptags were observed in fewer then 1% of the samples. Based on this data it is reasonable to assert that highly differential behavior can be observed through the use of kernel heap tags, which could be used for effective malware classification and identification in the future.

In terms of file accesses, we have observed the executing samples interacting on average with 70.17 files and/or folders, with a standard deviation of 93.6 and skewness of 7.99. In terms of total file accesses observed on the entire virtual machine, we have observed 256.05 files on average, with a standard deviation of 246.46 and skewness of 3.73. On average, 0.82 files were deleted with a standard deviation of 1.69 and skewness of 5.41. These deleted files were further captured by carving them from memory and file-

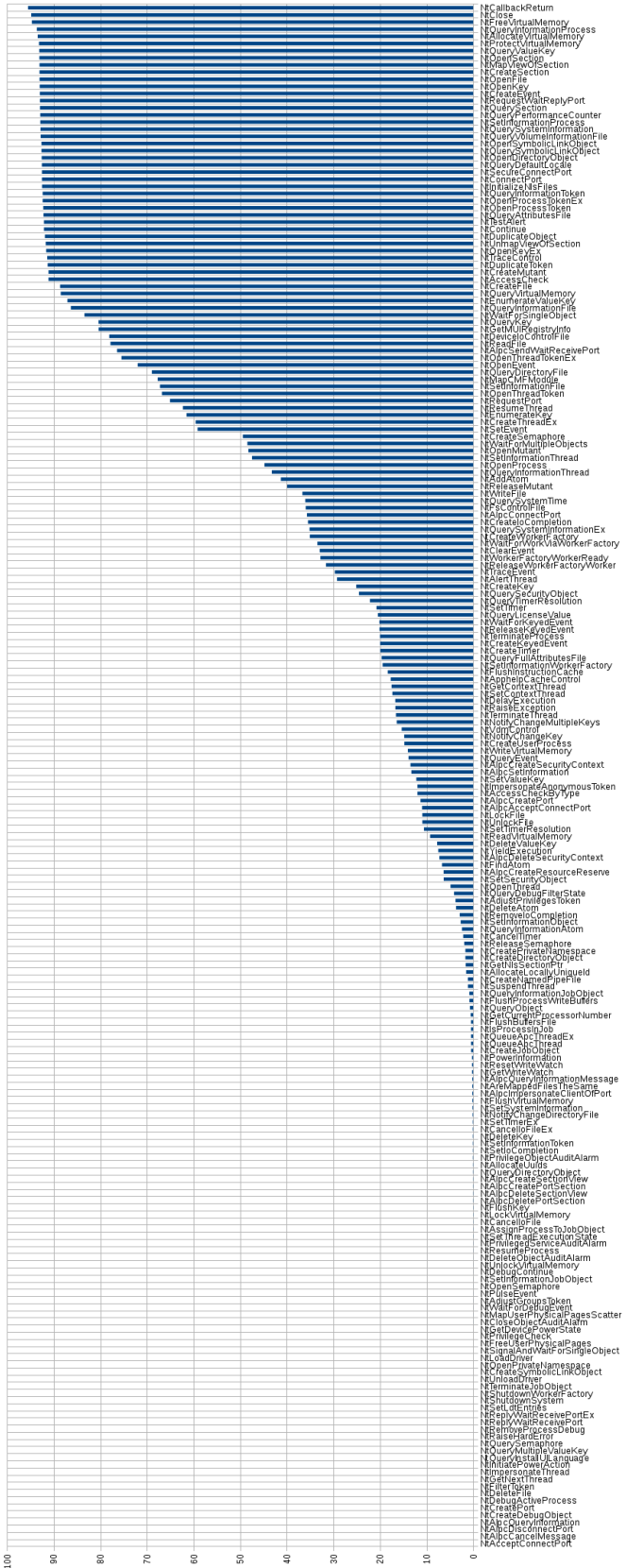

Figure 31: Distribution of observed syscalls used by percent of malware samples

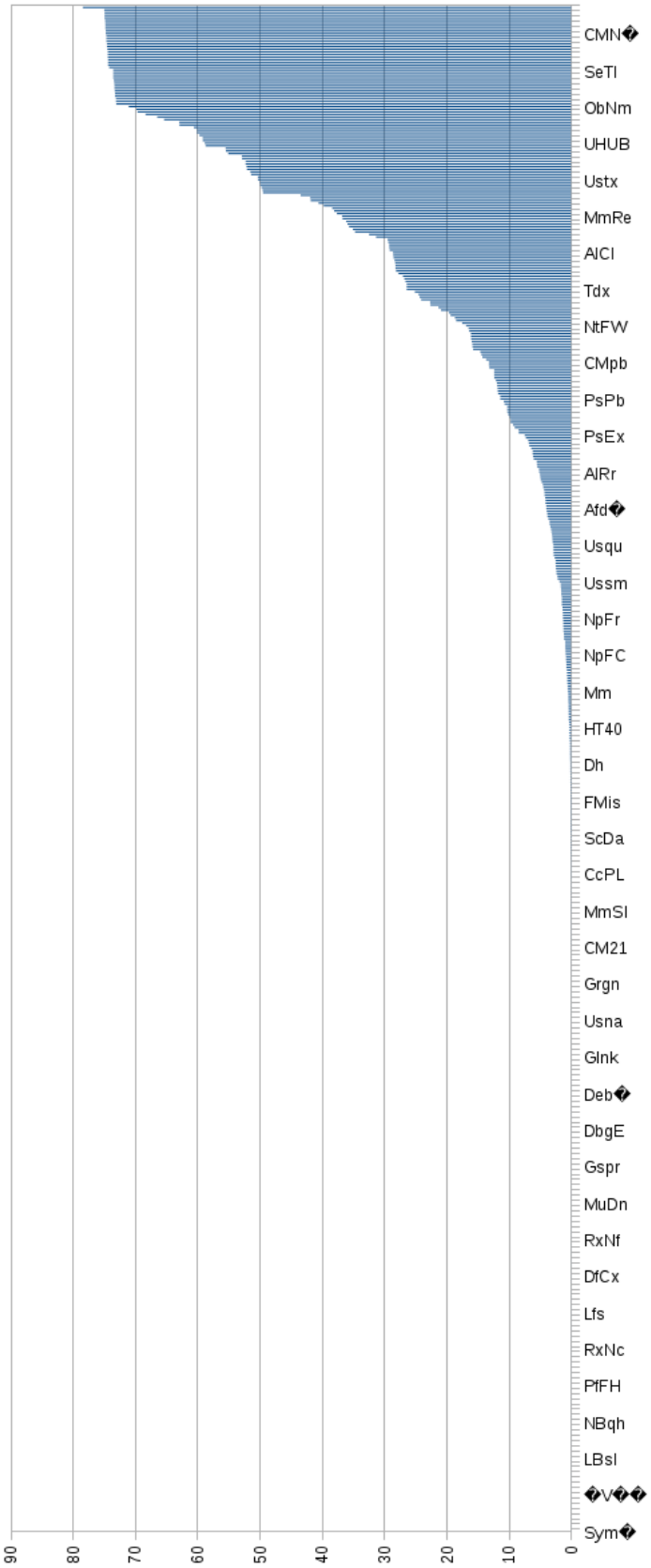

Figure 32: Distribution of observed heap allocations used by percent of malware samples

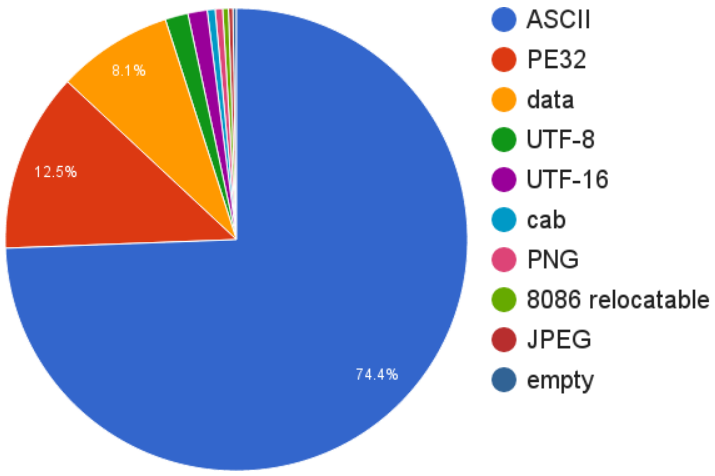

Figure 33: Top 10 most commonly deleted files by type

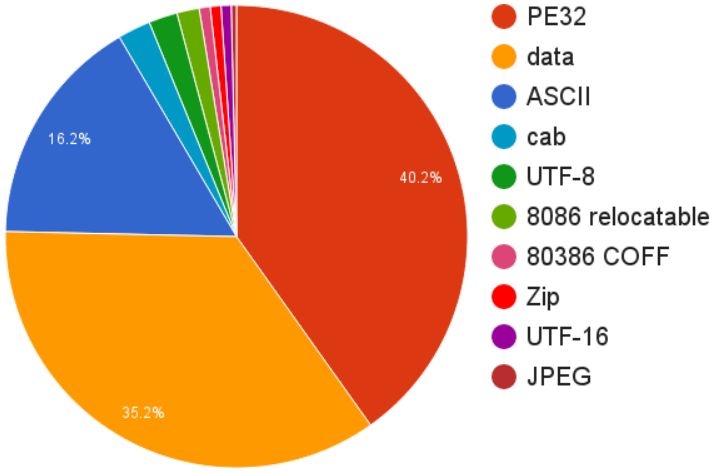

Figure 34: Top 10 most commonly deleted files by type, larger then 10KB

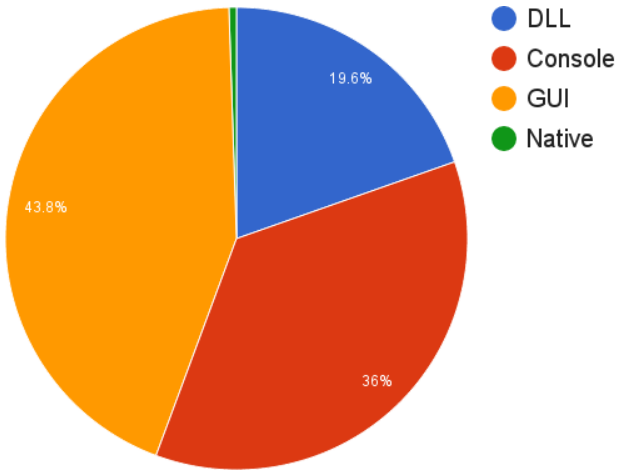

Figure 35: Types of executables dumped

type analysis revealed that 37 different types of files were deleted. The types of the most commonly deleted files are shown in Figure 33. A significant number of the deleted files were of type ASCII and were smaller then 10KB, thus another breakdown is shown for the most common file types where the files were larger then 10KB in Figure 34. We further evaluated the types of PE32 executables that were deleted during the analysis session, with the result seen in Figure 35.

ability-counter.com	accident-muscle.com
airportwake-money.com	ambition-lawyer.com
art-spite-tune.com	assignmentrent.com
attempttune-temperature.com	beachloose-appeal.com
bedwater-spite.com	bicyclereply.com
bite-team-indication.com	black-meet-fat.com
bone-twist-swimming.com	brain-recommend.com

Table 7: DNS record lookups of type NS

Requests	TLD	Requests	TLD	Requests	TLD	Requests	TLD
472	com	82	net	52	ru	47	biz
45	org	32	info	14	it	8	me
8	fr	8	de	7	cn	6	xyz
5	jp	4	br	3	pl	3	es
3	cc	2	kr	2	im	2	co
2	ca	1	website	1	vn	1	us
1	uk	1	tv	1	tt	1	rocks
1	link	1	hk	1	cm	1	club
1	ch	1	am				

Table 8: Top level domain names recorded in the DNS requests

In this experiment the executing malware samples had only restricted network connectivity with no incoming or outgoing connections allowed. The analysis VMs however had been configured with a DNS server IP that was proxying DNS requests via dnschef, so each DNS request was logged. It is possible that some malware may reconfigure this DNS setting thus all DNS logs may not be comprehensive. Not counting standard DNS requests performed by Windows to domains such as `*.microsoft.com`, `*.msftncsi.com`, `*.windows.com` and `*.windowsupdate.com` we have observed

a total of 4795 DNS requests. In total 822 unique records were requested, with 808 being type 'A' record, and 14 were of type 'NS', shown in Table 7. Some of these domains are not yet registered at the time of writing, or were only registered months after the records were collected. In total 34 unique top-level domain names were observed, as can be seen in Table 8, with the full list of domains shown in Table in the Appendix.

4.6.4 Stalling code

A critical limitation of dynamic malware analysis systems is *time*. When a malware binary is run through the system, it is unknown how long it has to be executed before it's malicious behavior is observed. This problem has been reduced to the *halting-problem* [100], thus it is generally believed to be undecidable. Recent work on the topic raised some glimmer of hope that it may not be the case if time and resource constraints are taken into considering [15], for all practical purposes today it is still undecidable.

This presents an easy opportunity for malware to avoid detection with minimal effort, as dynamic analysis systems can't execute the sample indefinitely. Most execute them only for a couple minutes at most. In our experiments we have chosen execution times of 1-2minutes at most. Thus, if the malware can wait out the analysis period before activating, the analysis won't be effective. On the other hand, stalling may also be a risky strategy for malware when infecting live systems, as it provides an opportunity for AntiVirus systems to detect and remove the process before it is able to deploy it's more advanced payloads.

This problem has of coursed been long known by the creators of analysis systems. For example, Kolbitsch *et al.* [65] described various stalling techniques that were targeted against their emulation-based analysis system. In their experiments they observed malware attempting to execute instructions that were known to execute slower under emulation then on real hardware. While they were able to counter some of these techniques by detecting the stalling loops by creating control flow graphs, such an approach is not

viable under full system virtualization as we are not observing every instruction of execution. This type of stalling may not necessarily be applicable to analysis systems relying on hardware virtualization, as instruction can execute natively on the CPU without much overhead. It has also been observed that malware may abuse the logging overhead associated with trapping system calls. For example, malware may choose to rapidly call system calls that normally exit fast (like NTCREATESEMAPHORE, or invalid syscalls), but under monitoring they incur the full overhead of logging. While certainly a valid approach, these methods are increasingly noisy, and thus the risk of detection is higher.

In practice many malware sample resorts to simple environment checks to detect the presence of debuggers or other applications that are usual signs of an analysis platform. We have seen such checks during our tests, as shown in Table 5. Another easy check malware can perform is to check the system uptime before executing, as many dynamic malware analysis platforms require new VMs to be booted form scratch for each analysis session. As our monitor tool runs outside the monitored OS, and can be snapshotted after letting it run for an arbitrary time, such environment checks are easily defeated.

Another behavior observed is malware simply *sleeping* for a while before starting its execution. Provided the first stage payload of the malware that performs the stalling is benign enough, it could effectively bypass AntiVirus systems. To counter such attempts the open-source Cuckoo sandbox for example monitors and alters all calls to `NtDelayExecution` and `NtQuerySystemTime` to skip such sleep attempts.

Based on these finding it is reasonable to formulate the following two hypothesis:

1. Samples performing anti-sandbox behavior via fast syscalls will exhibit a lower utilization of available syscalls as the malware will continuously ”spin” these syscalls to time the sandbox out.

2. Samples that perform stalling by sleeping will exhibit lower overall execution of syscalls as compared to normally executing samples as the malware will be in a sleep state and executing no instructions.

To test the first hypothesis we filtered the data for all samples that have used the NtCreateSemaphore syscall and calculated the average number of API's used by these samples. What we actually observe is that samples using this syscall exhibit an overall higher then average number of syscalls being used: 85.32 vs 69.17. Furthermore, the standard deviation of this group is also lower, 15.3 as compared to the 25.04 observed with all samples included. Thus the simple case of the syscall being used does not validate the hypothesis. We can further filter the sample set by checking whether the samples exhibiting higher then average usage of this syscall match the hypothesis. NtCreateSemaphore on verage has been issued 7.77 times and has been observed in 45,383 sample. The above average sample set is 23,022 samples. In this set, we can observe an even higher average syscall utilization: 91.47 with a standard deviation of 15.48. Thus, in the case of this particular syscall, we can observe no evidence of it being used as sandbox stalling mechanism on a large scale.

If we further check the usage of NtCreateSemaphore by samples, we observe one sample [19] that has used this syscall well above the average: 17,453 times. In comparison, the sample using this syscall the second-most time has issued it only 44 times. This sample indeed shows lower then average utilization of available system calls, with only 61 syscalls used. When we submitted this sample to the online service `malwr.com`, it has also identified the sample as attempting to delay the execution of the analysis task, but it has identified this attempt based on a request to sleep for 2,088,812 seconds [79].

As our current prototype has not been capturing syscall arguments, the most straight forward check to detect a sleeping malware is not possible based on our current logs. For example, in the malware sample discussed above, a long sleep request would be a clear indication of a stalling malware. Nevertheless, the second hypothesis should arguably still hold if the malware successfully sleeps. In Figure 36 and Figure 37 we can see the distribution of samples that used either NtDelayExecution or NtQuerySystemTime. The data is further contrasted with the samples that have never issued these calls. Based on

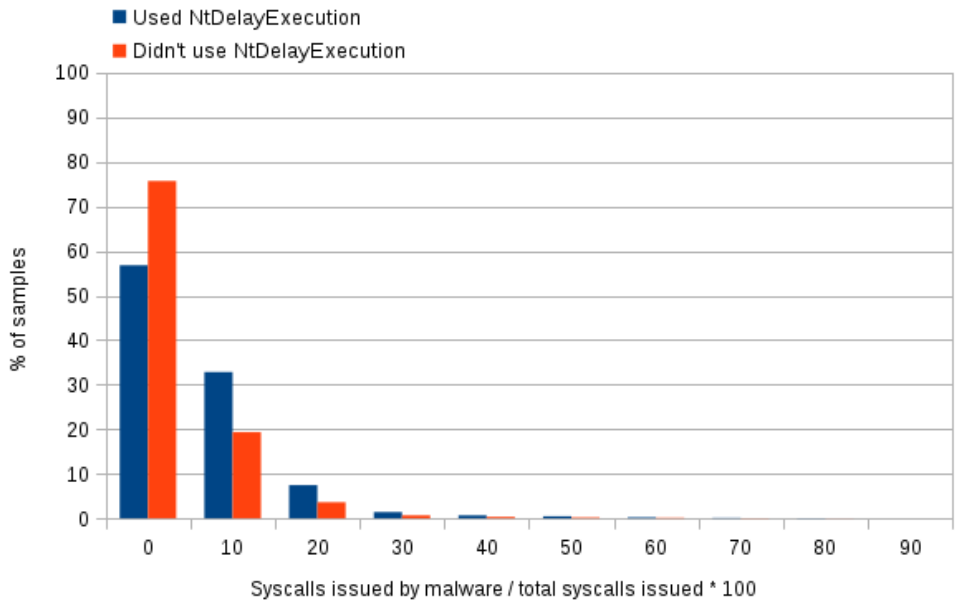

Figure 36: Distribution of samples based on its usage of NtDelayExecution

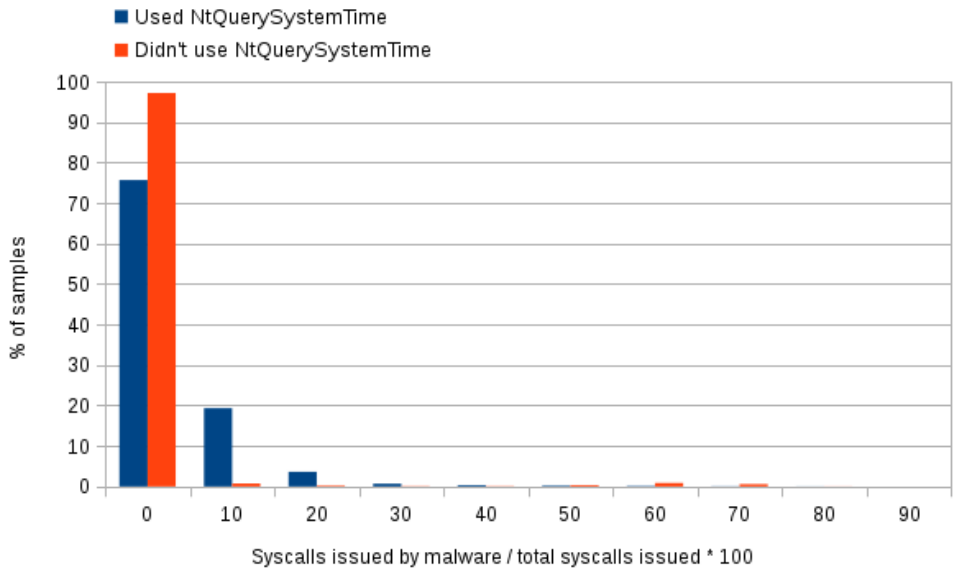

Figure 37: Distribution of samples based on its usage of NtQuerySystemTime

these results we actually see the opposite of what would have been expected if malware sleeps when these syscalls are used: malware *not* using these syscalls on average show a lower execution rate then those malware that do use it. This ratio flips only when the ration goes above 10% of syscalls being issued by the malware samples. However, when more then 10% of all syscalls are being issued by the malware sample, it is arguably harder to label the sample as sleep-stalling.

4.6.5 Measuring overhead and throughput

In this section, we turn our attention to measuring the overhead of our system, a standard metric calculated for any monitoring engine, and to measuring the effective throughput we achieved.

To illustrate the source of the overhead in our system, we measured how many VMEXITs are triggered by our monitoring when all internal kernel functions are trapped. By using the debug symbols for the Windows 7 SP1 kernel all 10,853 internal functions for x86, and 11,011 functions for x86_64 were trapped. The traps were further protected with EPT execute-only permissions, which on x86 resulted in trapping 727 pages, and 915 pages on x86_64, respectively. The VM was unpaused for 60 seconds and was running idle with only the default Windows system processes being active. As we can see in Figure 29, breakpoints were the main source of VMEXITs we triggered in the VM for both versions of Windows.

Also worth noting that no write events were observed within this time period. The experiment was repeated with trapping functions found in ntdll.dll, consisting of 2,196 functions on 103 pages for x86, and 2,263 functions on 133 pages for x86_64. The results were similar as the ones we seen for the kernel, the #BP traps were by far the majority of source of VMEXITs. From these benchmarks it is clear that the overhead on x86 is larger than on x86_64 For example, only 3,970 Read EPT violations were observed on the ntdll.dll pages.

It is worth highlighting that the experiments consisted of trapping all internal kernel functions, thus performance could be improved by selectively placing traps at locations deemed of particular value, reducing the number of #BPs and the number of pages that are trapped. To illustrate how selective trapping affects monitoring we choose to further benchmark the responsiveness of the VM using the *iperf* network performance testing tool. In this test, Windows 7 SP1 x64 had an emulated Intel E1000 network device attached. With no traps injected, the throughput measured in 10 seconds was 943 Mbit/s. When all 11,011 internal kernel functions were trapped performance dropped to 0.34 Mbit/s, a noticeable slowdown. However, when we trapped only those kernel functions starting with Nt, which are the functions available through system calls and listed in the System Service Dispatch Table (SSDT), the performance was 432 Mbit/s.

While the overhead thus added is still more than 50%, this overhead doesn't interfere with the execution of the sample even if it maintains connections to external nodes outside *DRAKVUF* as the uplink bandwidth of the physical network itself is only 100 Mbit/s. In case such monitoring was to be deployed on production systems however a more selective monitoring approach would have to be chosen as to minimize overhead.

To measure the overhead in terms of CPU cycles by the VMEXITs caused by #BP injection, we performed a benchmark similar to that in SPIDER [23]. The test consisted of the monitored domain calling a function within a loop where the function increments a counter. The loop iterates from 10^4 to 10^6 with a step value of 10^4. Timing information was collected by reading the Time-stamp counter (TSC) register before and after the loop. The benchmark was performed with and without trapping at the function entry point to determine the overhead. The overhead thus measured was on average a factor of 10502, which is comparable to the overhead in SPIDER. The difference in the overhead likely arises from using different hypervisors, Xen in *DRAKVUF* and KVM in SPIDER respectively, as by design Xen requires an extra VMENTRY/VMEXIT for each trap that is forwarded to the control domain.

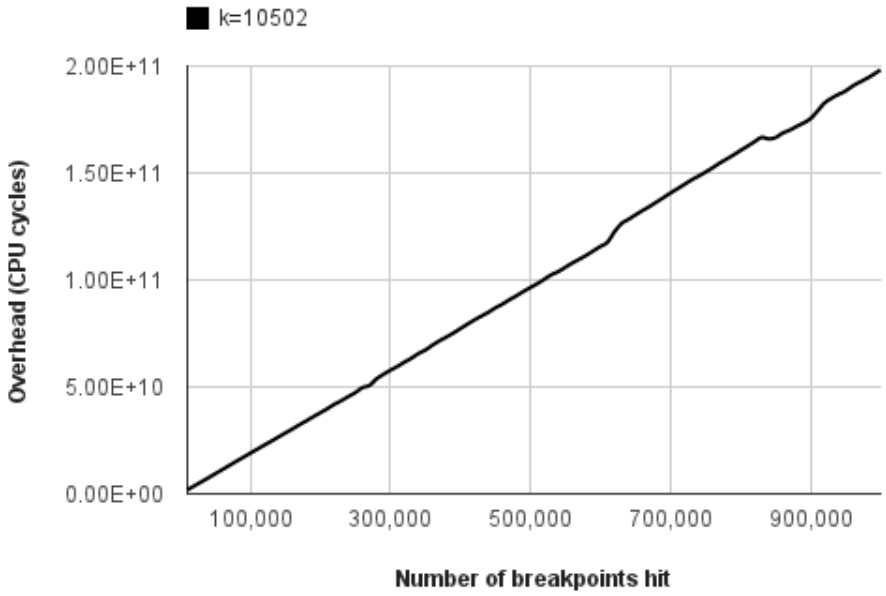

Figure 38: Relation between the overhead and the number of `#BP` hit.

While calculating overhead this way is a standard and informative metric used to evaluate malware analysis systems, in the context of malware analysis the fact that a malware sample's execution speed is degraded is only relevant in case such a degradation actively interferes with or alters the malware's execution. In other words, overhead on a per sample base is mainly a *stealth* concern for dynamic malware analysis. In our opinion the performance of a malware analysis system is more realistically measured by determining how many concurrent analysis sessions can be executed simultaneously given specific hardware constraints, as shown by for example Paul Royal [99].

From a *scalability* perspective the throughput of the system is of particular importance, which we further evaluated based on the experiments described in Section 4.6.2. The main hardware constraint in our system was the amount of available RAM: 16 GB. Considering the standard 2 GB of RAM recommended for running Windows 7, the maximum number of concurrent sessions with existing open source tools [14] would be limited to 8 sessions (not counting memory allocated for the control domain). In our first set of exper-

iments, discussed in Section 4.6.2, we achieved on average an effective memory saving of 62.4% by using copy-on-write memory, with a standard deviation of 7.3%. Projecting the memory savings as concurrent sessions, shown in Figure 30, we can see the immediate memory savings that can be achieved via CoW memory (the area shown in black). In our case, the number of concurrent analysis sessions has improved by a factor of two, already a significant improvement in throughput. In the second, larger scale experiment described in Section 4.6.3, we observed that on average 77% of RAM remained shared with a standard deviation of 4.49%. Thus, based on this sample set, on average we could overcommit RAM to 4x the number of VMs. This second experiment was also performed with concurrent analysis sessions, in two phases. In the first phase, the max concurrency level was set to a maximum of 4 VMs. These results further strengthen our projected overcommit capabilities deduced from the malware collection experiments, discussed in Section 3.3.4.

4.7 Summary

As these experiments prominently show, hardware virtualization offers a variety of properties which can be used to great effect when developing malware analysis tools. The tools we developed have highligthed during live experiments that using hardware virtualization based analysis effectively allows us to peek into the behavior of modern malware. Using our collection techniques we have gained valuable insight into previously unknown behaviors. Furthermore, using the aggregate data generated during our extensive experiments we had obtained a general overview of the current malware landscape, which can further aid analysts in better focusing their efforts. To reflect on our core objectives, we summarize our findings in the following.

(O1) Scalability: Our tests performed with live malware allowed us to effectively evaluate how copy-on-write disk and memory techniques cope with this use-case. Our experiments have prominently showed that significant improvements are possible

via this technique. Nevertheless, as we have seen during our malware collection tests as well, the benefits rapidly decrease the longer the analysis sessions are active. Future work should explore the possibility of performing continuous memory deduplication to preserve the benefits achieved herein.

(O2) Stealth: The techniques implemented and deployed in our experiments have successfully shown that hardware virtualization can be effectively used for steathy malware analysis. By addressing the hitherto overlooked problem of starting the malware sample without leaving a trace successfully greatly increased the stealthiness of such systems.

(O3) Fidelity: We have demonstrated how a wide variety of information can be obtained from the executing virtual machine purely with the use of out-of-band introspection tools. In our prototypes we have been able to implement standard data collection techniques previously applied or proposed, such as system call interception, kernel heap monitoring and file extraction. The breadth and depth of the data we collected highlights how hardware virtualization is a flexible technique allowing the monitoring of highly varied types of behaviors. This breadth of information captured greatly increases the visibility into the behavior of modern malware.

(O4) Isolation: Improving upon our disaggregated design, we have greatly reduced the attack surface of our analysis system by moving the data-collection tools into deprivileged secondary control domains. Furthermore, the combined use of virtual switching technology with our external injection tool allows unprecedented flexibility and ease in deploying such systems, without sacrificing stealth in the process.

5 Hardware and Software limitations

During our investigation into hardware virtualization as a platform to build security tools, several hardware and software limitations were encountered. In the following chapter we provide a summary of our findings which need to be considered when building such applications. First, we highlight the inherent limitations of our execution monitoring approach introduced in Chapter 4. Afterwards, we revisit subversion attacks proposed in the past for VMI applications and evaluate their prevalence on modern hardware. Finally, we take a look at the most privileged operation modes found on modern CPUs and how these may affect future development directions.

5.1 Evading monitoring

In the current implementation of our prototypes, as described in Section 4.5, monitoring the malware's execution is performed by trapping internal kernel functions corresponding to system calls. This monitoring is performed by overwriting the kernel function entry points with the software breakpoint instruction, `0xCC`. Furthermore, the kernel heap allocation and deallocation routines are trapped to monitor critical datastructure placements on the OS's heap. While this tactic is effective against DKOM attacks, as discussed in Section 4.5.1, it would not be enough to overcome other types of attacks.

In our implementation monitoring starts on a clean system, thus we have a reasonable trust in the initial reconstruction of the OS's state. However, as the malware infestation is performed, a kernel-mode rootkit could break the assumption our monitoring is based on. *Hooking* the system call table has been a standard rootkit technique to redirect critical system calls to attacker-controlled functions to perform stealthy monitoring and filtering of the data going through the system call. Similar hooking can be performed on the IDT and GDT as well. In the prototype implementation of our malware analysis tool we have not implemented any additional protection and monitoring mechanism to detect such hooking

behavior. There are no technical reasons why such monitoring cannot be implemented, it has simply been out-of-scope for our initial prototype. Placing EPT write-notification events on the critical system tables would immediately alert the external monitor to such hooking behavior. Furthermore, once the hooks are placed, the external monitor could further trap the new location of the functions and log when the hooks are executed.

Once a rootkit is thus placed itself inline of the systems execution, the level of trust in the data collected decreases. A rootkit can theoretically implement many of the OS's standard functionality on its own, thus avoiding our monitoring which relies on the malware making use of the original OS for its functioning. For example, a rootkit could implement its own heap allocation and memory management routines, thus avoiding going through the standard - and monitored - OS heap allocator. However, the more functions the rootkit implements such a way, the larger its size becomes, which may be a limiting factor in how far the malware authors would be willing to go. Another possibility a rootkit could do is to make use of the standard functions, but instead of executing the functions at their proper entry points, implementing the first couple instructions on its own, then jumping into the function further into the code then the entry point. Since our monitoring hooks are placed only at the function entry point, the rootkit would successfully avoid our monitor while also limiting its binary size. It is thus necessary to stress the point that while our prototype is capable of monitoring kernel level rootkits as well, the level of trust in the collected data can be impacted by rootkits.

Nevertheless, it is still possible to develop routines to counter such rootkits in a transparent and automated way. For example, after the installation of a kernel-mode rootkit is detected, we are able to determine where the rootkit's code is located by following the hooks placed in the OS. Once the infected memory space is identified, the external monitor could switch monitoring from the breakpoint-based, low-overhead monitor, to the fully EPT based monitoring. The benefit of the EPT based monitoring would be that full execution trace of the rootkit can be collected, without it being able to jump over monitor points.

Using this monitoring technique, we would be able to detect if the rootkit is attempting to jump over monitor hooks. We anticipate this to be an on-going arms-race where malware authors come up with ever more exotic execution paths to avoid and confuse automated monitoring, such as data-only rootkits [118].

5.2 Attacks via the TLB

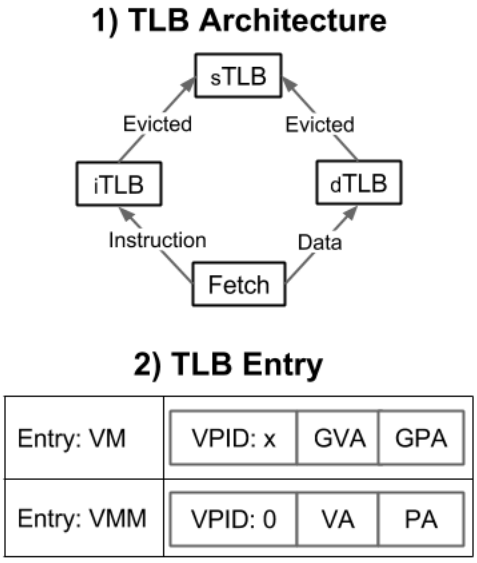

Figure 39: Overview of the split and tagged TLB architecture.

As V2P address translation is expensive, even with hardware acceleration available, modern CPUs maintain a transparent cache to store the translation results called the Translation Lookaside Buffer (TLB). To further improve performance, Intel implemented a split TLB architecture which separates the cache into two disjoint sets. The iTLB stores translations for instruction fetches and the dTLB stores translations for data fetches. In newer CPUs, Intel added a secondary cache called the sTLB, which stores the evicted entries from both the iTLB or dTLB to offer even faster address resolution. An overview of the TLB system is shown on Figure 39.

The split TLB architecture has been used both for defense and offense. The first system to make use of the split TLB was GRSecurity's PAGEEXEC feature in which they

tackled the problem of marking a page non-executable without explicit hardware support for it (like the NX-bit in later CPUs) [46]. In recent years it has also been proposed and used in similar fashion to ensure the integrity of code which resides on pages that mix code and data segments [96, 112].

Input: Splitting Page Address (addr), Pagetable Entry for addr (pte)
1. `invalidate_instr_tlb` (*pte*);
2. `pte = the_shadow_code_page` (*addr*);
3. `mark_global` (*pte*);
4. `reload_instr_tlb` (*pte*);
5. `pte = the_orig_code_page` (*addr*);

Table 9: TLB poising algorithm as described by [6].

From an offensive perspective, the Shadow Walker rootkit leveraged this architecture for stealth purposes [107]. The rootkit took advantage of the fact that a virtual address can point to different physical addresses based on which TLB is utilized. In such a split, the rootkit's code can safely execute without antivirus software being able to scan its code pages [130]. To further make the rootkit more persistent against TLB flushes, the rootkit's code pages are also marked as global. The algorithm to perform this TLB poisoning is shown in Table 9. The poisoned global pages can only be flushed if the TLB is full and the entry is evicted by the hardware, or by turning off and on the Page Global Enabled (PGE) bit on the CR4 register. For example, Windows 7 actively and frequently flushes the global pages from the TLB and disabling the routine leads to a system crash almost immediately. This behavior effectively limits the life-time of the poisoned TLB. Linux on the other hand does not perform any such TLB flushes, making it a more potent target for TLB poisoning.

In recent Intel CPUs (Nehalem and newer), a secondary victim-cache has been added to the Intel architecture: the sTLB. The sTLB holds all entries which are evicted from either the iTLB or the dTLB, and in the event of a TLB-miss, the sTLB is checked before the system performs a real address translation. This further complicates the development

of stealthy rootkits, as the sTLB can only hold one version of the evicted TLB entries. This in effect synchronizes the split-TLB without triggering a pagetable lookup. Thus, a rootkit leveraging the split-TLB technique is now unreliable, as its custom #PF handler code is not invoked to re-split the TLB. This leads to either a non-hidden rootkit code-page if the iTLB entry is brought back from the sTLB, or a system crash / infinite loop if the dTLB entry is brought back and is being accessed as if it was code.

As such, malicious code running within the guest on modern Intel hardware is unable to leverage the split-TLB for hiding malicious code from other applications running within the guest. However, with a little help from the VMM, the behavior of the CPU can be adjusted to skip entries being evicted into the sTLB. Unlike regular page-table entries, EPT entries allow setting a page to execute-only; that is, it can be accessed only by code-fetching. When the CPU detects that the iTLB and dTLB have different EPT permissions (one with R/W for data and the other with X only for code), evicted entries skip the sTLB [113]. Thus, if the in-guest TLB-split routine is created by or in coordination with the VMM, the sTLB can be by-passed so that the address translation goes through the primed page-tables again and effectively enabling the Shadow Walker technique.

On the ARM platform, similarly to Intel, a separate iTLB and dTLB is maintained, while the virtualization extensions also offer TLB tagging akin to VPID. However, there is no sTLB present to potentially synchronize a split-TLB setup. Another important difference between the Intel and ARM architecture is that while on Intel it is not impossible to perform stage-1 only translation (GVA to GPA), the ARM instruction set provides such facilities. Thus, an external observer can repeat the translation in the context of the virtual machine such that the lookup also queries the TLB. In effect, even if the pagetables no longer represent the cached translation, the external observer can receive the de-facto used translation result. By further performing a software-based pagetable lookup the external observer can further determine if the TLB entries our out if sync with the pagetables. The only limitation with this hardware based lookup method is that it performs the lookup as a

data-fetch access - thus only allowing one to obtain the cached dTLB entry. In case the iTLB is poisoned with the pagetables no longer representing the cached translation, it is impossible for an external entity to recover the in-use instruction-fetch address. This is a particularly dire limitations, as the second-stage translation faults only report the glsgva to the hypervisor. Thus in effect it is impossible to tell what instructions the guest is actually executing in case the iTLB is poisoned.

5.2.1 The effects of tagged TLB

In the first implementations of Intel VT-x, the TLB entries were completely flushed during VMENTRY and VMEXIT operations. As a side-effect, this provided a good security counter-measure to a Shadow Walker style TLB technique for hiding in the guest. With newer VT-x implementation, the concept of the tagged TLB has been added to Intel, dubbed Virtual Processor Identification (VPID). With VPID, the hardware does not flush TLB entries during VMENTRY and VMEXIT. This is accomplished by a new field in the VMCS which the CPU can now use to distinguish between VMs based on the assigned tag. This naturally results in significant performance boosts on modern hardware.

In a footnote Bahram *et al*. [6] speculated that with tagged TLB being available, hiding in the TLB will reemerge as a method of achieving stealth against VMI applications. However, at the time no hardware was available with tagged TLB support to test their proposal. In contrast, nowadays most modern Intel CPUs have both the sTLB and the VPID feature. As we already discussed, the introduction of the sTLB already affects how TLB-splitting can be performed, which consequently diminishes the utility of the split-TLB as a technique to hide malicious code in a guest. However, hiding from VMI doesn't necessarily require malicious code to use a split TLB. For VMI applications, the TLB itself can be a problem, as VMI always emulates address translation in software using the in-guest page tables. Any translation cached in the TLB that is no longer reflected in the page tables is effectively invisible to external monitors. In the following, we aim to highlight the

different VPID implementations available in modern open-source hypervisors to highlight how these implementations affect VMI applications.

The Intel VPID is a 16-bit field included in the VMCS for each vCPU. The assignment of tags is left to the hypervisor with the exception being that tag 0 is a magic tag specifying the VMM. The tagged TLB entries can be flushed by specifying the tag, flushing all tagged entries, or assigning a new tag to the vCPU so the hardware will eventually evict the stale tags. While the description of the tagged TLB is straight forward, the actual implementation in open-source hypervisors varies greatly.

Xen implemented the VPID as a round-robin counter, where a tag is assigned in the VMCS simply by incrementing the counter. When an overflow occurs, all TLBs are flushed, and the iteration restarts at 1. During regular operations, the tagged TLBs are never flushed, instead a new VPID is assigned to the vCPU. As the comment describes it in the Xen source: ”Each time the guest’s virtual address space changes (e.g. due to an INVLPG, MOV-TO-CR3, CR4 operation), instead of flushing the TLB, a new [VPID] is assigned. This reduces the number of TLB flushes to at most 1/#[VPID]s. The biggest advantage is that hot parts of the hypervisor’s code and data retain in the TLB”. However, on a closer look at the source we see that the comment is only partially true: the VPID does not get invalidated on MOV-TO-CR4. Additionally, with new VPIDs being assigned on MOV-TO-CR3, the hypervisor effectively disables the guest’s ability to use global pages.

If we recall, the purpose of having global pages is to make the TLB entries survive a MOV-TO-CR3 so that the translation can be shared across processes. On Xen, the hardware won’t be able to utilize global TLB entries as the VPID tag of the global page is stale after the context switch. As a side-effect, the TLB priming would have to be performed on every context switch. On KVM the VPID implementation is done by using a bitfield. When KVM creates a new vCPU structure, the first unused bit is reserved to this vCPU. As a peculiar decision, the tag is assigned even to vCPUs that never execute; that is, it is assigned during the creation of the vCPU, not during the first VMENTRY. When

the VMM runs out of available tags to assign, it simply disables VPID in the VMCS. This implementation means KVM guests can prime the TLB with global pages without having to perform this operation on every context switch.

5.3 Limitations of the EPT

In the following section we take a closer look at the EPTs from a VMI tracing perspective. We briefly introduce the extension and how it is used for great effect in various VMI applications. Afterwards we turn our attention to the limitations of the extension and discuss various pitfalls that need to be taken into consideration when building security applications relying on the extension. While these limitations do not automatically break VMI applications, without proper handling they could lead to subversion attacks. While developing real-world security applications the authors have encountered these limitations.

5.3.1 Catching modifications

Now that we have a brief overview of how EPT violations can be used to trace memory accesses performed by a VM, we aim to highlight the limitations of EPT through some examples. While the limitations are not necessarily prohibitive, without proper consideration while developing security applications, they can result in a knowledgeable in-guest attacker hiding from 'naive' protection schemes.

DKOM attacks are a well-known way of breaking both forensic memory analysis (FMA) and VMI tools. DKOM works by modifying data-structures in the kernel's heap. The classic example of this is by hiding a malicious process by unhooking its data-structure from the linked-list that the OS uses to report active processes to tools such as *ps*. As the linked-list is a non-critical and independent structure from what the OS uses to schedule processes, this type of DKOM attack breaks the assumption that the list is well maintained and accurately describes the list of active processes.

In the context of detection of unhooked processes, a security application can trace

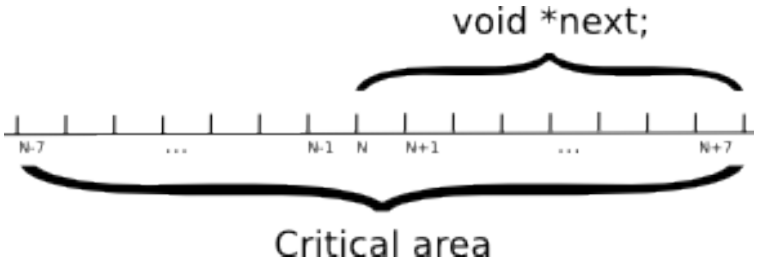

Figure 40: The critical memory region at which EPT violations may occur that could mean an access to the protected region (void *next).

when updates are made to the linked list via the EPT. This is done by checking the violation information contained in the VMCS to see whether it is at the offset of the pointers *next* and *prev*, as a security application has direct knowledge of the updates made to the linked-list. Such an approach may seem straightforward because during normal operations the offset at which the violation is reported matches the location of the pointers. That is, the operating system updates the pointers *directly*. However, a critical limitation in the way the hardware reports EPT violations needs to be taken into consideration in this scenario. During an EPT violation, the CPU only records the starting address where the violation occurred on the monitored page. However, the violation may involve a read/write operation up to 8-bytes. This short-coming is known by Intel, as they already patented the solution [114]. As such it may be the case that some future CPUs will provide this information as well.

If an attacker knows that only the exact addresses are going to set off an alarm and the rest are filtered as unrelated violations, it then becomes possible to perform a successful DKOM or other attack. The only task is to break the assumption that the violation will be at exactly the pointer locations. Figure 40 highlights the entire critical memory region that security applications looking for EPT violations need to consider. For example, overwriting the pointers in two steps could perform the attack: first, write 8-bytes starting at (N-1); second, write 1-byte starting at (N+7). The DKOM attack will still trigger VMEXITs, but our naive protection scheme would disregard the violations as unrelated write-events. This limitation has to be kept in mind when building systems against memory disclosure

based attacks, such as in HeisenByte [111]. In this system Tang *et al.* proposed using EPT to catch when code is being read to construct return-oriented-programming (ROP) gadgets, and subsequently destroying the code to prevent it being used. While the authors note that the code destruction can be extended to arbitrary lengths, in their prototype implementation they have limited it to a single byte, potentially exposing up to 7-bytes per read for code-reuse attacks.

5.3.2 Catching data-leaks

Now we will turn our attention to another potential security feature that EPT could be used for: data-leak prevention. In data-leak prevention systems, it is crucial that specific memory locations are accessed only under certain circumstances. An external security application can potentially use EPT's read protection to enforce a mandatory access control system. The limitation described in the previous section is applicable under this scenario as well. However, EPT has another crucial limitation which could be utilized to siphon protected data without triggering an alert.

The limitation is in how EPT violations are reported when a read-modify-write (r-m-w) instruction is executed. According to the Intel manual: "An EPT violation that occurs during as a result of execution of a read-modify-write operation sets bit 1 (data write). Whether it also sets bit 0 (data read) is implementation-specific and, for a given implementation, may differ for different kinds of read-modify-write operations" [56]. The implications of this behavior are apparent: any memory event subscriber solely looking for read EPT violations as the trigger for enforcing an access control system can be subverted by employing r-m-w operations instead to access the data.

Despite the fact that such unpredictable behavior surrounds these operations, current hypervisors make no attempt in mitigating it in software. For example, up until recently Xen simply forwarded the violation information from the hardware to memory event subscriber applications. Only with our recent patch does Xen mask the hardware limitation from

applications by unconditionally marking all write violations also as read violations.

5.3.3 Virtual DMA and emulation

Thus far we have looked at the hardware limitations of using EPT and its effects on VMI application. In the following we discuss two more scenarios where EPT protected memory regions can be accessed without triggering EPT violations, based on a recent discussion thread on the xen-devel mailinglist [70].

In modern virtualization environments device emulation is a critical component in allowing off-the-shelf operating systems to run without requiring it to be virtualization-aware. For example, Xen uses QEMU to provide various emulated device backends for Windows (or Linux) virtual machines, such as VGA, disk, and network devices. Device emulation however is known to be complex and error-prone, thus being a fertile ground for various exploits. For example, it has been used in the past for breaking out KVM virtual machines [37].

As to mitigate the risk involved in running the QEMU instance in the Trusted Computing Base (TCB), Xen introduced the concept of stub-domains, where the QEMU instance is running in a light-weight paravirtual VM next to the main VM it provides emulation for. Thus, even if an attacker breaks out of the VM via the emulated devices, it only gains access to another de-privileged VM. Nevertheless, such break-outs are not without consequence from a VMI perspective.

On Xen, even the de-privileged QEMU stub-domain requires Direct Memory Access (DMA) into the main VM in order to provide the I/O emulation it is tasked with. In a hypothetical break-out where an attacker successfully compromised the QEMU stub, the DMA functionality can be used to by-pass any type of EPT traps set on the main domain. That is because the stub can request any memory page of the main VM to be mapped into its own address space by the hypervisor. However, the stub being a paravirtual domain, does not use EPT to access the same memory.

Similar problems can be potentially found with the emulated interrupts and timers. For performance and scalability reasons with many-vcpu guests, Xen provides a fast-path emulation for a variety of interrupts and timers, such as RTC, PIT, HPET, PMTimer, PIC, IOAPIC, and LAPIC. Since the emulation happens within the hypervisor, any updates to pages with EPT traps, that happen as a result of emulation, avoid triggering these traps.

5.4 Layers below the hypervisor

In recent years there has been a number of attempts to move VMI applications to a layer below the hypervisor, namely the SMM and the TrustZone. However, these layers present particular challenges for VMI applications, especially when faced with non-cooperating or malicious VMMs. In the following we present a quick overview of these CPU layers and discuss potential problems in utilizing these modes as a platform for VMI. Fundamentally, both the SMM and TrustZone share similar characteristics in that these CPU execution modes co-exist with the VMM and VM execution modes. Similar to how the VMM is a more privileged execution mode then the VM mode, the SMM/TrustZone is yet another layer more privileged then the VMM.

5.4.1 ARM TrustZone

The ARM TrustZone execution mode is an ARM specific mode designed to provide an isolated enclave for sensitive applications to operate in. The main design principle behind TrustZone has been that an application running in normal mode - be it the VMM or an application within a VM - can request the sensitive operation to be performed by a process within the TrustZone, which would provide protection to sensitive data, such as encryption keys or passwords. The scheduling of the TrustZone relies on two mechanisms: execution of the Secure Monitor Call (SMC) instruction; or alternatively the pre-configured trapping of certain interrupts directly to the TrustZone.

When executing in the TrustZone, the CPU has access to all memory within the sys-

tem, thus performing VMI from the TrustZone is possible, thus two of the main VMI requirements are readily met: *Isolation* and *Interpretation*. To achieve effective *Interposition* the TrustZone based VMI system can rely on the re-routed interrupts as a mechanism to evaluate the system state, albeit it potentially resulting in significant overhead. Alternatively, the TrustZone based system could achieve interposition by the injection of SMC instructions into target code locations, as also proposed by the SPROBES systems [44], and also by Samsung [5].

A non-cooperative VMM can pose particular problems when attempting to use the SMC injection mechanisms, as SMC is trappable by the hypervisor as well. In such a case, the TrustZone would not be notified of the SMC being triggered in a target VM if the VMM decides to forgo forwarding such events. Furthermore, the SMC can only be triggered from the guest operating system, thus it is not possible to trace the execution of user-mode code directly via this method.

The non-cooperative VMM scenario could be overcome by injecting a *trampoline SMC* into the SMC handling routine within the VMM. This in effect would force the VMM to trigger the TrustZone whenever an in-guest SMC instruction has been trapped. The VMM could still attempt to detect if it's SMC handler has been trapped in such a way, and subsequently remove those traps. Provided that the TrustZone is always in a more-privileged mode, as long as some interrupts trigger the execution of the TrustZone, the traps can always be reinserted periodically. This further highlights the implications if we consider the opposite setup, where a malicious entitity has been able to establish a foothold in the TrustZone. In such a case, normal oeprating modes would have no recourse to protect themselve from the TrustZone based malware, thus VMM based security solution would be susceptible to subversion attacks.

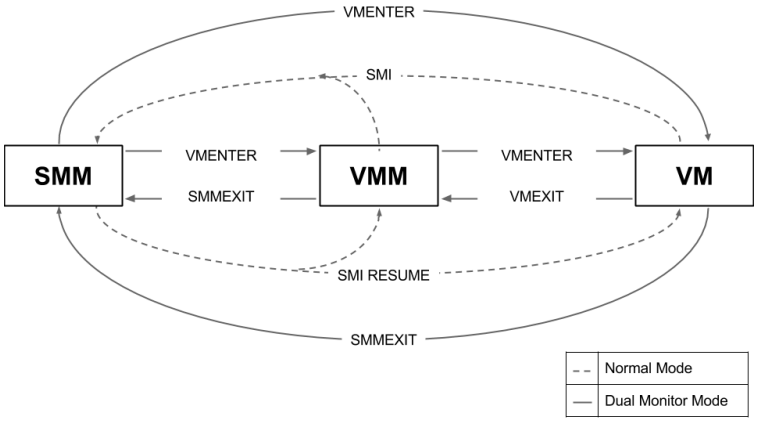

Figure 41: Overview of relationship between SMM, VMM, and VM.

5.4.2 System Management Mode

The SMM, according to the Intel manual [56], was designed to provide an alternative, transparent operating environment to chipset manufacturers and BIOS vendors. This mode can be used to monitor and manage various critical system resources for more efficient energy usage, control system hardware, and respond to thermal emergencies in the event of a non-responsive OS. It is highly privileged and cannot be interrupted by regular interrupts (including non-maskable interrupts), thus guaranteeing the execution of the SMM code once it is triggered. Furthermore, code running within the SMM has full access to the system and can perform arbitrary modifications to the system RAM. In recent years the SMM has received considerable attention from security researchers; for example, it has been shown that SMM could be used to implement stealthy rootkits [108, 125], VMM integrity verification systems [101, 122] and full-scale VMI applications [134].

In order to trigger the execution of code within the SMM, a System Management Interrupt (SMI) is issued via the local Advanced Programmable Interrupt Controller (APIC) or via the SMI# pin. In practice the APIC method is the preferred trigger mechanism as the SMI# pin requires extra hardware to be attached [22]. When the interrupt is received, the active CPU context is saved into a dedicated memory region, known as System-Management RAM (SMRAM). The chipset can be configured to trigger an SMI on a multi-

tude of possible events, from USB activity, writes to certain I/O ports, thermal events, and even periodically [123]. When the SMI handler finishes executing, the previous interrupted operating mode is restored from the snapshot that was saved into SMRAM. An overview of this operation is shown in Figure 41 with the dotted lines. While the SMI handler can modify the CPU register values that were saved, the CPU always returns to the same operating mode that was active when the SMI was received: VMX root or VMX non-root.

As pointed out by Jain *et al*. [57]: ”A limitation of any SMM-based solution [...] is that a malicious hypervisor could block SMI interrupts on every CPU in the APIC, effectively starving the introspection tool. For VMI, trusting the hypervisor is not a problem, but the hardware isolation from the hypervisor is incomplete.” While it is true that the VMM could starve the SMM in such a way, the SMM is capable of modifying the VMM’s code to remove such code-paths as long as some SMI is triggered. Furthermore, as the SMM is initialized as part of the BIOS before the VMM, any protection the VMM may attempt to leverage against the SMM could be potentially circumvented before it is applied.

5.4.3 Dual-monitor mode SMM

The Intel manual also describes an additional CPU mode referred to as *dual-monitor mode SMM (DMM))*. DMM was created to prevent or limit the damage from the exploitation of vulnerabilities in SMI handlers to gain control of SMM. An example of such an attack can be found in [129]. While the original idea behind this implementation was to compartmentalize the SMI handlers into less privileged SMM VMs [80], the capabilities of the hypervisor running in SMM, the SMM Transfer Monitor (STM), in our opinion far exceed what would be warranted.

The primary difference between the two SMM modes is that in dual-monitor mode the SMM enters VMX root mode itself. In this mode, the STM is allowed to create virtual machines within the SMM to run the SMI handlers. In comparison, AMD and ARM simply enabled the regular hypervisor mode to trap SMIs, thus being able to compartmentalize

SMI handlers into regular VMs [2, 3].

With the introduction of this new mode, the behavior of the VMCALL instruction has also been extended. On CPUs without DMM support, the VMCALL instruction is only valid if executed in VMX non-root - that is, in a virtual machine. On CPUs with dual-monitor mode, the VMCALL instruction is valid even in VMX root and triggers an unconditional transfer into the SMM.

This behavior has particular importance when we consider how a VMM would attempt to starve the SMM, as we discussed for normal-mode SMM. To recall, in normal mode the APIC could be configured by the VMM to starve the SMM. Now with a dedicated instruction it is no longer possible, as the instruction unconditionally triggers the switch without relying on the APIC to trigger the required interrupt.

As this instruction is no longer pre-emptible from the VMM, it can also be utilized by the STM to position itself inline into the execution of the rest of the system. The STM could inject this instruction as a trap to trigger the transfer of control when code-paths of interest are executed within the VMM. While the presence of these hooks may be detectable if the VMM performs code-integrity checks on itself, the SMM could disable such integrity checks. While this instruction cannot be used to instrument virtual machines - as it would trap into the VMM - it can be used to hook the VMM's trap handlers. Afterwards, the STM can use any of the regular instrumentation methods the VMM has over VMs, thus attaining total control over the execution of the entire system.

Another particularly powerful addition in the dual-monitor mode is that the SMM can jump into any of the other execution modes of the system. In a non-DMM configuration, execution is returned to the same mode that was active before the SMI was triggered whereas in DMM, there are no such restrictions. For example, this mechanism allows the SMM to schedule the execution of VMs that the VMM deliberately suspended. Interestingly, the *stm* can also specify if further SMIs should be blocked for the duration of a VMENTER: "VM entries that return from SMM and that do not deactivate the dual-monitor

treatment may leave SMIs blocked. This feature exists to allow an SMM monitor to invoke functionality outside of the SMM without unblocking SMIs" [56]. This design decision is particularly interesting, as now even SMIs can be blocked.

5.5 Summary

In this chapter we have highlighted the inherent limitations of our execution monitoring approach and revisited VMI subversion attacks proposed in prior research. By examining modern open-source hypervisors that make use of current hardware virtualization extensions, we have been able to determine their pertinence for current applications. We further discussed inherent limitations in using hardware virtualization extensions for VMM-based security monitoring and highlighted how these limitations can have unintended side effects when not taken into consideration. We further explored ARM's TrustZone and Intel's system-management mode and discussed how it can be used to both implement and to subvert VMI applications. As these system represent a more privileged mode of operation as our VMM based systems, it is important to understand the implications these modes represent when building security tools, as exposure of these modes could undermine all other security systems on modern hardware.

6 Conclusions

6.1 Summary

As the threat of malicious software attacking and infiltrating our computers constantly increases, in-depth understanding of these threats is of tremendous importance. A first requirement to be able to better protect our assets is to gain a detailed under standing of threat vectors by examining the malicious software's effect on our systems. Honeypots and dynamic malware analysis systems have been of great importance in this on-going arms race and we have presented a comprehensive evaluation of hardware virtualization as an underlying platform to significantly improve these types of systems. We started by identifying the core requirements these systems require: scalability; stealth and unhindered, tamper-resistant view into the behavior of modern malware. We followed by reviewing existing systems and prior research which attempted to address these requirements. We then introduced different software solutions to help evaluate the potential of hardware virtualization for achieving all requirements simultaneously. The cornerstone of our approach has been a VMI based observation platform that is capable to perform state-reconstruction triggered by hardware events directly, without any identifiable software being present in the sandbox environment. Further combined with state-of-the-art hypervisor and virtualization technology, our prototypes have successfully demonstrated that virtualization provides the unparalleled scalability, stealth and tamper-resistance. We successfully evaluated our system using a variety of modern malware, both in the forms of network-based attacks as well as in bulk-analysis of previously captured malware samples. Our final malware analysis prototype marks a major step toward leveraging virtulization technologies to create powerful security solutions.

6.2 Contributions

The first contribution of our work has been a comprehensive implementation of the required VMI software systems that enable out-of-band state reconstruction. We developed various routines to allow in-depth examination of the Windows kernel from an external perspective, and have worked with the open-source community to make our contributions available to the defensive research community at large. All the requirements for our prototypes have been successfully contributed back into projects such as the LibVMI library and the Xen Project Hypervisor, making it possible for other researchers to easily deploy and repeat our experiments.

The second contribution of our work has been identifying and evaluating the major requirements facing malware collection and malware analysis tools. We have identified several misconceptions in prior research regarding what metrics to use in the attempt to show viability of proposed systems. In our work we have focused our attention on metrics that are of immediate importance, some of which have thus far been overseen by other proposed solutions.

Our third contribution is to have deployed and tested our systems on a large scale, including live malware as well as bulk processing of large malware sets. Thanks to the scalability of our systems, we have been able to effectively analyze tens of thousands of malware samples on commodity hardware. Our tests have further provided an in-depth insight into the behavior of modern malware.

Our fourth contribution is to have identified hardware and software limitations inherent in the use of out-of-band VMI tools. We have shown several limitations in the hardware, both in Intel and ARM, that could adversely affect the reliability of security systems based on this technology, and have provided recommendations in avoiding encountering such problems in the future.

6.3 Future directions

6.3.1 Intel Virtualization Exceptions

As virtualization extensions continue to evolve, alternative security models become more viable for defense security research that were previously unable to guarantee secure isolation. In recent years Intel has introduced a new extension, dubbed #VE (short for Virtualization Exceptions). This has been in direct response to the defensive research community's finding that EPT based tracing present significant overhead that can prove prohibitive for certain applications. The first - and thus far only - feature known that will use the #VE mechanisms is the newly introduced `VMFUNC` instruction's `EPTP switching` option.

The EPT extension on Intel has been from the beginning capable of maintaining up to 512 distinct EPTs in the `VMCS` for each vCPU. Nevertheless, all modern open-source hypervisor currently use only one EPT shared across all `VMCS` of the VM. However, in case the hypervisor used more pages, it would be possible to maintain one restrictive table used for trapping and one table for allowing the execution to flow normally. The `VMFUNC` instruction is designed to allow switching between such "views" from within the guest. Further allowing EPT violations to be selectively delivered to the guest in the form of interrupts without performing a `VMEXIT`, the performance overhead of the EPT based tracing can be significantly reduced.

As the hypervisor can still selectively configured which EPT Page Table Entry (PTE)s are injected via #VE into the guest and which ones are still trapped to the hypervisor, the hybrid model of using both in-band and out-of-band delivery model first proposed by Payne et al. [87] is becoming a lot more viable in the near future.

The first open-source implementation by Intel for the upcoming #VE extension has already appeared in the Xen Project Hypervisor 4.6 release, and has been dubbed the *altp2m* subsystem. In a joint effort, we have worked with Intel to enable this feature to be

used in a purely out-of-band monitoring scenario as well. The core feature added to Xen that has immediate use-case for out-of-band scenarios is the support added for multiple second-stage pagetables. In Section 5 we have identified the shared EPT as a core limitations facing multi-vCPU guests. With the possibility of maintaining multiple sets of tables, it is now possible to selective switch the view only on the violation-causing vCPU. While this method still doesn't obtain the performance benefits of the hybrid in-band/out-of-band model, it will enable our prototypes to perform analysis on multi-vCPU guests as well.

6.3.2 Mobile malware

With the rapid growth of the use of smart-phone devices there has already been a rapid expansion of malware targeting this new platform [69], thus malware analysis has to be able to also observe these types of malware. In our evaluation we have performed an initial exploration of the ARM CPU's virtualization extensions to support the type of malware analysis discussed in this thesis. As part of this effort, we have implemented an initial set of tracing features based on the ARM two-stage paging mechanisms that will be available as part of the Xen Project Hypervisor 4.6 release. As discussed in Section 5, execution monitoring cannot be considered to reliable at this stage and alternative mechanisms need to be implemented. The SMC instruction has already been proposed as a viable method to achieve execution monitoring [5, 44], but there is no hypervisor support present for it at this time. Furthermore, at this time there is no singlestepping support available for ARM platform which forms another obstacle to porting our prototypes to the ARM platform. As such, further research is required to properly identify all the capabilities of the ARM platform.

6.3.3 Data-only malware

As new and thus far unknown forms of malware appear, malware analysis systems will also need to adapt. In recent research Vogl et al. [118] demonstrated the feasibility of persistent data-only malware, capable of performing arbitrary computations without inserting any new code into the system. The detection and analysis of such malware will pose a particular challenge in the future, albeit some proposals have already been made to potentially detect such malware using performance monitoring counters [135]. These counters have been shown in prior research to be trappable to the hypervisor [116], thus malware analysis systems will need to adopt its monitoring strategy accordingly.

6.4 Concluding thoughts

Malware is a problem we are likely to be dealing with for the foreseeable future. The effective countering of this phenomenon will not be possible without the appropriate tools and without the appropriate training of security personnel. As the complexity of our systems continue to increase, it will be increasingly difficult for people to grasp and identify the inherent security risks present in our systems. There are no easy solutions to this problem, as our systems have not been designed with security in mind and are becoming more complex to secure. As Brendan Dolan-Gavitt has very eloquently put it, we ”require deep changes to the way we construct systems, as we would need a way to tie its low-level implementation to a high-level semantic description of its runtime behavior in a way that is *verifiable* [...] by asking whether we can ensure that a program 1) says what it does; 2) does what it says; and 3) can prove it” [28].

Whether it is possible to construct such systems with today’s technology remains to be seen. In many respect, the fundamental reason why we are dealing with security problems is because of our use of general purpose computers. By design, these systems can perform arbitrary computations, which is the reason why it has been so economically

feasible to manufacture them on large-scale. On the flip side, this makes it difficult to differentiate between malicious and non-malicious computations.

Today arguably the most effective solution has been employing software white-listing to battle malware. By preventing software with an unknown hash to even start on the system, malware would arguably have a hard time getting started. The price of this security model however is to give up our ability to run arbitrary software without prior approval of the maintainer of the whitelist - be that the operating system vendor, the hardware vendor, or some other third party. Furthermore, the security provided by current implementations is limited as the underlying hardware can still execute any code; thus, all an attacker needs to find is a vulnerability in an already running software to obtain a foothold.

An alternative white-listing approach may be to build purpose-made chips designed to execute specific functions only, effectively moving away from general purpose computers. It is not hard to imagine, considering the popularity of the 3D printer movement today, that in the future we may be able to print our own chips and do so in a reasonably economical manner. If we succeed in that endeavor, attackers would be restricted to find ways to induce glitches in the hardware to subvert the computation flow (such attacks have already appeared against commodity hardwares today). Nevertheless, provided that the purpose-made chips would be more varied, targeted attacks would become increasingly more difficult.

In the interim, what we can do is to refine our existing layered, defense-in-depth security model and continue increasing the cost of breaking through all layers. Today deploying hypervisors for security purposes have already been effectively employed, as we have shown in this dissertation as well. On today's hardware a number of extra layers are also present that could be further used for this purpose. What must be realized however is that an extra layer only provides extra protection if it is implemented such that each layer has viewer and viewer attack vectors. Great care must be taken to restrict layers to only interact with code and data from layers immediately next to them, or we open a path for

malware to by-pass all other layers. Unfortunately, with the mixed definition of what we are trying to protect and against who, it is the tendency to move DRM software into the most privileged sections of our system to protect against the actual owner of the system. Such design decisions effectively undermine the defense-in-depth model of having layers to begin with, exposing the owner of the system to greater risks then would be warranted. Without the ability of the owner to opt-out from having such software pre-installed or disabling these systems, the risk of exposing the most privileged components of our system to malware will continue to be present, regardless how many layers we use.

Acronyms

ABI Application Binary Interface. 16

APC Asynchronous Procedure Call. 67, 68

API Application Program Interface. 24

APIC Advanced Programmable Interrupt Controller. 113

AV anti-virus. 83

CoW Copy-on-Write. iii, 13, 50, 71, 80, 82, 98

DKOM Direct Kernel Object Manipulation. ii, 19, 20, 24, 25, 73, 107, 108

DKSM Direct Kernel Structure Manipulation. 20

DMA Direct Memory Access. 110

EAT Export Address table. 24

EPT Extended Page Table. iii, 25, 61, 65, 82, 107–111, 119, 120

FMA forensic memory analysis. 107

GPA Guest Physical Address. 61, 104

GVA Guest Virtual Address. 61, 104

HIH High-Interaction Honeypot. iii, 11, 12, 30–33, 35–38, 40, 41, 44, 46, 48, 50

IAT Import Address table. 24

IDT Interrupt Descriptor Table. 60

LIH Low-Interaction Honeypot. 11, 12, 44

NAT Network Address Translation. 41

NX eXecute-Never. 61

OS Operating System. 57, 67, 107

PE Portable Executable. 68

PEB Process Environment Block. 68

PGE Page Global Enabled. 103

PTE Page Table Entry. 119

RVA relative virtual address. 73

SMC Secure Monitor Call. 111, 112, 120

SMI System Management Interrupt. 113–115

SMM System Management Mode. 15, 111, 113, 114

SMRAM System-Management RAM. 113, 114

SSDT System Service Dispatch Table. 77

STM SMM Transfer Monitor. 114, 115

TCB Trusted Computing Base. 110

TLB Translation Lookaside Buffer. 102, 103, 105, 106

TSC Time-stamp counter. 96

V2P virtual-to-physical. 60, 102

VM Virtual Machine. 12, 14, 16, 18, 19, 21, 22, 24, 25, 41, 51, 58, 59, 61, 66, 78, 83, 107, 110–112, 114, 115, 119

VMCS Virtual Machine Control Segment. 58–61, 106–108

VMI Virtual Machine Introspection. 16, 20–22, 59, 105, 107, 110–112, 117, 118

VMM virtual machine monitor aka. hypervisor. 14, 16, 22, 24, 57–59, 62, 106, 107, 111–113, 115

VPID Virtual Processor Identification. 105–107

VT VirusTotal. 78

Appendix A

Notes

[1]md5sum 13ce4cd747e450a129d900e842315328

[2]TDL4 md5sum a1b3e59ae17ba6f940afaf86485e5907

[3]TDL4 cryptbase.dll md5sum d39d6893117cf1a80c77de1f7ff3d944

[4]TDL4 syssetup.dll md5sum 9b6e5b9c0deb825d5aed343beb090853

[5]SpyEye2 md5sum 8e7c7dde842223bfa7d09680f9b74f5c

[6]Zeus md5sum ea039a854d20d7734c5add48f1a51c34

[7]Zeus msimg32.dll md5sum e051308c2f0c1b280514c99aabd36e34

[8]CryptoLocker md5sum 0246bb54723bd4a49444aa4ca254845a

[9]CryptoLocker carved from memory md5sum 67b2d99f5be8c76259159aab6b20d470

[10]CryptoLocker unpacked md5sum 0246bb54723bd4a49444aa4ca254845a

[11]Zusy md5sum bca0660d52f01c7e99b8a6378f436cb8

[12]Zusy dropped executable md5sum 0468b7bce4741d965c9ede5d5367993f

[13]MultiPlug md5sum 000472488152b778726c110e83239f9a

[14]Zusy dropped executable md5sum 8b9d09589579a5e61325a33ccecb5b72

[15]Zusy dropped 64-bit DLL md5sum 286b89c2f7ef8f21ea59ec803b634f21

[16]Zusy dropped 32-bit DLL md5sum e5af4aae60e3e80554bbc23dae088d84

[17]Sisbot.A md5sum 0108aa26cdce45318214278e3eaf730c

[18]Sisbot.A unpacked md5sum ac8bb161c73a1c37018cfbc49d02d3b2

[19]md5sum fe85ff523902c8ed4a328d7aa66b180a

Listing 1: _POOL_HEADER definition on Windows 7

```
typedef struct _POOL_HEADER {
    union {
        struct {
            ULONG32 PreviousSize : 8;
            ULONG32 PoolIndex : 8;
            ULONG32 BlockSize : 8;
            ULONG32 PoolType : 8;
        };
        ULONG32 Ulong1;
    };

    // This tag is used for scanning and fingerprinting
    ULONG32 PoolTag;

    union {
        struct _EPROCESS* ProcessBilled;
        struct {
            UINT16 AllocatorBackTraceIndex;
            UINT16 PoolTagHash;
            UINT8  _PADDING0_[0x4];
        };
    };
} POOL_HEADER, *PPOOL_HEADER;
```

Listing 2: Basic process creation on Windows using the CreateProcessA function

```
#include <windows.h>
void main(void)
{
    STARTUPINFOA si = {0};
```

```
    PROCESS_INFORMATION pi = {0};
    char *cmdline = "calc.exe";
    CreateProcessA(NULL,   // No module name (use command line)
        cmdline,           // Command line
        NULL,              // Process handle not inheritable
        NULL,              // Thread handle not inheritable
        FALSE,             // Set handle inheritance to FALSE
        0,                 // No creation flags
        NULL,              // Use parent's environment block
        NULL,              // Use parent's starting directory
        &si,               // Pointer to STARTUPINFO structure
        &pi);              // Pointer to PROCESS_INFORMATION structure
}
```

Listing 3: IDA Pro x86 disassembly of the basic process creation binary compiled with Visual Studio Community 2013

```
push    ebp
mov     ebp, esp
sub     esp, 98h
push    ebx
push    esi
push    edi
mov     dword ptr [ebp-44h], 0
push    40h             ; Size
push    0               ; Val
lea     eax, [ebp-40h]
push    eax             ; Dst
call    j__memset
add     esp, 0Ch
mov     dword ptr [ebp-54h], 0
xor     eax, eax
```

```
16 mov     [ebp-50h], eax
17 mov     [ebp-4Ch], eax
18 mov     [ebp-48h], eax
19 mov     [ebp+lpCommandLine], offset aCalc_exe ; "calc.exe"
20 lea     eax, [ebp-54h]
21 push    eax             ; lpProcessInformation
22 lea     ecx, [ebp-44h]
23 push    ecx             ; lpStartupInfo
24 push    0               ; lpCurrentDirectory
25 push    0               ; lpEnvironment
26 push    0               ; dwCreationFlags
27 push    0               ; bInheritHandles
28 push    0               ; lpThreadAttributes
29 push    0               ; lpProcessAttributes
30 mov     edx, [ebp-58h]
31 push    edx             ; lpCommandLine
32 push    0               ; lpApplicationName
33 call    ds:__imp__CreateProcessA@40 ; CreateProcessA(x,x,x,x,x,x,x,x,x,x)
34 xor     eax, eax
35 pop     edi
36 pop     esi
37 pop     ebx
38 mov     esp, ebp
39 pop     ebp
40 retn
```

Listing 4: IDA Pro x86-64 disassembly of the basic process creation binary compiled with Visual Studio Community 2013

```
1 push    rdi
2 sub     rsp, 0E0h
3 mov     dword ptr [rsp+70h], 0
```

```
4  lea     rax, [rsp+78h]
5  mov     rdi, rax
6  xor     eax, eax
7  mov     ecx, 60h
8  rep stosb
9  mov     qword ptr [rsp+58h], 0
10 lea     rax, [rsp+60h]
11 mov     rdi, rax
12 xor     eax, eax
13 mov     ecx, 10h
14 rep stosb
15 lea     rax, aCalc_exe  ; "calc.exe"
16 mov     [rsp+50h], rax
17 lea     rax, [rsp+58h]
18 mov     [rsp+48h], rax  ; lpProcessInformation
19 lea     rax, [rsp+70h]
20 mov     [rsp+40h], rax  ; lpStartupInfo
21 mov     qword ptr [rsp+38h], 0 ; lpCurrentDirectory
22 mov     qword ptr [rsp+30h], 0 ; lpEnvironment
23 mov     dword ptr [rsp+28h], 0 ; dwCreationFlags
24 mov     dword ptr [rsp+20h], 0 ; bInheritHandles
25 xor     r9d, r9d        ; lpThreadAttributes
26 xor     r8d, r8d        ; lpProcessAttributes
27 mov     rdx, [rsp+50h]  ; lpCommandLine
28 xor     ecx, ecx        ; lpApplicationName
29 call    cs:__imp_CreateProcessA
30 xor     eax, eax
31 add     rsp, 0E0h
32 pop     rdi
33 retn
```

Requests	Domain	Requests	Domain
421	s3.amazonaws.com	26	api.shuame.com
391	majuwe.com	25	n5.linkpc.net
206	api.mediaconfig.net	25	google.com
199	sendme9.ru	24	tracking.uniblue.com
172	softonic-analytics.net	23	www.secondofferdelivery.com
159	imp.myappz02.com	23	nowtake.me
123	stiekehelp.gameassists.co.uk	22	secure.pn-installer7.com
78	4threquest.me	21	service.downloadadmin.com
72	srv.desk-top-app.info	21	sendme8.ru
70	pastebin.com	21	secure.5-pn-installer.com
62	rghost.net	20	cdn.backupgrid.net
58	imp.myappz01.com	19	secure.pn-installer28.com
56	www.google.com	19	ilo.brenz.pl
51	data.dmccint.com	17	easysetupinstall.com
49	cms.dmccint.com	17	api.couponcute.com
48	install.oinstaller9.com	16	secure.oinstaller7.com
44	www.4threquest.me	15	www.djapp.info
44	stat.miniload.org	15	rp.baixakialtcdn2.com
42	url.hongdafood.cn	15	info.baixakialtcdn2.com
40	secure.oinstaller6.com	15	imp.myappz11.com
40	errors.crossrider.com	14	dl4.getz.tv
39	imp.softwareinstaller.org	13	youssef20.ddns.net
36	config.softwareinstaller.org	13	api.e54h3p0o.com
30	dl.iwin.com	12	saila2014.no-ip.biz
26	syscos18.ru	12	downloads.updatersoft.com

Table 10: Top 50 DNS requests

References

[1] AlienVault. Virtual machine detection tricks. `https://github.com/jaimeblasco/AlienvaultLabs/blob/master/malware_rulesets/yara/vmdetect.yar`, February 4 2014.

[2] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, October 2012.

[3] ARM. *ARM Architecture Reference Manual*, July 2012.

[4] Kurniadi Asrigo, Lionel Litty, and David Lie. Using vmm-based sensors to monitor honeypots. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 13–23. ACM, 2006.

[5] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds: real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 90–102. ACM, 2014.

[6] Sina Bahram, Xuxian Jiang, Zhi Wang, Mike Grace, Jinku Li, Deepa Srinivasan, Junghwan Rhee, and Dongyan Xu. Dksm: Subverting virtual machine introspection for fun and profit. In *Reliable Distributed Systems, 2010 29th IEEE Symposium on*, pages 82–91. IEEE, 2010.

[7] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Efficient detection of split personalities in malware. In *NDSS*, 2010.

[8] Ulrich Bayer, Imam Habibi, Davide Balzarotti, Engin Kirda, and Christopher Kruegel. A view on current malware behaviors. In *USENIX workshop on large-scale exploits and emergent threats (LEET)*, 2009.

[9] Michael Beham, Marius Vlad, and Hans P Reiser. Intrusion detection and honeypots in nested virtualization environments. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pages 1–6. IEEE, 2013.

[10] Boldizsár Bencsáth, Gábor Pék, Levente Buttyán, and Márk Félegyházi. Duqu: Analysis, detection, and lessons learned. In *ACM European Workshop on System Security (EuroSec)*, volume 2012, 2012.

[11] Robin G Berthier. *Advanced honeypot architecture for network threats quantification*. ProQuest, 2009.

[12] Bill Blunden. *Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*. Jones & Bartlett Publishers, 2012.

[13] Rodrigo Rubira Branco, Gabriel Negreira Barbosa, and Pedro Drimel Neto. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies, 2012.

[14] Jurriaan Bremer. Blackhat 2013 workshop: Cuckoo sandbox - open source automated malware analysis. `http://cuckoosandbox.org/2013-07-27-blackhat-las-vegas-2013.html`, 2013.

[15] Denis Bueno, Kevin J Compton, Karem A Sakallah, and Michael Bailey. Detecting traditional packers, decisively. In *Research in Attacks, Intrusions, and Defenses*, pages 184–203. Springer, 2013.

[16] bugcheck. Grepexec: Grepping executive objects from pool memory. *Uninformed*, 2006.

[17] Jamie Butler. Dkom (direct kernel object manipulation). *Black Hat Windows Security*, 2004.

[18] Jamie Butler and Peter Silberman. Raide: Rootkit analysis identification elimination. *Black Hat USA*, 47, 2006.

[19] Martim Carbone, Matthew Conover, Bruce Montague, and Wenke Lee. Secure and robust monitoring of virtual machines through guest-assisted introspection. In *Research in Attacks, Intrusions, and Defenses*, volume 7462 of *Lecture Notes in Computer Science*, pages 22–41. Springer Berlin Heidelberg, 2012.

[20] Peter M Chen and Brian D Noble. When virtual is better than real [operating system relocation to virtual machines]. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 133–138. IEEE, 2001.

[21] George Coker. Xen security modules (xsm). *Xen Summit*, pages 1–33, 2006.

[22] Robert R. Collins. The caveats of system management mode. `http://www.rcollins.org/ddj/May97/May97.html`, October 29 2014.

[23] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 289–298, New York, NY, USA, 2013. ACM.

[24] 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*, pages 51–62. ACM, 2008.

[25] Dionaea. catches bugs. `http://dionaea.carnivore.it`, November 20 2014.

[26] DNSchef. http://thesprawl.org/projects/dnschef, March 16 2012.

[27] B. Dolan-Gavitt, B.D. Payne, and W. Lee. Leveraging forensic tools for virtual machine introspection. Gt-cs-11-05, Georgia Institute of Technology, 2011.

[28] Brendan Dolan-Gavitt. http://www.cc.gatech.edu/~brendan/research.pdf, July 14 2015.

[29] Brendan Dolan-Gavitt. Graphical malware actuation with panda and volatility. http://laredo-13.mit.edu/~brendan/BSIDES_NOLA_2015.pdf, July 5 2015.

[30] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 297–312. IEEE, 2011.

[31] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 566–577. ACM, 2009.

[32] Brendan Dolan-Gavitt and Patrick Traynor. Using kernel type graphs to detect dummy structures. Technical report, Georgia Tech, 2008.

[33] Brendan F Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering for the greater good with panda. *Technical Reports, Department of Computer Science, Columbia University*, 2014.

[34] Maximillian Dornseif, Thorsten Holz, and Christian N Klein. Nosebreak-attacking honeynets. In *Information Assurance Workshop, 2004. Proceedings from the Fifth Annual IEEE SMC*, pages 123–129. IEEE, 2004.

[35] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.

[36] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2):6, 2012.

[37] Nelson Elhage. Virtunoid: Breaking out of kvm. *Black Hat USA*, 2011.

[38] ESET. Tdl4 rebooted. http://blog.eset.com/2011/10/18/tdl4-rebooted, June 9 2012.

[39] Elia Florio. When malware meets rootkits. *Virus Bulletin*, 2005.

[40] Fox-IT. Tilon/spyeye2 intelligence report. http://foxitsecurity.files.wordpress.com/2014/02/spyeye2_tilon_20140225.pdf, 2014.

[41] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is not transparency: Vmm detection myths and realities. In *HotOS*, 2007.

[42] Tal Garfinkel and Mendel Rosenblum. When virtual is harder than real: Security challenges in virtual machine based computing environments. In *HotOS*, 2005.

[43] Tal Garfinkel, Mendel Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, 2003.

[44] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. Sprobes: Enforcing kernel code integrity on the trustzone architecture. *arXiv preprint arXiv:1410.7747*, 2014.

[45] Jason Gionta, Ahmed Azab, William Enck, Peng Ning, and Xiaolan Zhang. Seer: practical memory virus scanning as a service. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 186–195. ACM, 2014.

[46] GRSecurity. Pageexec. https://pax.grsecurity.net/docs/pageexec.txt, December 30 2006.

[47] Zhongshu Gu, Zhui Deng, Dongyan Xu, and Xuxian Jiang. Process implanting: A new active introspection framework for virtualization. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*, pages 147–156. IEEE, 2011.

[48] Fanglu Guo, Peter Ferrie, and Tzi-Cker Chiueh. A study of the packer problem and its solutions. In *Recent Advances in Intrusion Detection*, pages 98–115. Springer, 2008.

[49] David Harley. http://www.welivesecurity.com/2012/02/02/tdl4-reloaded-purple-haze-all-in-my-brain/, February 3 2014.

[50] Takahiro Haruyama and Hiroshi Suzuki. One-byte modifications for breaking memory forensic analysis. *Black Hat Europe*, 2012.

[51] S. A. Hofmeyr, A. Somayaji, and S. Forrest. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.

[52] Greg Hoglund and James Butler. *Rootkits: subverting the Windows kernel*. Addison-Wesley Professional, 2006.

[53] Honeynet. The qebek project. https://projects.honeynet.org/sebek/wiki/Qebek, November 20 2014.

[54] Honeynet. The sebek project. https://projects.honeynet.org/sebek, November 20 2014.

[55] Xen Project Hypervisor. http://www.xenproject.org, July 14 2015.

[56] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3C: System Programming Guide, Part 3*, June 2013.

[57] Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E. Porter, and Radu Sion. Sok: Introspections on trust and the semantic gap. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 605–620, Washington, DC, USA, 2014. IEEE Computer Society.

[58] Xuxian Jiang and Xinyuan Wang. out-of-the-box monitoring of vm-based high-interaction honeypots. In *Recent Advances in Intrusion Detection*, pages 198–218. Springer, 2007.

[59] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 128–138. ACM, 2007.

[60] Stephen T Jones, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *USENIX Annual Technical Conference, General Track*, pages 1–14, 2006.

[61] Mikael Keri. Detecting dionaea honeypot using nmap. http://blog.prowling.nu/2012/04/detecting-dionaea-honeypot-using-nmap.html, April 3 2012.

[62] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barecloud: bare-metal analysis-based evasive malware detection. In *Proceedings of the 23rd USENIX conference on Security Symposium*, pages 287–301. USENIX Association, 2014.

[63] Peter Kleissner. The art of bootkit development. http://www.stoned-vienna.com/pdf/The-Art-of-Bootkit-Development.pdf, June 26 2015.

[64] Peter Friedrich Klemperer. *Efficient Hypervisor Based Malware Detection*. PhD thesis, Carnegie Mellon University, 2014.

[65] Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel. The power of procrastination: detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 285–296. ACM, 2011.

[66] Christian Kreibich, Nicholas Weaver, Chris Kanich, Weidong Cui, and Vern Paxson. Gq: Practical containment for measuring modern malware systems. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 397–412. ACM, 2011.

[67] Peter Kruse, Feike Hacquebord, and Robert McArdle. Threat report: W32.tinba (tinybanker) the turkish incident. http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp_w32-tinba-tinybanker.pdf, 2012.

[68] KVM. Kernel based virtual machine. http://www.linux-kvm.org, November 4 2013.

[69] Motive Security Labs. Motive security labs malware report h2 2014. Technical report, Alcatel-Lucent, 2014.

[70] Andrés Lagar-Cavilla and Andrew Cooper. Xen-devel: Handle resumed instruction based on previous mem_event reply. http://www.gossamer-threads.com/lists/xen/devel/347492#347492, September 11 2014.

[71] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M Rumble, Eyal De Lara, Michael Brudno, and Mahadev Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 1–12. ACM, 2009.

[72] John Leitch. Process hollowing. http://www.autosectools.com/process-hollowing.pdf, November 4 2013.

[73] Bin Liang, Wei You, Wenchang Shi, and Zhaohui Liang. Detecting stealthy malware with inter-structure and imported signatures. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 217–227. ACM, 2011.

[74] libguestfs. Library for accessing and modifying vm disk images. http://libguestfs.org, April 10 2012.

[75] libvirt. The virtualization api. http://libvirt.org, April 10 2012.

[76] LibVMI. http://libvmi.com, April 4 2015.

[77] Michael Ligh. Torpig vmm/idt signatures. http://www.mnin.org/write/2006_torpigsigs.pdf, 2006.

[78] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. Detecting environment-sensitive malware. In *Recent Advances in Intrusion Detection*, pages 338–357. Springer, 2011.

[79] malwr.com. Analysis results. https://malwr.com/analysis/MzhlZWI5ZjM2NWUwNDAzYjgxNDJmMjIwNzBlMGE2YjE/, July 13 2015.

[80] Pete Markowsky. Ring -1 vs. ring -2: Containerizing malicious smm interrupt handlers on amd-v. *ShmooCon*, 2010.

[81] McAfee. http://blogs.mcafee.com/mcafee-labs/conficker-worm-using-metasploit-payload-to- June 9 2012.

[82] Microsoft MSDN. Createprocess function. https://msdn.microsoft.com/en-us/library/windows/desktop/ms682425(v=vs.85).aspx, May 11 2015.

[83] Jose Nazario. Phoneyc: a virtual client honeypot. In *Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more*, pages 6–6. USENIX Association, 2009.

[84] Novetta. Operation smn - axiom threat actor group report. http://www.novetta.com/wp-content/uploads/2014/11/Executive_Summary-Final_1.pdf, November 2014.

[85] James S Okolica and Gilbert L Peterson. Extracting forensic artifacts from windows o/s memory. Technical report, DTIC Document, 2011.

[86] Mila Parkour. Contagio dump. http://contagiodump.blogspot.com, February 4 2014.

[87] Bryan D Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 233–247. IEEE, 2008.

[88] Bryan D Payne, MDP de Carbone, and Wenke Lee. Secure and flexible monitoring of virtual machines. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 385–397. IEEE, 2007.

[89] Gábor Pék, Boldizsár Bencsáth, and Levente Buttyán. nether: In-guest detection of out-of-the-guest malware analyzers. In *Proceedings of the Fourth European Workshop on System Security*, page 3. ACM, 2011.

[90] Roberto Perdisci, Wenke Lee, and Nick Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *NSDI*, pages 391–404, 2010.

[91] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Nitro: Hardware-based system call tracing for virtual machines. In *Advances in Information and Computer Security*, pages 96–112. Springer, 2011.

[92] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[93] Rapid7. Metasploit penetration testing software. http://www.metasploit.com, April 12 2012.

[94] Rekall. Memory forensics analysis framework. https://code.google.com/p/rekall, May 12 2014.

[95] Junghwan Rhee, Ryan Riley, Dongyan Xu, and Xuxian Jiang. Kernel malware analysis with untampered and temporal views of dynamic kernel memory. In *Recent Advances in Intrusion Detection*. Springer, 2010.

[96] Ryan Riley, Xuxian Jiang, and Dongyan Xu. An architectural approach to preventing code injection attacks. *Dependable and Secure Computing, IEEE Transactions on*, 7(4):351–365, 2010.

[97] Anthony Roberts, Richard McClatchey, Saad Liaquat, Nigel Edwards, and Mike Wray. Introducing pathogen: A real-time virtual machine introspection framework. Technical report, HP, 2013.

[98] T. Roy. x64 deep dive. `http://www.codemachine.com/article_x64deepdive.html`, June 26 2015.

[99] Paul Royal. Entrapment: Tricking malware with transparent, scalable malware analysis. *Blackhat EU*, 2012.

[100] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 289–300. IEEE, 2006.

[101] Joanna Rutkowska and Rafal Wojtczuk. Preventing and detecting xen hypervisor subversions. *Blackhat Briefings USA*, 2008.

[102] Andreas Schuster. Searching for processes and threads in microsoft windows memory dumps. *digital investigation*, 3:10–16, 2006.

[103] ShadowServer. The shadowserver foundation. `https://shadowserver.org`, February 4 2014.

[104] Monirul I Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 477–487. ACM, 2009.

[105] Asaf Shelly. Asynchronous procedure calls. `https://msdn.microsoft.com/en-us/library/windows/desktop/ms681951%28v=vs.85%29.aspx`, May 11 2015.

[106] Sophos. Troj/msil-kw. `http://www.sophos.com/en-us/threat-center/threat-analyses/viruses-and-spyware/Troj~MSIL-KW/detailed-analysis.aspx`, February 14 2014.

[107] Sherri Sparks and Jamie Butler. Shadow walker: Raising the bar for rootkit detection. *Black Hat Japan*, pages 504–533, 2005.

[108] Sherri Sparks and Shawn Embleton. Smm rootkits: A new breed of os independent malware. *Black Hat USA, Las Vegas, NV, USA*, 2008.

[109] Abhinav Srivastava and Jonathon Giffin. Tamper-resistant, application-aware blocking of malicious network connections. In *Recent Advances in Intrusion Detection*, pages 39–58. Springer, 2008.

[110] Sahil Suneja, Canturk Isci, Eyal de Lara, and Vasanth Bala. Exploring vm introspection: Techniques and trade-offs. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 133–146. ACM, 2015.

[111] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 256–267. ACM, 2015.

[112] Jacob Torrey. More: measurement of running executables. In *Proceedings of the 9th Annual Cyber and Information Security Research Conference*, pages 117–120. ACM, 2014.

[113] Jacob Torrey. More shadow walker: Tlb-splitting on modern x86. *BlackHat*, 2014.

[114] K.L. Tseng, B. Liu, R. Sood, M.R. Castelino, and M. Tallam. Determining policy actions for the handling of data read/write extended page table violations, June 27 2013. WO Patent App. PCT/US2011/067,038.

[115] VirusTotal. Free online virus, malware and url scanner. `http://virustotal.com`, February 4 2014.

[116] Sebastian Vogl and Claudia Eckert. Using hardware performance events for instruction-level monitoring on the x86 architecture. In *Proceedings of EuroSec'12, 5th European Workshop on System Security*. ACM Press, April 2012.

[117] Sebastian Vogl, Fatih Kilic, Christian Schneider, and Claudia Eckert. X-tier: Kernel module injection. In *Network and System Security*, pages 192–205. Springer, 2013.

[118] Sebastian Vogl, Jonas Pfoh, Thomas Kittel, and Claudia Eckert. Persistent data-only malware: Function hooks without code. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[119] Volatility. The volatility framework: Volatile memory artifact extraction utility framework. `https://www.volatilesystems.com/default/volatility`, January 15 2014.

[120] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C Snoeren, Geoffrey M Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 148–162. ACM, 2005.

[121] AAron Walters and Nick L Petroni. Volatools: Integrating volatile memory into the digital investigation process. *Black Hat DC 2007*, pages 1–18, 2007.

[122] Jiang Wang, Angelos Stavrou, and Anup Ghosh. Hypercheck: A hardware-assisted integrity monitor. In *Recent Advances in Intrusion Detection*, pages 158–177. Springer, 2010.

[123] Jiang Wang, Kun Sun, and Angelos Stavrou. An analysis of system management mode (smm)-based integrity checking systems and evasion attacks. *George Mason University Department of Computer Science Technical Report*, 2011.

[124] Y-M Wang, Doug Beck, Binh Vo, Roussi Roussev, and Chad Verbowski. Detecting stealth software with strider ghostbuster. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 368–377. IEEE, 2005.

[125] Filip Wecherowski. A real smm rootkit: Reversing and hooking bios smi handlers. *Phrack Magazine*, 13(66), 2009.

[126] Andrew White, Bradley Schatz, and Ernest Foo. Integrity verification of user space code. *Digital Investigation*, 10:S59–S68, 2013.

[127] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. *Security & Privacy, IEEE*, 5(2):32–39, 2007.

[128] Carsten Willems, Ralf Hund, and Thorsten Holz. Cxpinspector: Hypervisor-based, hardware-assisted system monitoring. Technical report, Ruhr-Universitat Bochum, 2013.

[129] Rafal Wojtczuk and Joanna Rutkowska. Attacking smm memory via intel cpu cache poisoning. *Invisible Things Lab*, 2009.

[130] Glenn Wurster, Paul Van Oorschot, and Anil Somayaji. A generic attack on checksumming-based software tamper resistance. In *Security and Privacy, 2005 IEEE Symposium on*, pages 127–138. IEEE, 2005.

[131] James Wyke. The zeroaccess rootkit. `http://sophosnews.files.wordpress.com/2012/04/zeroaccess2.pdf`, 2012.

[132] Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on*, pages 297–300. IEEE, 2010.

[133] Dennis Yurichev. *Reverse Engineering for Beginners*. http://beginners.re, 2015.

[134] Fengwei Zhang, Kevin Leach, Kun Sun, and Angelos Stavrou. Spectre: A dependable introspection framework via system management mode. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2013.

[135] HongWei Zhou, Xin Wu, WenChang Shi, JinHui Yuan, and Bin Liang. Hdrop: Detecting rop attacks using performance monitoring counters. In Xinyi Huang and Jianying Zhou, editors, *Information Security Practice and Experience*, volume 8434 of *Lecture Notes in Computer Science*, pages 172–186. Springer International Publishing, 2014.