Maitland: Analysis of Packed and Encrypted Malware via Paravirtualization Extensions

by

Christopher Adam Benninger
B.Sc., University of Victoria, 2010

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Maitland: Analysis of Packed and Encrypted Malware via Paravirtualization Extensions

by

Christopher Adam Benninger
B.Sc., University of Victoria, 2010

Supervisory Committee

Dr. Y. Coady, Co-Supervisor
(Department of Computer Science)

Dr. S. Neville, Co-Supervisor
(Department of Electrical and Computer Engineering)

Dr. P. McGeer, Departmental Member
(Department of Computer Science)

Mr. M. Salois, Additional Member
(Defence Research & Development Canada)

Supervisory Committee

Dr. Y. Coady, Co-Supervisor
(Department of Computer Science)

Dr. S. Neville, Co-Supervisor
(Department of Electrical and Computer Engineering)

Dr. P. McGeer, Departmental Member
(Department of Computer Science)

Mr. M. Salois, Additional Member
(Defence Research & Development Canada)

ABSTRACT

Malicious software (*malware*) attacks are an ever-increasing cyber-security problem. One reason for this trend is the widespread adoption of *packing* technology as a way to mask the semantics of binary instructions, hiding them from detection. Packing is so successful that it is estimated 70-80% of malicious programs utilize it to avoid detection [1]. The popularity of virtualization provides new tools for dealing with this threat. Researchers have successfully used facilities provided by virtualization to develop new ways of detecting and analyzing packed and encrypted malware. Methods like these typically require changes to the virtualization platform, making them difficult to deploy as well as hard to reuse. This thesis presents Maitland, a proof-of-concept unpacking system which achieves similar functionality to existing research, using paravirtualization extensions instead of requiring changes to the hypervisor. During our experiments, Maitland successfully exposed instructions in software that was packed by the *UPX* and *gzexe* packers. Maitland's avoidance of changes to the hypervisor means it is better suited for quick deployment in a cloud environment.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank:

my supervisors Yvonne Coady and Stephen Neville for their encouragement, advice, and energy.

my lab mate Chris Matthews for his patience and mentorship.

my family for their constant, enthusiastic, and unconditional support.

Dedication

To my grandfather, who taught me to cherish my curiosity and to never give up trying to understand.

Somewhere, something incredible is waiting to be known.
Carl Sagan

Chapter 1

Introduction and Related Work

The software industry has matured and adapted over the years, creating more efficient, resilient, and feature-filled software systems. But as the software industry has grown and advanced, so have people looking to exploit it. The authors of malicious software commonly referred to as *malware* have also adapted their strategies.

This chapter begins by surveying the state of malware and malware detection methods in a standard OS environment. We continue by providing some required technical background before discussing how virtualization changes the problem of malware and malware detection. At the end of the chapter, we examine the problem this thesis attempts to solve and provide some properties that a solution might have.

1.1 Malware

The term 'malware' is used to refer to any type of malicious software. This includes *Viruses*, *Worms*, *Trojans*, *Rootkits*, etc. Malware takes advantage of vulnerabilities in software to infect systems for various purposes. Creating and using malware is becoming increasingly lucrative as the world relies more heavily on computer systems [2]. The fact that the average person is aware (at least in part) of the dangers is yet another signal of how serious the problem is [3].

1.2 Malware Detection Methods and Techniques

Many techniques for detecting, neutralizing and analyzing malware exist and are being used in practice. There are two main families of analysis techniques. These

are *Static* and *Dynamic* malware analysis. Both methods provide their own set of strengths and weaknesses while their success hinges largely on the type of malware involved.

1.2.1 Static Analysis

Static malware analysis can be loosely defined as searching for malicious intent within a binary executable file [4]. Static analysis has been the primary method of malware detection for many years. This method looks at patterns of instructions within a program's binary executable image. *Binary signature analysis* is the most common form of static malware analysis. Binary signature analysis involves generating a *signature* or model which represents a particular piece of malicious behavior (or code) and some variations of it. The signature is then used to characterize that malicious behavior. Anti-malware vendors typically maintain large databases of precomputed signatures [5]. Malware detection tools access these databases to search for known signatures among the executable binary images and sometimes program memory space on a system. When a signature is identified within a program, that program is considered to be a potential variant of the malware family the signature represents. A signature must be created in advance before it can be used by an automated tool to scan for a known malware variant [6]. Thus, binary signature-based detection methods cannot detect zero-day attacks.

A zero-day attack is a malware infection by a previously unknown malware variant usually exploiting a new and unknown vulnerability. What this means is that no signature will have been created for a new variant and therefore, any system using purely signature detection will be unable to detect such an attack. Zero-day attacks are particularly effective because there is no standing automated defense against them. Usually these attacks are dealt with by deploying a patch to fix the vulnerability which takes time to build, test and deploy.

Signatures can take many forms but are frequently nothing more than a *regular expression* which can express a variation of sequences of different pieces of binary. Creating signatures can be done in several different ways and there is a large body of work looking at the best way to create them to optimize for best coverage and re-usability [7–10]. Signature creation is often manual, therefore, it is extremely valuable for anti-malware vendors to find a way to better automate the creation of

good signatures [11–13]. Moreover, the rapid rise in the number of malware variants provides further motivation [14, 15].

There are other approaches to static analysis. These approaches often utilize principles from data-mining and statistics to model the patterns of data within a binary executable [16–20]. Methods such as these attempt to make sense of binary code from a more theoretical perspective. They look at patterns of program code and attempt to understand the semantics of that code without executing it. Others simply extend the use of signatures by supplementing them with other approaches.

The use of signatures is generally considered less resistant to packing and encryption techniques than dynamic methods [21] resulting in inconsistency and ineffectiveness against them. In a study by Christodorescu et al., it was found that the top three anti-malware vendors had different and overlapping signatures for the same malicious executable [21]. This created a significant variation in performance between each vendor's product. Because of this, modern tools appear to be migrating to the usage of a combination of dynamic and static analysis.

1.2.2 Dynamic and Behavior-Based Analysis

Dynamic techniques for malware detection and analysis attempt to solve the problem from a different perspective. Although relatively successful and able to solve several of the problems which hinder static analysis relative to packing and encryption, purely dynamic approaches introduce a set of their own pitfalls.

Dynamic analysis focuses on observing a program's behavior as it runs. Rather than attempt to detect malicious intent before run-time, dynamic analysis systems monitor running programs for malicious behavior [22]. The benefit to this approach is that one doesn't have to guess whether the program will do something malicious or not, but can observe it. In reality though, defining malicious behavior is itself not always so clear-cut. The resulting high number of false-positives over static methods are one of the issues researchers face when developing new dynamic analysis techniques [23]. However, because this method no longer depends on preemptive knowledge of a malicious program, dynamic analysis has the potential to detect some forms of zero-day attack.

Monitoring a process' environment for certain properties is the most common way dynamic analysis tools track process behavior. This can (and does) have an adverse effect on performance [24]. A common approach to this type of monitoring is to run a

program inside an emulated environment because of the extra monitoring capabilities available. Emulation typically does not perform particularly well though, especially while simultaneously handling and processing events occurring within the emulation platform.

An analysis tool will allow the program to execute until something suspicious happens. Unfortunately, suspicious activity often consists of a sequence or combination of unsuspicious activities, meaning that a malicious program can sometimes manage to perform some or all of its malicious task before it is detected. Additionally, emulation is detectable. Because dynamic analysis is only able to analyze the execution trace which actually occurs, a malicious executable might avoid doing anything to indicate malicious intent when it detects its environment is being emulated. This would result in a false-negative.

An example of such a system can be found in CWSandbox [22]. This system attempts to take the next step toward a completely automated method for detection, requiring little or no human intervention. CWSandbox does not use a newly devised technique for dynamic analysis, but a unique and simple way to execute a combination of known ones while acknowledging the trade-offs between them. CWSandbox expects a human to always be close at hand, generating a report that is used to filter or audit decisions which were made automatically.

Although dynamic analysis techniques are often considered to be more resilient to packing and encryption, Moser et al. demonstrate that there are weaknesses to be exploited when the methods used by dynamic analysis tools are understood by malware writers [25]. Using *opaque constants* the authors tailor a technique which is designed to work against dynamic analysis approaches with high effectiveness.

Not all dynamic analysis tools are designed to replace static methods. In a paper by Bayer et. al, the authors present a system which is designed to automatically cluster and group together malware with similar behavior [26]. The goal is to help malware analysts by reducing much of the work of correlating between malware samples. As there is a human in the loop, if a sample cannot be grouped with high accuracy, a human will eventually get around to seeing it. Thus the penalty of failure is not as high. Because of it, a system such as this one will generally favour false negatives over false positives unlike most other dynamic approaches.

In a paper analyzing some highly-successful research for malware detection, Preda et al. present a set of proofs which they believe show that neither static signature

analysis nor dynamic behavioral analysis techniques are complete [23]. This is further evidence that a hybrid approach is likely the best choice moving into the future.

1.3 Packing and Encryption

Packing is a strategy used by malware writers to mask the meaning of their malicious software in an attempt to foil signature-based analysis; it has proven relatively effective [27]. The problem of packed malware is likely the biggest one the anti-malware community has ever faced [28].

A packing operation compresses an executable, masking it's behavior and producing a packed version of the original executable. This interferes with most static analysis of the resulting file.

Recent studies have estimated that at least 70% of new malware variants utilize some form of packing to defend against detection [1,29,30]. Symantec's top malicious code family of 2010, W32.Sality [31], utilized packing, as did the W32.Stuxnet [32] worm which infected several uranium enrichment facilities in 2010 and is considered one of the most expensive and complex pieces of malware ever created. It exploited 4 of the total 14 zero-day vulnerabilities discovered in 2010 [15].

Although packing is commonly used by malware writers, there are legitimate uses of packing. Packing is sometimes used to protect proprietary applications like computer games against reverse-engineering or piracy [1]. However, it has been estimated to be less than 35% (some estimates around 5%) of all packing use is legitimate [33] [1].

1.3.1 Packers

A *packer* is a packaging tool designed to perform compression on an executable program, with the added side-effect that much of the internal meaning of the program is hidden. A *packed* executable will typically look either normal, or entirely undecipherable at. At run-time however, the executable will *unpack* itself and execute the packed segments of program code. Packers are generally used by malware writers as an easy way to hide their software from standard signature detection.

There are many variations of packers to be found in the wild. Some may perform simple encoding, while others might utilize complex encryption or download instructions from a remote host. Regardless of the means however, the goal is essentially

the same: modify the external appearance of program code without changing its semantics, usually to hide malicious intent.

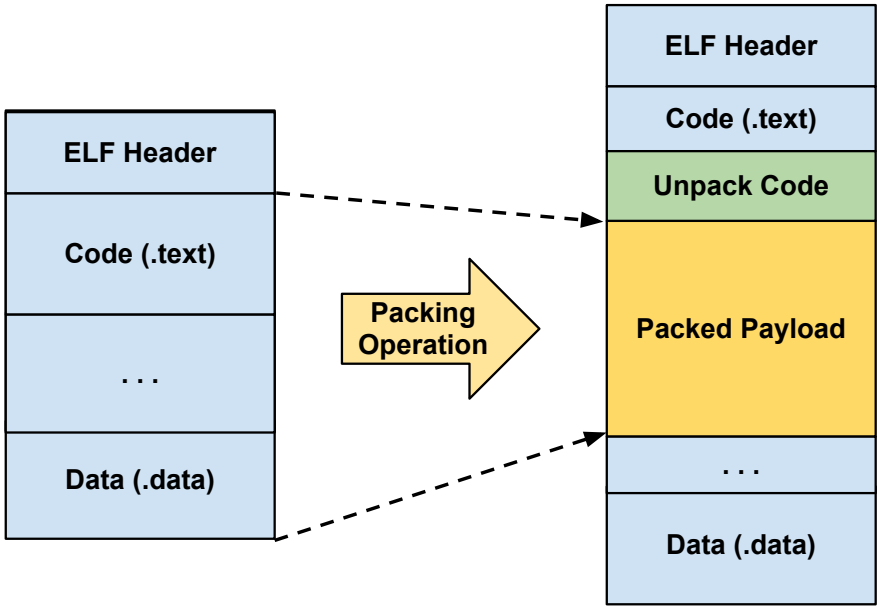

Figure 1.1: Example of a simple packer working on an ELF binary.

Many of the popular packers compress or encrypt a malicious executable's instructions using any number of algorithms. The resulting packed binary is then used to create a new executable file. Finally, stand-alone instructions to unpack/decrypt this data (also known as a *stub* [34]) are embedded in the output executable image making the program capable of unpacking itself. When the packed executable file is run, the unpacking instructions are executed. The instructions unpack the binary payload and pass the thread of execution to the original entry-point. From here, the original malicious code is executed. Only after unpacking and before executing the payload are the program's true intentions clear.

Once a specific variant of malware has been identified and signatures generated by an anti-malware vendor, it is relatively easy to identify. The problem lies in malware writers' ability to quickly and easily repack the binary, creating a new variant and forcing the vendor to create a new signature for the new binary. The vendor must now store multiple signatures for what is semantically the same binary code [27,33,35]. The effort involved in reverse engineering and signature generation is disproportionately high when compared to the creation of new malware variants using this technique. Additionally, some packers are polymorphic, meaning they will never produce the

same packed binary file twice given the same input. The result is that the list of signatures required to detect a particular variant comprehensively grows rapidly [27]. It has also been demonstrated that traditional static analysis is at least an NP-hard problem [25].

It has also become popular for malware writers to add many layers of packing or encryption. This is commonly referred to as *multi-round* or *multi-layer* packing. It is the process of recursively applying one or more packing operations to program code. These layers (or rounds) can also be done using entirely different packers for each layer. By alternating packers and their orders of execution, a malware writer can easily automate the creation of semantically-identical variations of their malware. This technique is used often to make reverse engineering and detection of malware much harder at a very small cost.

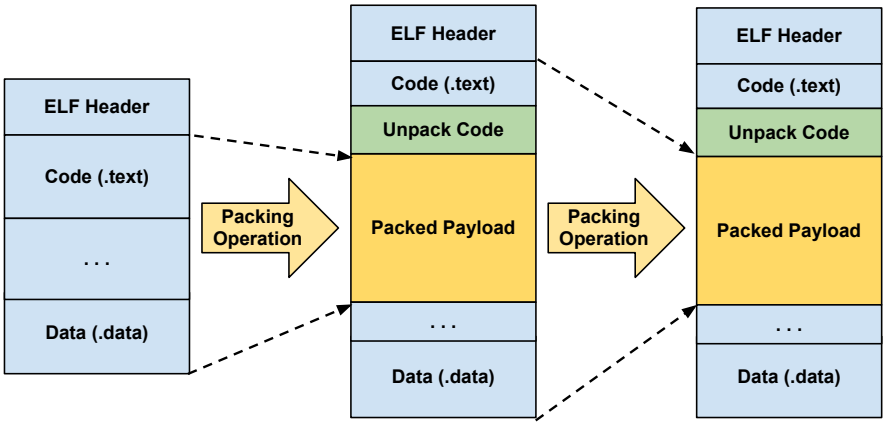

Figure 1.2: Example of multiple packing layers.

A large variety of different packers and packing techniques exists. There is no way to know how many packers exist, as new ones are created on an almost daily basis [28]. There are some more common packers however which are used by large percentages of malware writers. Some of these include *Ultimate Packer for eXecutables* (*UPX*), [36], *Themida* [37] and *ASPack* [38].

1.3.2 Unpackers

In response to the increasing use of packers, researchers and commercial vendors develop unpackers. Unpackers, like packers themselves are widely varied, many of them being specific enough to reverse only one packing technique. The existence

of and need for so many such entirely different unpackers hints at the scale of the problem. Most commercial anti-malware vendors and research labs have full time engineers building unpackers for malware variants as they are discovered [33].

Another, less obvious problem is the non-trivial task of detecting and identifying which packer was used to pack a binary file [33]. Tools such as PEid [39] exist to do just this. Some packing tools go through extreme lengths to confuse packer identification by doing things like including code segments from other packers among their own code.

1.3.2.1 Generic Unpackers

Although many unpackers are built to unpack binaries packed by a specific packer, generic unpackers do exist. A generic unpacker's goal is to unpack any arbitrary packed binary regardless of what was used to pack it. These tools are generally more complex than individual unpackers and building one requires understanding of how the common packing technologies work as well as having more control of the environment it works in. Several approaches have been published with varying degrees of overall success. It is fair to say that many generic unpackers are able to deal with multiple packer families, but not all, usually being most effective with a specific subset of packing methods.

Christodorescu et al. present a creative technique which involves identifying and isolating instructions commonly used while performing an unpack/decryption loop in an executable file [40]. This approach is a hybrid between static and dynamic analysis as it examines an executable file statically while attempting to understand the semantics of the instructions it examines.

A generic unpacker called *PolyUnpack* by Royal et al. at Georgia Tech, explores the virtues of a hybrid static-dynamic approach to malware analysis [41]. PolyUnpack extracts packed executables by comparing running state with a static code model of the program. This static code model is constructed before execution and then is later used to compare against each instruction as it is executed by the CPU. If an instruction somehow changes the existing code model, it is considered suspicious. The result is a tool to aid malware detectors break through packing and reduce false negatives. PolyUnpack is essentially stepping through a program, much like a debugger. The performance penalty for this type of approach is significant.

Table 1.1: Detection techniques used by recent approaches

System	Detection Method
PolyUnpack [41]	Instruction stepthrough
OmniUnpack [42]	Dirty page tracking
Renovo [43]	Dirty page tracking Syscall tracking
Justin [33]	Dirty page tracking

OmniUnpack is another tool released to deal with the problem of unpacking [42]. We based much of our design of Maitland on OmniUnpack due to its generic effectiveness. OmniUnpack monitors page-level memory writes and executions. If a dirty-page is executed, OmniUnpack considers this a potential unpacking operation. A heuristic is then used to approximate the completion of the unpack operation before invoking analysis of the program. This heuristic is key, as it allows OmniUnpack to significantly improve performance by invoking a malware detector a fraction of the number of times it otherwise would. In this way, OmniUnpack offers valuable modifications to an approach originally taken by other researchers.

Kang et al. have a system called *Renovo* which also extracts packed executables using the method of monitoring for dirty-page execution [43]. Renovo is different however in that it uses a custom emulator called TEMU (based on QEMU) to both watch for dangerous system calls and to isolate their analysis tool from the program being analyzed. They use *shadow memory*, a technique used in some paravirtualized hypervisors to provide isolation as well as added control regarding the memory space of a given program. To improve performance of emulating individual instructions, Renovo processes instructions in larger, contiguous segments rather than individually.

Justin is a similar system to Renovo and OmniUnpack in that it scans memory space when a program executes a dirty page [33]. What is interesting is the way it monitors events. Rather than hook the OS to inject fault-handler code or run programs in an emulator, it inserts its own fault-handler in the header of each binary it monitors. This removes much of the monitoring code of Justin out of kernel-space, reducing the privilege of a portion of the code base by placing it in user-space, but also making it must easier to tamper with.

The systems discussed here lay the groundwork for the techniques used in many of the ones discussed later. The main difference in relation to this thesis is that they do not apply specifically to a virtualized environment.

1.3.3 Metamorphic Malware

Metamorphic malware [44] is another type of malware that masks it's intent. Metamorphic malware re-writes itself each time it is deployed using different techniques. Its instructions and syntax change each time it is run, but its semantic meaning remains the same. This results in malware that is hard to detect using signature or pattern matching approaches. The techniques used to re-write these programs vary greatly in complexity and can be anything from adding dummy code and randomizing used registers to modify high-level program logic [45]. There is a large body of research looking at these types of attacks, but they are not the focus of this thesis.

1.3.4 Reverse Engineering

Reverse engineering is another term for manual analysis. It is how anti-virus vendors identify and relate malware variants to eventually create signature databases and anti-malware tools. Today, reverse engineering is the prevalent method of combating widespread malware infection. Much of the leading work focuses on incrementally automating more and more of the process to reduce the manual labor involved for analysts [4].

TTAnalyze aids engineers and analysts who reverse engineer malware by automating as much of the process as possible in an effort to combat the growing rate of new malware variants [3].The system runs an executable in an emulator but keeps track of all Windows API and system calls. Once execution is complete, it builds an elaborate and detailed report for an analyst to examine. This eliminates a significant portion of the work for a human as many executables can be run in batch and can be correlated together to find redundancy and similarity.

In a paper by Kreugel et al., a technique for disassembly is given [46]. By experimenting with static disassembly, researchers are able to reconstruct the control-flow of a packed binary. This makes both manual and partially automated analysis much easier.

1.4 Understanding Relevant Subsystems

Many common methods used to detect and analyze malware involve evaluating the states of subsystems in the target environment. Here we discuss some of the system internals which are relevant to approaches discussed so far as well as work presented later in this thesis.

1.4.1 Memory Management in Linux

The Linux kernel provides a convenient contiguous memory space for individual programs to use. This makes it easier for programmers to build applications by eliminating the complexities of dealing with physical memory directly. What is also provided is isolation between different program's memory spaces. Not only does this allow the kernel to enforce security and isolation restrictions, but also allows for some optimization to be done behind the scenes.

1.4.1.1 Paging

Physical memory from the perspective of the operating system kernel consists of a sequence of *frames*. The basic construct used for providing a memory abstraction by the kernel is a *page*. A page is a kernel structure which represents a physical frame and is used by the kernel to keep track of the frame's properties and the data stored within it. The size of a frame/page varies depending on the architecture and operating system but on 64-bit Linux is usually 4096 bytes. A process' memory space is made up of a collection of pages (may not be contiguous themselves) each with a unique page number. When a program in user space performs an operation on a memory address, the kernel decodes the address revealing the *Page Frame Number* or *PFN* and an offset into that page where the address points. With this information it finds the corresponding frame and can then perform the operation. This translation from memory address to page is done by the kernel with some help from the *Memory Management Unit* (*MMU*). The kernel must keep track of which pages make up different memory segments belonging to a process. It does this by maintaining a list called a *page table*.

1.4.1.2 Page Tables

A page table is a mutli-level tree structure specific to a given process. The depth of the page table depends on the architecture, but in 64-bit Linux (2.6.32 kernel) the page-table contains 4 levels [47]. The top level is called the *Page Global Directory* or *PGD*. The PGD is a list of references to *Page User Directories* (*PUDs*) which contain further references to *Page Middle Directories* (*PMDs*). Each PMD contains a list of *Page Table Entries* (*PTEs*) which refer to actual memory pages and contain a number of flags and properties associated with those pages [48].

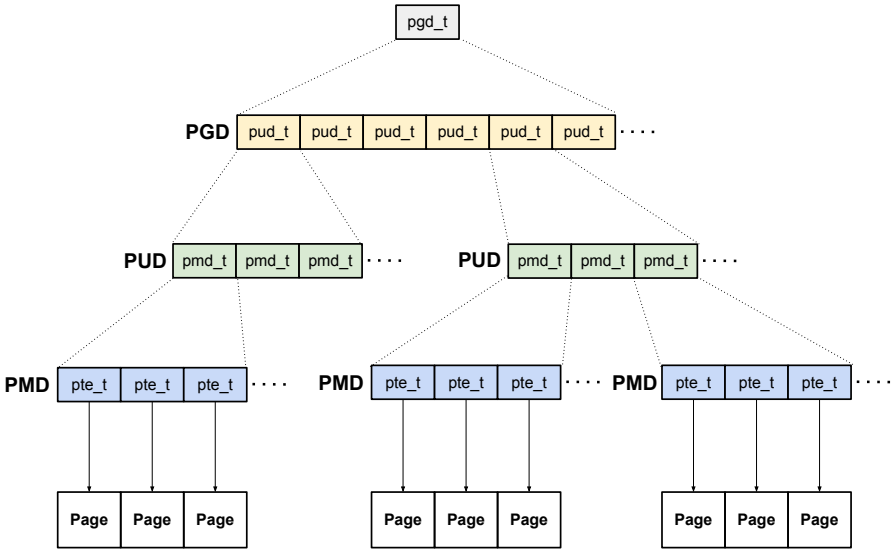

Figure 1.3: Example of a process' page-table on 64-bit Linux.

A user-space memory address contains an index for each level of the page-table. The kernel takes the index from the address and follows the tree down to find the page (and offset) associated with that address. The kernel uses the MMU for much of this work as it can be done very quickly in hardware [48].

1.4.1.3 Page Faults

A page fault is an interrupt which occurs when an illegal operation on a page or an operation which violates certain properties of a page is attempted. This facility is used to enforce read, write, execute properties on pages and is also used by the paging system to provide virtual memory. For example, the kernel typically will not allocate

actual frames of memory to a page-table entry until the owner process attempts to write to it. When this happens, a page fault will occur and the kernel will intercept it, allocate the frame and then continue execution normally. This helps with reducing memory utilization by not allocating memory which is never used. The kernel also uses this facility for *Copy on Write* (CoW) and for providing swapping features.

1.4.1.4 Memory Management Unit (MMU) Updates

An MMU update occurs when a change to the page table structure of a specific entry is done. Because the page structure or a page's property has changed, the MMU must be notified so it can update its references and keep functioning properly.

1.4.2 Virtualization

When an OS is virtualized, internal operation of some subsystems changes. There are also new subsystems added to facilitate virtualization. Here we provide some background and describe some of the relevant changes to existing internals as well as subsystems specific to virtualization.

1.4.2.1 Virtual Machines

A *virtual machine* or *VM* is an isolated software machine which resides within a virtualization layer. This virtualization layer provides an interface to hardware and other resources, multiplexing them and allowing multiple VMs to share them. An OS running in a VM may not be aware it is in a VM.

An OS being aware that it is virtualized or not is generally referred to as *paravirtualization* (PVM) and *hardware virtualization* (HVM) respectively [49]. A hardware-virtualized VM is unaware it is virtualized, making it possible to run nearly any OS without modification. Theoretically, any OS can be virtualized. A paravirtualized OS is one which is made aware it is running in a VM. This allows the OS to contain optimizations and short-cuts which improve performance and add features. The downside is that the OS must be modified to do this, which generally requires access to the OS source code.

Running software in VMs provides several benefits. These include security and performance isolation between VMs, multiplexing of hardware leading to improved utilization and the ability to move and duplicate VMs on the fly.

1.4.2.2 Hypervisors

A *hypervisor* (often referred to as a *Virtual Machine Monitor* or *VMM*) is an operating system which provides a virtualization layer for VMs. The hypervisor provides hardware abstraction and enforces security for VMs running inside of it. There are a number of existing hypervisors in use today, both proprietary and open-source. Most hypervisors utilize CPU features to improve the performance of VMs while some use a special software interface. A requirement of paravirtualization is to provide a special software interface for a VM to utilize directly [49].

Xen is a mature hypervisor, popular with academia and software vendors due to it being open source technology [50]. It is capable of multiplexing hardware between multiple VMs running different OSes simultaneously. Xen supports paravirtualized as well as non-paravirtualized VMs through hardware virtualization extensions.

1.4.2.3 Paravirtualization

Xen's main focus is on paravirtualized guests. There are certain operations that facilitate virtualization but are costly in terms of performance. One example might be creating the illusion that a VM has privileged access to its own memory. An OS inside a VM might normally expect full control of a system's memory. However, the hypervisor cannot allow this in order to enforce security and performance isolation. So the hypervisor creates the illusion of full control. It gives the VM only read and execute access to the memory pages containing its page structures. When the VM attempts to modify data on these pages, the hypervisor traps the operations and performs some enforcement and validation in the background without the VM being made aware.

A paravirtualized OS within a VM is modified so that a chosen number of these costly operations are bypassed. Comparatively instead of trying to write directly to these protected pages, a paravirtualized VM will simply request the hypervisor to do it on its behalf, avoiding the step of having the hypervisor trap it. These sorts of small changes can make a significant difference in the performance within a VM, and in achieving optimal hardware utilization for multiple VMs [51].

A paravirtualized OS makes requests to a hypervisor via an *Application Binary Interface* or *ABI*. On Xen, a call to this interface is called a *hypercall*, and is similar to the *system call* or *syscall* interface used to request kernel operations from userspace on Linux [51].

1.4.2.4 XenBus, XenStore and Event Channels

To facilitate some of the more advanced features available in a paravirtualized environment, Xen provides some interesting components. Two of these components are *XenStore* and *XenBus*.

XenStore is a simple filesystem-like database which is managed by Xen and can be used by VMs to store information and to communicate with each other [51]. The information contained in XenStore is in the form of hierarchical, key/value pair strings. The database contains information regarding each running VM and enforces a simple permissions system. This system is then used to enforce that each VM has it's own sandbox in which to store information. It can also be used to share information with other VMs.

XenBus is a bus-like abstraction which can be used for *interdomain* (inter-VM) communication. XenBus is part of the access mechanism used to interface with XenStore.

In order to provide these relatively high-level tools, Xen utilizes some low-level functionality as building blocks. One such important building block is a tool called an *event channel*, an interdomain software interrupt. Event channels are used heavily in nearly all component provided by Xen as they allow a convenient way to handle events and emulate hardware interrupts for VMs. They can be used by one VM to register for events occurring in another.

1.4.2.5 Grant Tables

Introspection is a term used in relation to virtualization to described the action of inspecting the internals of a VM from the outside [52]. Because Xen orchestrates the memory system used by VMs it has full access to any and all information contained inside the VM. However, it is sometimes necessary to allow VMs to share this information with each other in a structured way.

The *grant tables* system provided by Xen to paravirtualized VMs is the primary method VMs use to share information directly between each other. When sharing information via XenStore is not ideal and a lower-level, more direct form of sharing is needed, grant tables can be used. This system allows a VM to grant, specific pages to specific VMs and specify the level of access given for each. Each shared page has a unique identifier called a *grant reference* which must be used by other VMs to refer

to that page. VMs can share large segments of memory, or small blocks for nearly any case. Xen's grant tables are also used in the implementation of XenStore.

1.4.2.6 Split Drivers

To facilitate paravirtualization outside of the standard kernel source, Xen provides tools for building what are called *split drivers*. A split driver is a kernel driver which is split in two. The *frontend* resides in an unprivileged guest VM, and the *backend* reside in a privileged VM (referred to as "Domain 0" or "dom0"). The frontend is typically a lightweight interface to functionality or hardware which is indirectly provided by the backend. What this does is allow the guest VM to treat the frontend of the driver as if it were a normal hardware interface driver, while the privileged VM provides access to hardware indirectly, enforcing permissions and multiplexing access as it sees fit. This can be used for security isolation, resource allocation, quotas, etc. These two endpoints communicate primarily via event channels and XenStore. This method is used to generalize the paravirtualized kernel modifications and provides a convenient abstraction for porting frontend drivers to new platforms.

1.4.3 Memory Management in Xen

In order to provide isolation between virtual machines, Xen creates another layer of abstraction. Xen uses the same approach for isolating paravirtualized VMs as a normal OS uses to isolate processes.

1.4.3.1 Paging with Xen

With paravirtualized guests, Xen provides VMs only read access to their page tables. Modifications to pages containing related data structures must be explicitly requested through Xen. Because Xen is primarily a paravirtual hypervisor, this is done by running a modified OS in each VM. This works for paravirtualized guests, but for other types of guests Xen offers a couple of alternate memory management options.

One such option is called *Writable Page Tables*. This option provides a guest OS the illusion that it has full access to its page tables. However when an access occurs, Xen traps this operation and validates it behind the scenes. If validation passes, the hypervisor completes the operation and allows the VM to believe its operation completed normally. Thus, no paravirtualization modifications are required in the

guest OS. This is Xen's default method of operation for non paravirtualized guests. Trapping these operations is expensive, creating a performance cost.

A third option Xen provides is called *Shadow Page Tables*. When in this mode, Xen maintains its own duplicate copy of a guest's page tables. When a VM makes a change to its page table, these changes are propagated to Xen's copy and then validated. The reverse is also true, when Xen makes a modification, these changes can be propagated to the guests copy. In reality, the 'real' version of the page table is Xen's copy. Xen's version can have the true settings, while the copy inside the guest creates the illusion of full access. Shadow page tables are not commonly used during normal operation, but are used for features such as live migration of a VM across the network.

1.4.3.2 Pseudo-Physical Frames

In order to provide true isolation, Xen inserts a layer of abstraction underneath a guest VM in order to multiplex resources. A regular OS, works with PFNs which are then mapped to *Machine Frame numbers* (*MFNs*) by the MMU for use. When virtualized however, Xen adds two more layers of frame numbers to the mix.

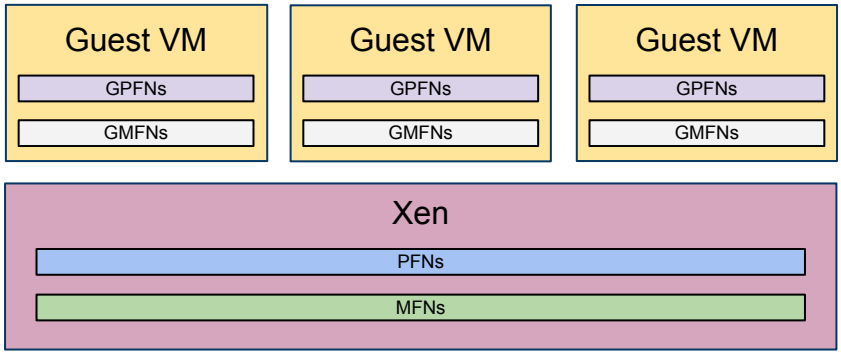

Figure 1.4: Layers of frame numbers with Xen.

From the perspective of a guest VM, PFNs and MFNs map to real memory. In truth, only the hypervisor has access to these items. From the perspective of the hypervisor, the PFNs and MFNs used inside the guest virtualized and are referred to as *Guest Physical Frame Numbers* (*GPFNs*) and *Guest Machine Frame Numbers* (*GMFNs*) respectively [49].

1.5 Malware in the Virtualization Context

The use of virtualization is becoming more prevalent. Virtualized environments now represent a significant portion of systems holding and transmitting valuable data across the globe. Inevitably, malware writers have adapted and turned their attention to virtualization and the cloud.

The advance of virtualization brings about an interesting twist to the story. A hypervisor provides an extra layer of control which can be used in several different ways for running executables in an isolated environment. Until virtualization hit critical mass in the commercial world to facilitate things like cloud computing, VMs were not used necessarily for production systems. Instead, they were often used for testing and, in this context, by malware analysts needing an isolated, controlled environment to run malware. To make their work harder, malware writers would often design their code to avoid being malicious on a virtual machine, so as to hinder reverse engineering and manual analysis. Since virtual machines were not prevalent in production environments, this was a cheap and easy solution to avoid being detected.

These days, due to the popularization of clouds and the production usage of virtual machines, avoiding malicious behavior on a VM is less of a viable strategy. Malware writers risk missing out on a significant portion of potential hosts if they refuse to act maliciously on VMs.

1.5.1 Modern Virtualization Extensions

Hardware virtualization features such as Intel VT [53] and AMD-V [54] introduced by processor manufacturers are intended to reduce the overhead of running a hypervisor in hardware virtualization mode. However, these extensions can be used for other things, such as providing a way to hook into events occurring in a VM and for controlling its internal state. This has become a popular method for introspection and analysis as it provides a hardware-supported method of observing and controlling a VM, while not requiring any special software inside it.

1.5.2 Hypervisor-Oriented Malware

Not only have malware writers started targeting virtual machines, they have begun using the specifics of virtualization to their advantage. Security researchers have shown two interesting developments in this area. These are *hypervisor rootkits* [55]

and *virtual machine-based rootkits* [56]. Each of these complex pieces of malware are quite different while at the same time taking advantage of the existence of a hypervisor.

A hypervisor rootkit is able to detect and compromise a hypervisor like many common pieces of malware might do to a regular OS. From the perspective of a user program in a VM, detection of a hypervisor that is using CPU-provided hardware virtualization extensions is itself a non-trivial task. However, detection of a hypervisor from within userspace has been shown possible by several techniques and must now be assumed possible [57] [58]. Hypervisor rootkits work by compromising an existing hypervisor, thereby gaining access to other VMs and all the power that this entails. A hypervisor rootkit by Wojtczuk et al. [59] is able to infect a pre-existing hypervisor, hijacking it and injecting adjacent VMs easily.

A virtual machine-based rootkit on the other hand, often abbreviated to *VMBR*, is able to act as a hypervisor itself. An example can be seen in a tool such as *Bluepill* [55]. It is able to load a small custom hypervisor into memory space from within an infected non-virtualized native OS. From here it performs a privilege escalation attack, granting privileged execution to the hypervisor. At the same time, it reduces privilege to the OS. As a result, native OS becomes virtualized on-the-fly without being aware of it, as a custom hypervisor is slipped underneath it. Another example of this technique can be found in the *SubVirt* system [56]. Once a machine has been forcefully virtualized, it can be transferred to a remote location, hijacked to perform a man in the middle attack or used for any number of malicious activities.

1.5.3 Hypervisor-Oriented Anti-Malware

The added layer of abstraction which virtualization provides creates new areas for researchers to explore regarding anti-malware and malware detection schemes. There have been many creative approaches to the problem of malware analysis by using a hypervisor environment as a tool.

Patagonix is a system which utilizes the hypervisor to detect binaries attempting to run covertly in a VM [60]. Based on their previous work called *Manitou* [61], Litty et al. achieve relative success in identifying covert binaries such as ones using packing. Patagonix utilizes the hardware virtualization support features which exist in many modern commodity CPUs.

Ether is a project which uses those same virtualization extensions in the CPU [24]. In the form of a Xen hypervisor patch, it monitors, with great detail, processes executing inside a VM. Running Ether severely effects the performance of the VM being monitored, placing it squarely in an auditing space. As a result, the concepts behind its design cannot be applied in their entirety to a running system. However it does give some insight into the tradeoffs between granularity of monitoring and performance penalties.

Sharif et al. argue that dynamic analysis has a better track record than static analysis alone due to techniques such as packing used by malware writers [30]. However if an unpacked binary is presented to a static analysis tool, the effectiveness of such a tool can be fully realized, either working in a stand-alone context or to supplement dynamic analysis. Their system *Eureka* is an implementation of their idea, merging these two families of analysis in a novel way by using a combination of system-call tracing and statistical analysis.

System	DM	HW/SW	VMM Mod	Focus
Patagonix [60]	Dirty page tracking	HW	Yes	Rootkits
Eureka [30]	Syscall tracking Statistical analysis	SW	Yes	Packed/Encrypted malware
Ether [24]	Syscall tracking Stepthrough	HW	Yes	Packed/Encrypted malware

Table 1.2: Virtual Machine-Based System Features – DM: Detection Method, HW/SW: Hardware/Software, VMM Mods: Requires hypervisor modifications

SecVisor is a super-lightweight hypervisor which enforces that only certified code runs in an OS kernel [62]. The most obvious item of interest is that rather than building a tool for a popular hypervisor platform, the authors implemented SubVirt, a custom hypervisor for this specific purpose. The reason this is desirable is it greatly reduces the codebase of the hypervisor implementation making it possible for formal verification [63]. Reduction of hypervisor size for the purpose of formal verification is one domain which has been explored in depth by several groups. Other attempts at this include *Terra* [64], *MAVMM* [65], *Lguest* [66].

1.6 The Trouble with Hypervisor-Based Unpackers

Systems representing the state-of-the-art such as Renovo [43], Ether [24], and Eureka [30] demonstrate how to implement different packed malware detection techniques. Each of these systems is built from the ground up to achieve its task by way of one or more specifically chosen detection techniques. Because of this, each achieves a different success rate depending on the type of malware used to test it. The result of these works is a collection of from-scratch, complex systems, none of which offers a complete solution.

The most recent tools require changes to the hypervisor they run on to enforce the rules they impose and methods they use to detect malware. Tools like Ether [24] and Patagonix [60] are generally proof-of-concept implementations and require a very specific or customized environment to function. Tools such as these do not provide a path for implementation and testing on real-world systems and are also not always available in source form for examination, or expansion by a third party.

The work leading up to now is interesting and often promising, but it would be useful to make it available for use and testing by the larger community. Instead of continuing to build an entirely new system from scratch whenever a new detection technique is to be tested, why not have a simple platform on which many of these techniques can be added? Obviously due to the variations in requirements for different detection techniques, it is hard if not impossible to build a common platform for which every existing technique could work flawlessly. However, the majority of the systems in existence use the same basic principles and functionality and we are confident that a well-designed platform could deal with the majority of cases. An open platform would both help with research and provide a method for industry and the users of such systems to add features they need and assist in improvement and testing. Such a platform would allow incremental addition of detection techniques and features, potentially reducing the overhead in trying new techniques and testing well-known ones in a cloud.

1.7 Desired Features for a Solution

Here we define a set of basic requirements which a potential solution should be able to meet.

- *Expose packed/encrypted executables* - The system should be able to detect and expose programs using packing or encryption.
- *Do not restrict analysis tools* - The system should provide an interface to allow different analysis methods in a simple and language-independent way.
- *Function without major adaptations to the platform* - The system should be installable and usable without needing to setup a custom or modified hypervisor.
- *Be secure* - The system should not introduce new security vulnerabilities.
- *Avoid imposing overwhelming overhead* - The system should not impose an unreasonable overhead to the programs it affects.

1.8 Thesis Statement

We have shown that previous works on detection and removal of packing in a virtualization context have required a modified hypervisor. These changes to the underlying platform often deal with objects at the process level, within guest VMs directly, bypassing the guest OS layer entirely. We have identified some benefits to a paravirtualization-based approach over a hypervisor-based approach which would make it more desirable for usage in a cloud scenario.

The goal of this thesis is to answer the question: **Can a paravirtualization-based approach to packed malware detection achieve the same performance and effectiveness as approaches that involve a modified or custom hypervisor?**

1.9 Thesis Outline

In this chapter, we provided the current state of the malware problem as well as a number of attempted partial solutions to it. We go into depth and take a look at some of the more relevant examples of work on which much of this thesis is based.

Finally Chapter 1 explains the problem this thesis explores and why it is useful in the context of the current state of the art. In Chapter 2 we present an overview of our solution while defining some requirements it should meet in order to be successful. Chapter 3 discusses some technical design choices and then goes into detail regarding its implementation. Chapter 4 outlines how we evaluated our solution, comparing it to the goals set out in Chapter 2 and quantifies the comparison with several experiments. Finally, Chapter 5 discusses some improvements and future work, concluding and determining whether our solution solves the original problem.

Chapter 2

Maitland: Design

In this chapter we present Maitland, a hypervisor-based tool to facilitate the detection and exposure of binaries using encryption or packing. Maitland is a prototype system which uses a known technique to detect unpacking in a VM without modifying the hypervisor it runs on. The design of Maitland is intended to be minimally invasive and easily extended by an arbitrary analysis tool. Maitland's purpose is to be easy to install, deploy, modify, and extend while retaining the power of other research prototypes which require a modified or different hypervisor.

Maitland's initial implementation uses one of many known techniques for unpacking executable and provides a simple method for an arbitrary binary analysis tool to gain access to a memory snapshot of a process taken at a strategic time. The system is minimally invasive, meaning it does not require changes to the hypervisor, and only minimal changes to the guest VM. These changes exist in the form of a loadable kernel module for guest VMs and another for a privileged VM to provide an interface for an analysis tool.

Maitland is designed with the intention of eventually running in a cloud and is easily modified to allow abstraction of certain features and interfaces as remote services. Because the majority of cloud platforms currently run on some form of a paravirtualized Linux kernel [67] [68], our initial implementation focuses on that.

2.1 Architectural Overview

Maitland is a thin layer which resides above a hypervisor, spanning kernel-space between the privileged VM and any instantiated guest VMs. An analysis tool typically

exists in user-space on a privileged VM and interfaces with Maitland through a standard UNIX file-handle and *IOCTL* commands [69]. The simplified architecture is shown in Figure 2.1

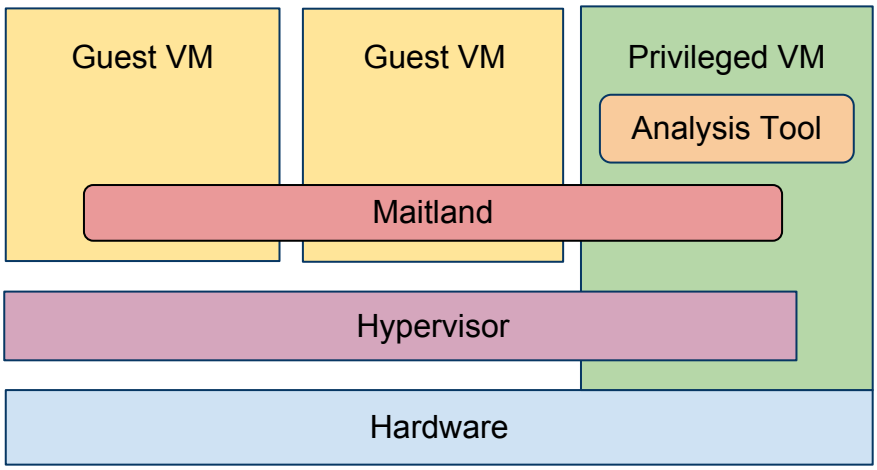

Figure 2.1: Simplified view of Maitland's architecture.

2.2 Unpacking Mechanism

Maitland uses dirty-page tracking as it's primary method of unpacking/decryption detection. Some form of this method is used by tools such as Patagonix [60], Omni-Unpack [42], Renovo [43] and Justin [33].

Maitland uses this technique by handling interesting system events which routinely happen inside the guest VM. A specific intersection of contexts could mean that a potential unpacking operation is happening. In general, a packed/encrypted malicious executable must perform an unpacking/decryption operation and write the resulting instructions to memory. The malware then executes these instructions. By keeping track of the dirty pages, a system can stop the instructions on those pages from being executed.

This can be achieved in a paravirtualized Linux kernel using the *page-dirty flag* and the *no-execute bit* two following tools.

2.2.1 MMU Updates and the Page-Dirty Flag

In order to keep track of which pages have been written to, we use what is called the page-dirty flag. The page-dirty flag is a standard flag used by most modern OSes to know which pages need to be flushed to permanent storage in order to preserve newly-written data. It is also used for copy-on-write mechanisms. The page-dirty flag is typically on the page-table entry corresponding to a given program's access to a page.

When a program allocates memory or modifies a previous allocation, the Memory Management Unit on the CPU detects the operation and notifies the OS via a hardware interrupt. This is one of the mechanisms we use to keep track of dirty pages. This mechanism allows us to monitor new allocations, re-allocations and when a page is being set as writable. In a standard Linux environment, an MMU update happens internally and is generally requested by a CPU and handled by an interrupt handler inside the OS code. In a hardware-virtualized environment, the hypervisor typically intercepts the interrupt instead and handles it internally before passing control transparently to the guest OS. When paravirtualized, the OS is modified to simply do a hypercall to the hypervisor and request an MMU update. We hook this operation to ensure our system is notified when these events occur.

These two mechanisms combined provide a tool for monitoring page writes by specific programs within an OS. With this information the system is aware of which pages are being modified by which programs and in this case, using the mechanisms discussed below, keeps them from being executed.

2.2.2 Page Faults and the NX Bit

As the system can now keep track of dirty pages, it must be able to know when one of them contains data which is being requested as an instruction.

On each page-table entry in a 64-bit Linux system, the 64th bit is known as the NX bit [50, 70]. The NX bit is also known as the Non-Execute bit on Intel x86_64 architecture and XD or Execute Disable bit on AMD64. This bit provides a hardware-enforced method of preventing the execution of specific memory pages in the case of x86_64 or AMD64. This mechanism is also available in a software-emulated form on some 32-bit OSes.

When a page whose page-table entry is marked not-executable contains data being accessed as instructions , the CPU causes a page fault. As there are many types of

page faults caused by a variety of different things, Maitland focuses on those caused specifically by an instruction fetch operation on a page marked not-executable. By inserting a callback in the paravirtualization code which interfaces with the hypervisor, Maitland can filter page faults and respond only to those of interest.

Thus, it checks for the type of page fault, seeing if it was caused by an NX bit violation. If so, it checks to see if the CR2 register contains a stack pointer which exists in the mapped memory of a process of interest at the instant of the page fault. If the fault makes it through these filters, it is a potential unpacking/decryption operation. What Maitland does in reaction to this is described in the next section.

2.3 Accessing a Snapshot

Once Maitland has detected a page fault it is interested in, it needs to determine whether or not the memory snapshot is of any value. It does this by asking an external analysis tool. First Maitland takes the process that caused the page fault off the scheduler, effectively pausing it. Once the process has been stopped, Maitland traverses its page-table making a list of it's physical pages. The system will find every page which is mapped into the program's virtual memory space and make them available to the analysis tool via a standard UNIX file handle in another VM. Maitland will block further reports of that process until it receives a command specifying a response from the analysis tool.

2.4 Responding to a Threat

The design of Maitland does not limit what can be done in response to report. There are different approaches which can be considered, depending on the circumstances, two of which are discussed here.

2.4.1 Guest-Internal Responses

The code in the guest OS allows Maitland to affect the internal environment. One option for an in-guest response might be to simply terminate or resume the process based on whether it is deemed malicious or not. It is also possible to dump process snapshots at each occurrence if malicious intent is suspected but not yet discovered.

This might be useful to help reversea engineer a packer, for auditing, or just to reduce the penalty of a false-positive.

2.4.2 Guest-External Responses

In a cloud or large cluster infection of adjacent nodes is unacceptable. If Maitland were deployed in a cloud, it might be desirable to automatically terminate, pause or take a snapshot of the entire VM containing an offending program.

One interesting approach might be to sandbox the VM. By sandboxing the VM contents, any network or disk commits made by the OS are temporarily queued up instead of actually completing. This capability could be enabled once we suspect malicious activity, or simply while a snapshot is being analyzed and we cannot afford to halt the process we are analyzing. It could also be useful for reverse engineering purposes. Another approach might be to trigger intensified analysis of the guest VM's network traffic.

2.5 Maitland as a Split Driver

Maitland consists of a frontend driver to be inserted into kernel space in an unprivileged VM and a backend driver similarly for a a privileged VM. Maitland handles events in side a guest OS and can provide information to the privileged VM using pre-existing tools.

Our initial implementation of Maitland supports Linux guest VMs only. Linux is well supported by Xen. A Windows version is certainly desirable because the majority of Malware targets that platform. Unfortunately building a Windows-based custom paravirtualized kernel for Xen would be much more work.

Several properties exist as a result of this approach. We list these properties below.

Work on a Common Platform

There are several existing systems built to perform a similar task to Maitland. The difference is that those systems are generally designed to work in a lab setting on a specific version of a hypervisor using specific hardware features. Maitland was designed to work on as wide a variety of virtualization setups as possible. By choosing the most common environment, Maitland's task of being compatible is already partially done. The most popular platform at the

beginning of Maitland's conception for large cloud providers (such as Amazon EC2) was the Xen hypervisor using paravirtualized guest kernels [71]. Not only is it the most common, but involving a paravirtualized guest OS gives much more flexibility regarding compatibility with various versions when compared to the alternatives. There are other inherent benefits of a paravirtualized guest OS which make it the most popular among cloud providers. One of these benefits is performance when compared to hardware-virtualized equivalents. [49, 50, 68] although hardware-virtualized performance is closing the gap [72] with the help of CPU manufacturers.

Do not Require Hypervisor Modifications

The system is implemented as a pair of small kernel modules and a user-space tool, each of which is described in detail in Chapter 3. The most successful previous work on this subject require either changes to the hypervisor or a completely new one. A third-party patched hypervisor or a custom-built one is less likely to be adopted by a cloud provider with thousands of servers. Because Maitland does not touch the Xen hypervisor itself, it is relatively simple to deploy in an existing system. All one must do is load a guest module in each guest VM, and a privileged module in a privileged VM. Depending on the setup, an administrator might not even have to reload the deployed VMs.

Monitoring Guest Internal State

In order for Maitland to operate, it must monitor states, which it gets from internal data-structures within a guest VM's OS. This can be done using *Binary Offsets* or by embedding code in the OS.

Binary Offsets

This method for accessing OS components in a VM from outside involves using memory offsets into a binary image. What this means is that the host or outside system must know in advance the byte offsets into a VM image of a given OS component or data-structure. The system can basically point to a specific offset, and treat that memory segment as the data-structure it expects to be there. This violates our desired property of not modifying the hypervisor. In addition, the required knowledge of image offsets is tedious to compute as they vary between most images and OSes. Some OSes have a feature called *address space layout randomization* [73],

which randomizes the location of important data-structures as a security measure against code injection, further complicating things. The strategy of using binary offsets is used by tools like Ether [24]. A benefit of this strategy is the transparency it grants, allowing the monitoring system to remain covert. Although current research suggests that covertness is not necessarily possible to maintain in reality [57, 58].

Embedded Code

The method we chose for Maitland, is to implement a kernel module (or driver) to provide an interface to those important internal data-structures. The benefit here is that these data structures can simply be referenced from within the OS. In addition, it is easier to affect the state of the system. By this we mean it is easier to respond to a threat and enforce a decision within the VM. The major downsides with this method are that Maitland loses full transparency (although we might not have been able to guarantee it [57, 58]) and introduces additional code into a guest kernel.

Small Code Size

A key goal of the implementation of Maitland was to keep as much code as possible outside of guest VMs. Even though it cannot be removed entirely when implemented as a split driver, it is better to reduce the code base in the guest as much as possible. Having a small code base is a good practice in order to reduce the possibility of introducing a vulnerability and limiting the impact on the system. Additionally, a reduction in code size allows for the possibility of formal verification. SubVirt also uses a small code base for the purpose of formal code verification [62].

Covertness

Many prior works attempt to keep programs in a guest VM unaware that they are running in a virtual machine or are being monitored. We would like to argue two points against this idea. The first point is that in the past, malware has predominantly targeted OSes running on native hardware. Malware writers might have attempted to detect the presence of a hypervisor and write their software to avoid malicious behavior. The emergence of cloud computing is shifting the paradigm toward a virtualized environment as the predominant one. Thus, any malware not willing to be malicious within a VM will quickly become irrelevant, as discussed in Section 1.5. In addition, there have been

several publications of techniques with which malware can detect the presence of a hypervisor [55, 57, 58, 74]. Because of this, it is not currently possible to guarantee that applications within a VM are ignorant that they are not running on native hardware. For these reasons, we believe achieving absolute transparency is both largely infeasible and unnecessary.

2.6 Summary

In this chapter, we presented Maitland our virtualization-based tool for detecting encrypted and packed binaries. Table 2.6 compares Maitland's basic features to those of other work.

System	DM	HW/SW	VMM Mod	Focus
Patagonix [60]	Dirty page tracking	HW	Yes	Rootkits
Eureka [30]	Syscall tracking Statistical analysis	SW	Yes	Packed/Encrypted malware
Ether [24]	Syscall tracking Stepthrough	HW	Yes	Packed/Encrypted malware
Maitland	Dirty page tracking	SW	No	Packed/Encrypted malware

Table 2.1: Maitland Compared To Other Approaches – DM: Detection Method, HW/SW: Hardware/Software, VMM Mod: Requires hypervisor modifications

We described some of the choices which were made regarding the design of Maitland and how it differs from other approaches. The next chapter details Maitland's implementation.

Chapter 3

Maitland: Implementation

In this chapter we examine Maitland in greater detail. We describe each of Maitland's components individually as well as their interaction with each other. We outline expected behavior of the system as well as details regarding functionality of Xen and the 64-bit paravirtualized Linux kernel that Maitland uses.

3.1 Detailed Architecture

The implementation of Maitland is built upon the Xen hypervisor. Xen is a popular product and, at the time of Maitland's conception, was more mature than competing open source hypervisors like KVM [75]. Because most hypervisors possess similar functionality, it would be possible to port Maitland to KVM or some other hypervisor.

3.1.1 Components

Maitland has three main components: a) a privileged VM kernel module, b) a guest VM kernel module and, c) a sample analysis tool. Each is located in a different memory space, guest VM kernel-space, privileged VM kernel-space and privileged VM user-space respectively. The guest and privileged kernel modules are loadable Linux kernel modules. The guest module is intended to be loaded onto any guest VM to be monitored. The privileged module must be loaded into a single privileged VM, usually dedicated for this purpose. Its job is to coordinate between the analysis tool and the guest module(s). The analysis tool resides in user-space within the privileged VM and interfaces directly with a privileged VM module using IOCTL and Fops

commands. In addition, a sub-component, a Python helper daemon to assist the privileged VM kernel module resides in the privileged VM user space.

The sample analysis tool provided with our implementation demonstrates how Maitland can be used to detect simple forms of packed malware but can be replaced or extended to deal with more complex variants.

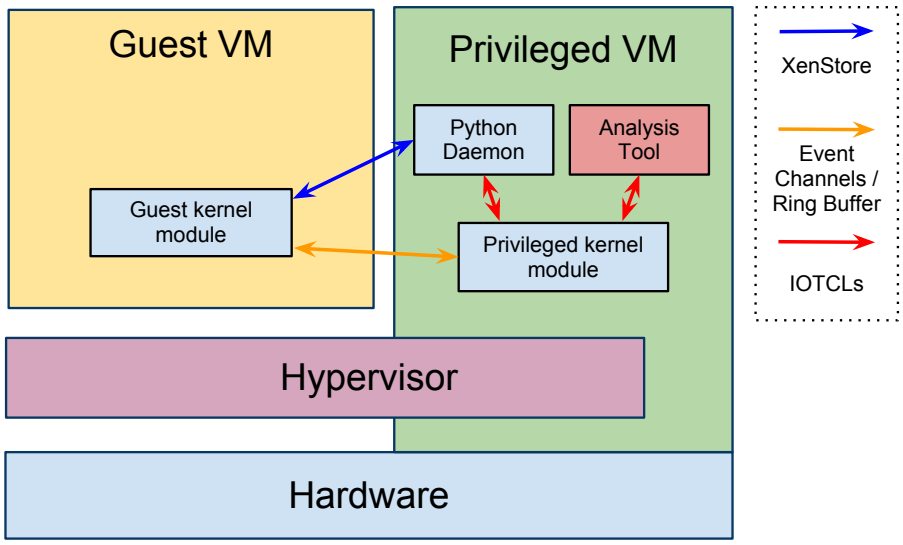

Figure 3.1: Maitland's detailed architecture.

3.1.1.1 Privileged VM Loadable Module

This component contains the main code-base of Maitland. It exists as a loadable kernel module inside a privileged VM. Maitland uses a combination of XenStore and a shared ring buffer for inter-domain communication between kernel modules. This is because XenStore, although simple and useful in many cases is not well suited to any data which cannot easily be mapped to a key/value pair of strings. A ring buffer supported by event channels is better suited for quick, well-defined passing of raw data back and forth. This component's purpose is to:

- set up an initial environment necessary to interact with other VMs
- set up communication channels to those VMs
- respond to commands it receives from an analysis tool
- filter events incoming from monitored VMs

- notify an analysis tool of suspicious events
- facilitate a response to suspicious events

This component communicates directly with an adjacent guest VM module as well as an analysis tool. It also receives some assistance from a helper Python daemon which also resides on the privileged VM.

3.1.1.2 Guest VM Loadable Module

This component contains the basic logic, hooks and handlers for events generated both internally and externally to a guest VM. It exists as a loadable kernel module within a guest VM. This component's purpose is to:

- register with a privileged VM and set up a communication channel with it
- intercept and filter important events within a guest VM, including page faults and MMU updates
- notify a privileged VM of important events happening inside the guest VM kernel
- export process-specific information, including memory snapshots to a privileged VM
- enforce decisions sent by a privileged VM

This component represents the in-guest code, which our design holds here rather than in the hypervisor code-base. Maitland uses this component to watch guest VMs and programs inside them.

3.1.1.3 Analysis Tool

This component is not a part of Maitland itself, but still plays an important role in Maitland's operation. This tool is intended to be a user-space application which lives in the privileged VM's user space. The job of the analysis tool is to interface with Maitland, setting watches on programs, and analyzing process reports provided by Maitland. This component's purpose is to:

- specify which programs in a guest VM are to be monitored

- analyze memory space of suspicious monitored programs
- specify a response to observed suspicious activities

3.1.1.4 Privileged VM Python Daemon

Interaction between kernel space in a privileged VM and XenStore/XenBus is not well supported by Xen. The Python user-space interface for XenStore on the other hand is simple, effective and relatively well supported. Thus, it greatly simplifies our implementation to use a Python tool to perform this interaction on behalf of the privileged VM kernel module. This component's purpose is to:

- detect newly created guest VMs through XenStore
- facilitate the passing of information required to set up a proper communication channel between a guest VM module and a privileged VM module
- invoke an analysis tool on behalf of the privileged module

3.2 Terminology

These are a few terms which are used to refer to certain operations within Maitland.

3.2.1 Registration

Registration refers to the action taken between a guest VM's kernel module and the privileged VM's kernel module. This action occurs when a new guest VM boots up and makes the privileged VM kernel module aware of it. After registration, the privileged VM will monitor events from the registered guest VM. This process is detailed in Section 3.3.1.

3.2.2 Watch

A *watch* is a notification that a specific process within a guest VM is to be monitored. A watch is usually set by the privileged VM, a procedure which can be automatic or manual. When a watch is set on a process, filters within the privileged VM that sort through guest VM events are modified to include those pertaining to that process. Maitland is explicit about programs it monitors, meaning a watch must be set on a

program if it is to attempt to detect suspicious activity. As previously stated, this can be automated to include all of a user's programs if desired. This process is described in detail in Section 3.3.2.

3.2.3 Report

A *report* is a contextual snapshot of a watched process within a registered guest VM. A report is requested by a privileged VM after it has noticed events of interest on a watched process. A report is triggered only on watched programs and is received directly by the privileged VM module and forwarded to the analysis tool. A process report will be presented to an analysis tool as a binary file. This file contains the content of that process' memory space and can also contain additional metadata. This process is described in detail in Section 3.3.4.

3.3 Operational Sequence

Here we illustrate Maitland's operation in a typical situation, Figure 3.3 outlines this sequence.

3.3.1 Initialization and Registration

Maitland is initialized by first inserting the privileged VM loadable module into the kernel of the chosen privileged VM. The module will prepare certain internal data-structures it will use to track process memory space and register a virtual device in the privileged VM at */dev/monitor*. Through this device, other tools will pass commands via IOCTL calls.

At this point, the Python helper daemon will register a XenStore watch (different from a watch on a process discussed in Section 3.2.2) on paths which will be created when a new guest VM is started. This is a callback that is triggered when a path has been accessed in any way. It can be placed on a non-existent path and is registered via the Python API call *xswatch()* (illustrated in Figure 3.3).

The system now waits for a guest VM to start. When a guest VM is started and begins to boot, the python daemon will detect the new VM. It will create a new registration path accessible only to that VM on which it will place a XenStore watch. Even if a guest has been detected it must register by writing into the path made

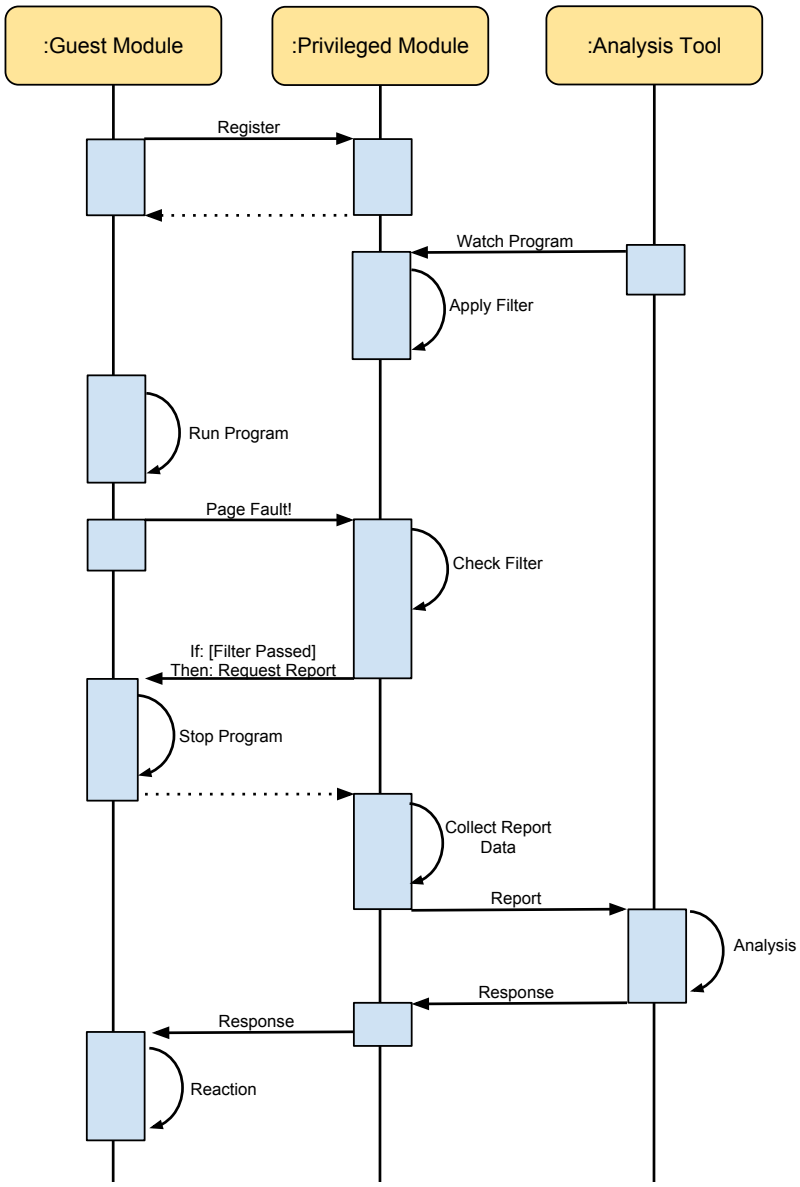

Figure 3.2: Sequence diagram of Maitland's component interaction

specifically for it inside XenStore. Thus, if a VM does not register, processes running in it cannot be monitored by Maitland.

Once booted, the VM will then initialize the guest VM loadable module. For convenience, we include a script to automatically set watches on new programs started by a user. For this reason, the module also registers a device called */dev/malpage* in order to receive these watches directly via IOCTL commands. The module then loads a dedicated report thread, which is responsible for independently processing

```
1  from xen.xend.xenstore.xsutil import *
2  from xen.xend.xenstore.xswatch import *
3
4  #Begin a XenStore Transaction
5  th = xs.transaction_start()
6
7  #Creating a path
8  xs.mkdir(th,watch_path)
9
10 #Registering a watch
11 xswatch(watch_path, callback_function, xs)
12
13 #Ending a XenStore transaction
14 xs.transaction_end(th)
```

Figure 3.3: Interaction with XenStore via the Python API

report requests in the background while keeping them in serial. The kernel functions *do_page_fault*, *mmu_update* and *multi_mmu_update* are then injected with code which will call custom handling procedures within the module's codebase. These handlers are as streamlined as possible and include the filters which eventually determine whether faults or MMU updates are suspicious.

A communication channel to the privileged VM module is then prepared by allocating a memory page for a ring buffer and initializing it. Initialization of a shared ring buffer is done using the *SHARED_RING_INIT()* call and then set as the frontend of a split driver using the *FRONT_RING_INIT()* call. The privileged VM is then granted access to the page using the *gnttab_grant_foreign_access()*. The guest VM module allocates an event channel with the *HYPERVISOR_event_channel_op* hypercall and then binds it to a local handler with *bind_evtchn_to_irqhandler()* (Figure 3.4). Once this has been done, the communication channel is ready for use and the guest VM module finally writes a notification into its XenStore registration directory.

The change in the guest VM's registration directory in XenStore triggers the Python helper daemon to wake up. The daemon notifies the privileged VM loadable module via an IOCTL, passing it information regarding the new guest VM. It creates a path in XenStore to contain report and watch metadata, then registers a watch on it.

The privileged VM loadable module takes the information given to it by the python daemon and begins by mapping the remote shard page into its own memory space. It continues by completing the grant using *HYPERVISOR_grant_table_op()* and then

```
1  struct as_sring *sring;
2  struct as_front_ring fring;
3  int err;
4  struct evtchn_alloc_unbound unbound;
5  domid_t domid; //The domain id of the VM to share with
6  unsigned int evtchn;
7  char* chan_name = "mychannel"
8  struct shared_structure_t* shared;
9
10 //Allocate individual page
11 sring = (struct as_sring*)__get_free_page(GFP_NOIO | __GFP_HIGH);
12
13 //Put a shared ring structure on this page
14 SHARED_RING_INIT(sring);
15 FRONT_RING_INIT(&fring,sring,PAGE_SIZE);
16
17 //Grant the page
18 err = gnttab_grant_foreign_access(DOM0_ID,virt_to_mfn(fring.sring),0);
19 //On success, err contains the grant reference
20
21 //Allocate event channel
22 unbound.dom = DOMID_SELF;
23 unbound.remote_dom = domid;
24 err = HYPERVISOR_event_channel_op(EVTCHNOP_alloc_unbound,&unbound);
25 evtchn = unbound.port;
26
27 //Bind event channel to event handler
28 err = bind_evtchn_to_irqhandler(evtchn,irq_handler,0,chan_name,shared);
```

Figure 3.4: Guest VM set up of a shared ring buffer in C

initializing the ring buffer as its backend with *BACK_RING_INIT()*. It attaches an event channel and then creates a new entry in the data structure it uses to track monitored processes (Figure 3.5).

Maitland is now initialized and exists in a state where the guest VM and privileged VM possess a configured and shared memory ring buffer.

3.3.2 Starting and Watching a Process

One method of setting a watch is to have a pre-defined list of process/program names which the system notices and automatically sets a watch on. Our implementation works in a simpler way by running monitored processes with a simple wrapper application. The wrapper executes a process and then automatically reports its process ID to Maitland. The process ID is sent to the guest VM loadable module via an IOCTL.

```
1  int err;
2  struct vm_struct *v_start;
3  struct gnttab_map_grant_ref ops;
4  struct gnttab_unmap_grant_ref unmap_ops;
5  unsigned int gref; //Contains the pre-made grant reference
6  domid_t domid; //The id of the adjacent VM
7  struct as_back_ring bring;
8  unsigned int evtchn; //Contains the existing event channel
9  char* chan_name = "mychannel"
10 struct shared_structure_t* shared;
11
12 //The following function reserves a range of kernel address space and
13 //allocates pagetables to map that range. No actual mappings are created
14 v_start = alloc_vm_area(PAGE_SIZE);
15
16 //Map in the remote page
17 gnttab_set_map_op(&ops, (unsigned long)v_start->addr,GNTMAP_host_map,
       gref,domid);
18
19 //Accept the grant
20 if (HYPERVISOR_grant_table_op(GNTTABOP_map_grant_ref,&ops,1)) {
21         return -EFAULT;
22 }
23
24 //Get a handle on the ring sitting in the page
25 unmap_ops.host_addr = (unsigned long)(v_start->addr);
26 unmap_ops.handle = ops.handle;
27 sring = (struct as_sring*)v_start->addr;
28
29 //Initialize as backend
30 BACK_RING_INIT(&bring,sring,PAGE_SIZE);
31
32 //Set up an event channel to the frontend
33 err = bind_interdomain_evtchn_to_irqhandler(domid,evtchn,irq_handler,0,
       channel_name,shared);
34 //err now contains the event channel for this end
```

Figure 3.5: Privileged VM connection to a shared ring buffer in C

The guest VM loadable module then makes a summary and places it in a dedicated XenStore directory for the privileged VM.

The helper Python daemon intercepts this information and gives it to the privileged VM loadable module which updates its internal data structures, taking note and preparing itself for the reception of process reports. This information is used later to decide whether the page faults emitted from inside the guest VM are actionable.

3.3.3 Intercepting Dirty Page Executions

Once a guest VM has registered and at least one watch has been set on a process, the guest VM loadable module processes page faults and MMU updates occurring within its VM.

When an MMU update is intercepted, if the cause was a process attempting to allocate memory, marking a page as writable, or marking a page as dirty, Maitland uses the ring buffer to forward the event to the privileged VM module. The privileged module then looks to see if it is a watched process performing the operation. If the operation was performed by a watched process, the privileged VM loadable module requests the page(s) involved in the operation be set as non-executable. The guest VM loadable module accesses the page-table entry for the involved page(s) and sets the *_PAGE_NX* flag.

If a page fault was intercepted by the guest VM loadable module rather than an MMU update, the guest module checks if the fault was caused by an attempt to execute a page marked with the *_PAGE_NX* flag. If so, the fault is summarized and sent to the privileged VM and the intercepted fault is immediately resumed to avoid excessive overhead. The privileged VM loadable module then checks to see if the fault was caused by a watched process by checking its hash table. If a watched process attempted to perform an instruction-fetch operation (execute an instruction) on data contained on the involved page, the privileged VM immediately responds to the guest VM module requesting a report of the process.

Communication between these two modules is asynchronous to avoid unnecessary overhead caused by blocking.

3.3.4 Generating a Process Report

When a report is requested by the privileged VM, the dedicated report thread running in the guest VM module is woken up via a *semaphore*. This thread takes the process off the scheduler, effectively pausing it. The thread then locks the kernel data-structures related to the process' page table and begin page discovery.

There are two ways this is able to happen. The first is to recursively traverse the page table pointed to by the *pgd_t* pointer inside the process' *task_struct* structure. This works but can result in some redundancy as the same page can be mapped into a page table multiple times. The second method is to iterate through the contiguous memory segments pointed to by the process' *vm_area* structs in *PAGE_SIZE* sized

chunks. As the iteration continues, each address is resolved to a *pte_t* and then to a *PFN*. We found less redundancy, slightly faster speed, and simpler code in this approach, thus Maitland uses it by default.

Once a list of the process' *PFNs* is built, the privileged VM is granted access to the pages they represent using Xen's grant tables (again using the *gnttab_grant_foreign_access()* call). The result is a list of grant references. This list of key/value pairs consisting of PFNs and their corresponding grant references is then placed into XenStore by the guest VM loadable module.

At this point, the suspicious process is halted, its memory waiting to be accessed by the privileged VM and all the information required to access it is written into XenStore.

3.3.5 Analyzing a Process Report

The addition of the PFNs and grant references into XenStore triggers a handler in the Python helper daemon. This handler extracts the information and passes it to the privileged VM loadable module. Each grant is completed and the remote PFNs are then mapped into a local virtual memory area by the loadable module. When the privileged VM loadable module returns, signifying that the memory snapshot is ready, the Python helper daemon invokes an analysis tool.

The analysis tool then opens the */dev/malpage* file for reading. The privileged VM loadable module has an overridden implementation of the kernel *read()* function. This function implements the actions required to access the memory snapshot as it it were a file. The analysis tool opens the file and reads it in the same way it would a binary file requesting additional metadata via IOCTL if required.

In our implementation, we used a simple Python script as a wrapper for the grep utility. The grep utility is able to search binary files for a binary string.

3.3.6 Responding to Suspicious Activity

Upon completion of its analysis, an analysis tool must respond to Maitland. The analysis tool informs Maitland how to respond to a report, deciding what to do in response to the report. The majority of the time, this will be to simply resume the process. If an analysis tool detects malicious activity based on the report, it will request the process be terminated. This is done via an IOCTL command. More

options for responding to a threat such as stopping the guest VM or placing it in a sandbox were discussed in Section 2.4.

After receiving the correct IOCTL command, the privileged VM loadable module notifies the correct guest VM loadable module through the ring buffer and triggers the software interrupt corresponding to that VM. The guest VM then resumes or halts the process in question and dumps all information and state associated with the report. Further report requests will now be allowed and the reporter thread resumes its wait state in the context of that watched process.

3.4 Summary

In this chapter, the details of Maitland's implementation were discussed. We examined how Maitland's internal components interact and how other software interfaces with Maitland. We also discussed how Maitland handles events generated within the guest OS, something which is crucial in understanding both how well Maitland's technique of detection works and how its performance can be improved. In the next chapter, we will examine Maitland through a few experiments and compare it to other solutions.

Chapter 4

Evaluation and Analysis

This chapter describes our evaluation of Maitland. We look at whether or not Maitland is able to achieve feature parity with other solutions, without requiring a modified hypervisor. We also measure the performance penalty of running Maitland. We start by discussing how our experiments will be measured and how we will measure them in Section 4.1. In Section 4.2 we present the results of those experiments. Finally we analyze the results and compare Maitland's feature set with that of the original goal in Section 4.3 and also talk about a few extra interesting properties regarding Maitland in Section 4.4.

4.1 Evaluation Parameters

In this section we provide technical information regarding the setup and execution of our experiments.

4.1.1 Hardware and Software Environment

Our test system had an Intel Xeon dual-core 3.4 GHz and 2.5 GB of memory. The virtual machine environment used was Xen 4.0.1 the 2.6.32.27 64-bit kernel on Debian. Each test VM used the same kernel but with paravirtualized guest features enabled, 1 virtual CPU and 128 MB of ram. Details can be found in Appendix A.

4.1.2 Samples

Due to the fact that our implementation of Maitland is on a Linux paravirtualized kernel, our selection of compatible packers for testing was limited. However, there are still a small number of compatible packers available for Linux. For our tests we chose two packers. The first was UPX [36], one of the most common tools used for malware packing. The second was *gzexe* which comes pre-installed on some Linux distributions. Other packers compatible with Linux include the *HASP Envelope* [76], which we were unable to acquire, and *exepak* [77], which we were unable to successfully compile.

4.1.3 Procedures

We need to determine just how effective Maitland is at unpacking an executable packed with our sample packers. We also need to determine the overhead Maitland introduces to a monitored process if it does not contain malicious code, i.e. false positives. We have constructed some simple experiments to help determine these items. In order to provide a baseline for worst-case performance, all experiments are done with Maitland in synchronous mode, meaning that target processes are not permitted to execute while a report is in progress.

4.1.3.1 Test Programs

For sample programs, we chose two CPU-intensive applications and one memory-intensive one. These are as follows:

- Calculating π. We did this using the pi_css5 [78] tool to calculate π to an arbitrary number of decimal points. In this case we calculated π to 2^{17} (131072) decimals.
- Compressing a randomly generated binary file. We did this using the *gzip* [79] tool to compress a randomly generated 10 MB binary file.
- Allocating a large memory segment. We did this using a custom program which allocates a variable amount of memory, and writes data into each allocated page.

4.1.3.2 Analysis Tool

As Maitland itself does not include an analysis component, we built an arbitrary one to use for our experiments. This sample analysis tool was extremely simplistic. In advance, we gave the analysis tool a binary string which we extracted from the pi_css5 and gzip executables. The analysis stage was composed of simply using the UNIX grep tool to search the binary snapshot for the given binary string whenever a report was triggered. It turned out for our purposes this was adequate. The default policy was to kill the reported process when a known string was found, otherwise resume it.

4.2 Results

4.2.1 Experiment 1: Unpacking

The goal of this experiment was to determine how successful Maitland could be at unpacking executables packed with our sample packers. We took gzip and pi_css5, packed them with gzexe and UPX, resulting in 6 binary files: gzip, gzip_upx, gzip_gzexe, pi_css5, pi_css5_upx , pi_css5_gzexe. We extracted binary strings from both gzip and pi_css5. For confirmation, we used grep to search the gzip binary for the string with success, while failing to detect them in the two packed versions of that binary. The same was true for the pi_css5 executable and its packed copies.

This procedure simulates that gzip and pi_css are malware and that our analysis tool possesses a valid signature for each. The packed versions of those binaries hide these signatures.

We ran all six binaries while being monitored by Maitland and with our analysis tool searching snapshots for our preset strings. Table 4.1 contains our results, showing that Maitland saw through the packed layer and exposed the internal code to grep.

Executable Name	Grep	Maitland+Grep
gzip	detected	detected
gzip_upx	not detected	detected
gzip_gzexe	not detected	detected
pi_css5	detected	detected
pi_css5_upx	not detected	detected
pi_css5_gzexe	not detected	detected

Table 4.1: Detection Results

Figure 4.1 shows the time required by each executable to calculate 2^{17} decimals of π on a test VM compared with the amount of time Maitland took to detect a signature within the pi_css5 executable.

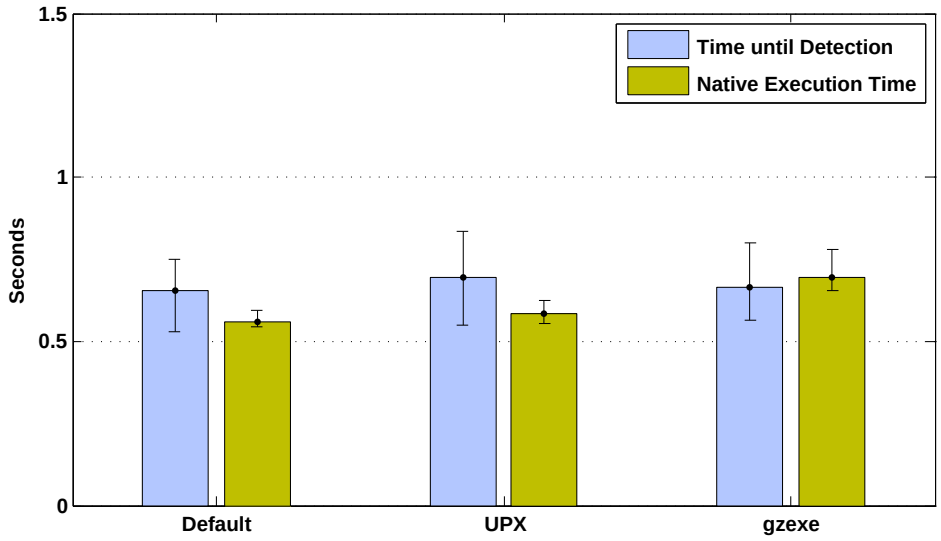

Figure 4.1: Time taken for Maitland to detect a signature while calculating 2^{17} decimals of π.

Figure 4.2 shows the time required by each executable to compress a 10 MB randomly generated file on a test VM compared with the amount of time Maitland took to detect a signature within the gzip executable.

4.2.2 Experiment 2: Overheads

In this experiment, the goal was to determine the overhead induced by Maitland when it monitored a program which was not malicious. We ran each executable without Maitland to get a benchmark for standard execution times. We then ran each again with Maitland monitoring in the background, without providing the analysis tool with a known binary string. We measured the time of execution for each case. Because Maitland utilizes software interrupts heavily as a result of using XenBus and event channels, we also measured the number of software interrupts used by the system during each test. Preliminary tests indicated that, as expected, simply having

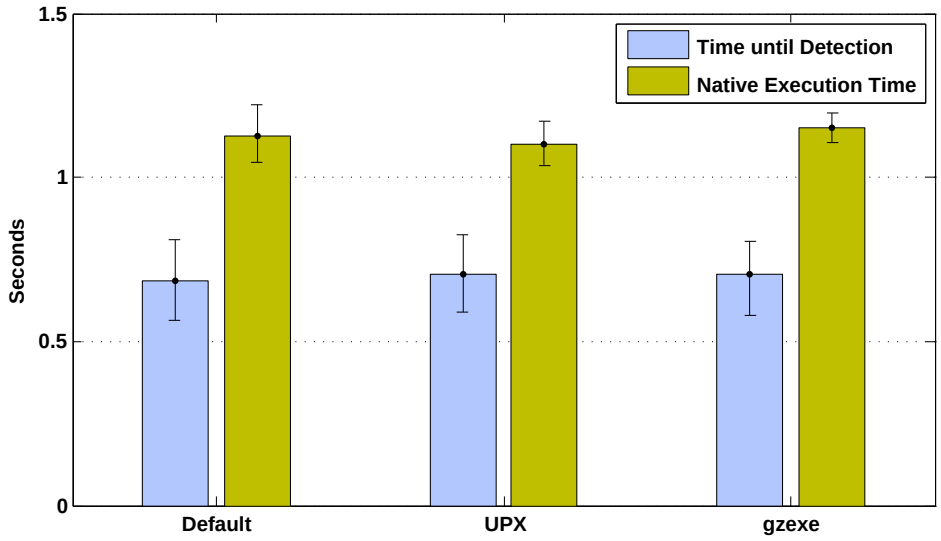

Figure 4.2: Time taken for Maitland to detect a signature while gzipping a 10 MB file.

Maitland in place but not monitoring induces no direct overhead on a program when compared to having a machine completely free of Maitland. These results are omitted.

Figure 4.3 shows the time required by each executable to compress a 10 MB randomly generated file on a test VM both with and without Maitland monitoring.

Figure 4.4 shows the time required by each executable to calculate 2^{17} decimals of π on a test VM both with and without Maitland monitoring it.

Figure 4.5 shows the number of software IRQs used by the system while compressing a 10 MB randomly generated file on a test VM both with and without Maitland monitoring it.

Figure 4.6 shows the number of software IRQs by the system while calculating 2^{17} decimals of π on a test VM both with and without Maitland monitoring it.

As we can see, Maitland induces a significant increase in system utilization. We attribute this to having too liberal of a filter policy, meaning Maitland requests a report too often, resulting in too many false-positives as far as page faults are concerned. We believe a significant portion of this overhead can be mitigated by tightening the filter and checking for more flags relating to the page faults occurring. The current set of filters results in a worst-case performance penalty because Maitland processes

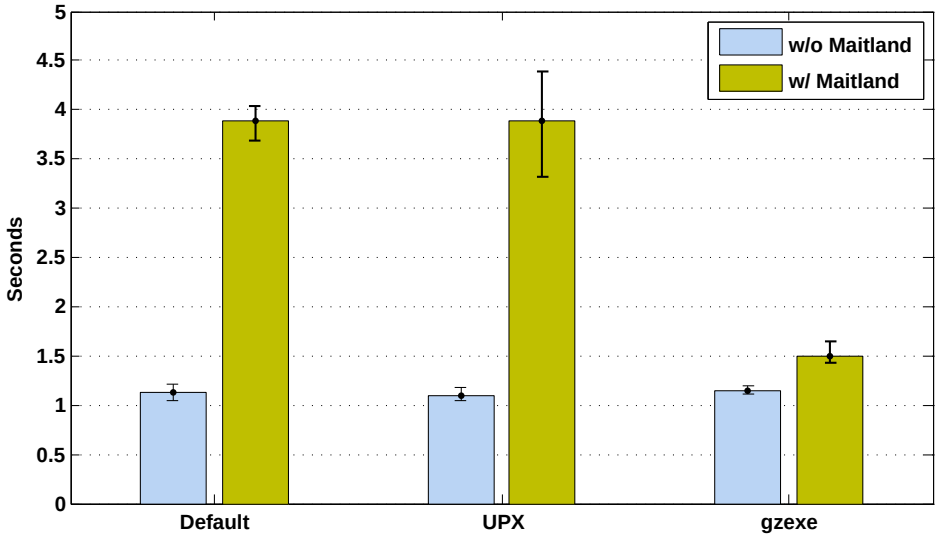

Figure 4.3: Time to gzip a 10 MB file.

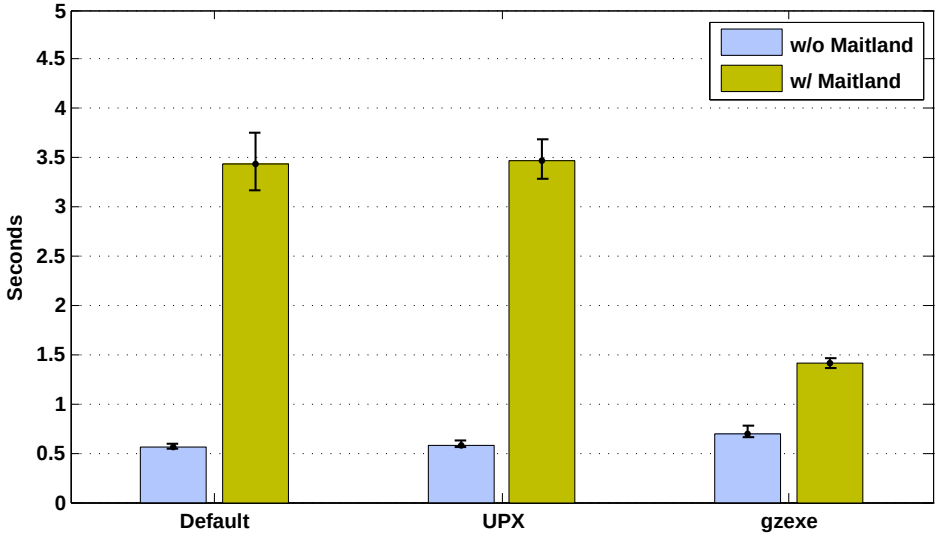

Figure 4.4: Time to calculate 2^{17} decimals of π.

all events. Keeping track of faults and their contexts would help Maitland more accurately choose which faults to act on and which to ignore.

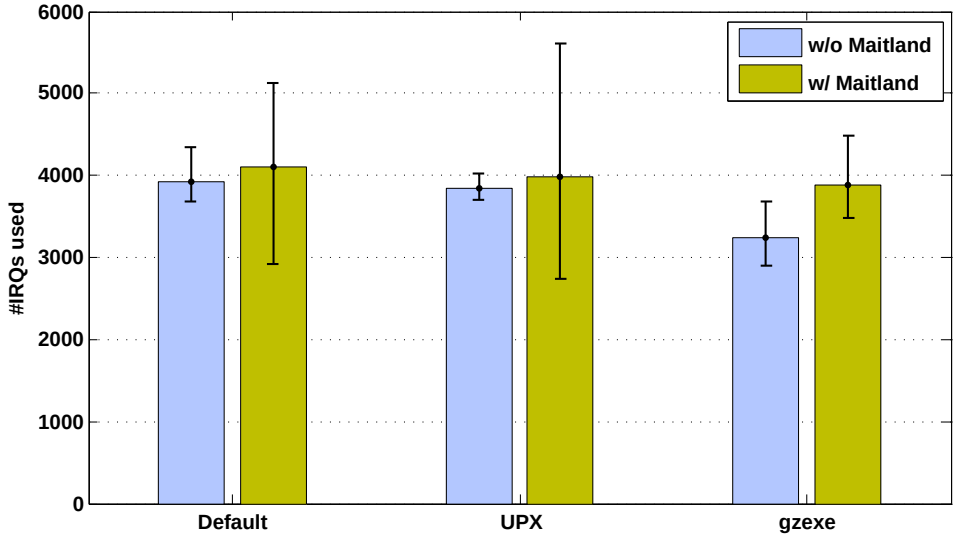

Figure 4.5: IRQs used while gzipping a 10 MB file.

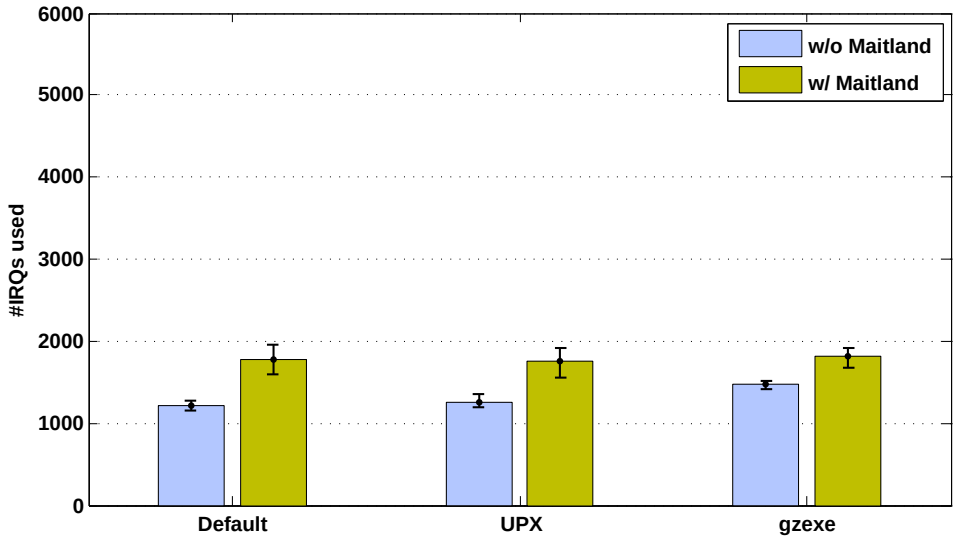

Figure 4.6: IRQs used while calculating 2^{17} decimals of π.

Here we combine the information from Sections 4.2.1 and 4.2.2 for a more condensed view. Figure 4.7 compares the time required by each executable to calculate

2^{17} decimals of π, the amount of time Maitland took to detect a signature within the pi_css5 executable, and the final execution time of the executable if Maitland recognized no signatures.

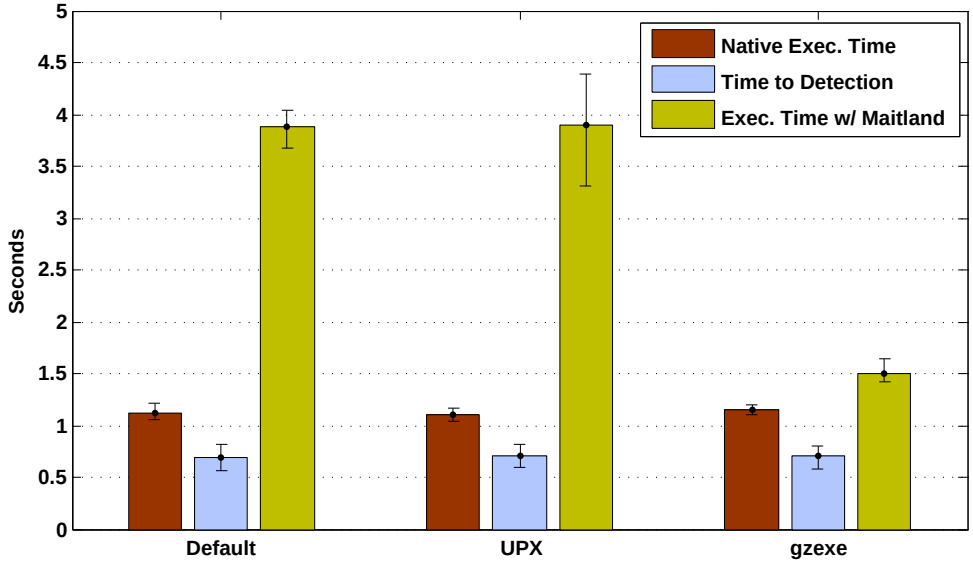

Figure 4.7: Summary of Maitland's effect on the pi_css5 executable.

Figure 4.8 compares the time required by each executable to compress a 10 MB randomly generated file, the amount of time Maitland took to detect a signature within the gzip executable, and the final execution time of the executable if Maitland recognized no signatures.

4.2.3 Experiment 3: Process Memory Size

The goal of this experiment was to determine just how the allocated memory size of a watch program affects Maitland and the overhead caused. Because Maitland must traverse process page trees to collect a memory snapshot, understanding how the process' page tree size affects Maitland is useful. It is logical that as the number of pages increase, so does the time Maitland takes to do its work. We completed this experiment by running our sample program a number of times with exponentially increasing memory size requests and with Maitland monitoring in the background.

Figure 4.9 shows the performance hit induced on a process by Maitland as memory size increases. At around 160 pages of requested memory, we see a dip in the

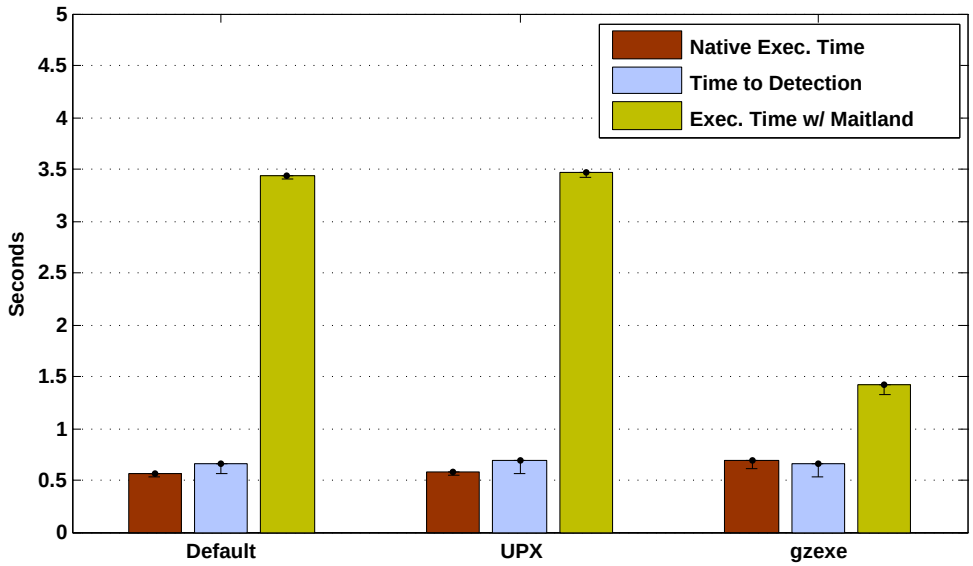

Figure 4.8: Summary of Maitland's effect on the gzip executable.

amount of time required to allocate it. This is probably due to the memory allocation optimization system inside the Linux kernel called the *slab allocator* (or *SA*) [80]. The slab allocator in the guest kernel likely had a cache of unallocated memory areas similar in size to the 160 pages being requested at the dip on the graph. In addition, Maitland's filter configuration meant it processed most events related to memory allocation and execution. Because of this, Figure 4.9 illustrates a worst-case performance penalty.

We see the time required to do the work grows rapidly with page-table size. As previously mentioned, this behavior was expected. We believe however that by modifying Maitland to cache unchanged pages and only process changed pages, Maitland could significantly reduce this explosion in workload.

4.3 Desired Goals

To determine how successful Maitland is, we compare it to the desired feature-set discussed at the end of Chapter 1.

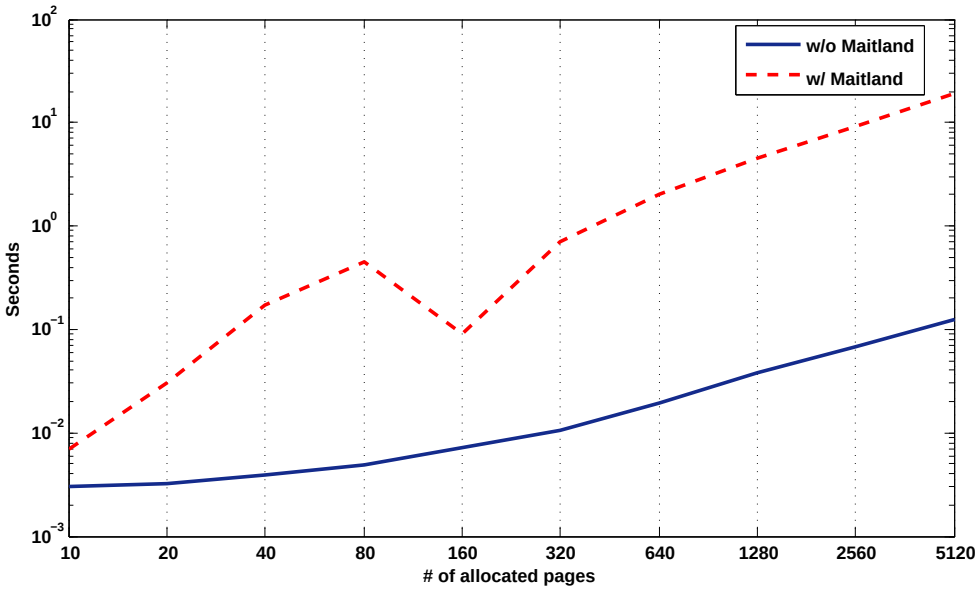

Figure 4.9: How process memory size affects execution time. (Log-Log scale)

4.3.1 Expose Packed/Encrypted Executables

Maitland's first and arguably most important feature is that it must be able to detect and expose packed or encrypted executables. Because this is an unsolved problem for all packers, it is not possible for Maitland to handle every case using only known techniques. As we demonstrated in Section 4.2.1, using a known technique, Maitland is able to unpack programs which were packed by UPX, one of the most commonly used packers by malware writers.

4.3.2 Do Not Restrict Analysis Tools

The interface Maitland provides to analysis tools is a standard file system. Because Maitland provides this interface, the demarcation point is the same as it would be for any standalone static analysis tool that takes files as input. Potentially, Maitland can provide information to a dynamic analysis tool as well. As demonstrated in Section 4.2.1, a simple UNIX grep wrapper was sufficient to detect known binary code after Maitland had exposed it. Because Maitland provides a filesystem interface, other tools can easily be used. One example is *ClamAV* [81].

4.3.3 Function Without Major Adaptations to the Platform

Maitland performs on a VM without m odifications to the hypervisor. Maitland does not have any code residing in the hypervisor, due to its implementation as a split driver. A system like Maitland can be deployed quickly without the need to take the physical machine down. This is possible because Maitland resides in the kernel space of VMs, rather than the hypervisor. This feature is important when considering deployment in a cloud.

4.3.4 Be Secure

Because Maitland resides only in guest kernels, it avoids introducing any potential vulnerabilities to the hypervisor that could be exploited. In addition Maitland's footprint is small. The code residing in the monitored VM guest kernel is around 1000 lines of C. A codebase of this size is small enough for formal code verification, which would significantly reduce the chance of potential vulnerabilities. Like SubVirt [62], Maitland could thus be formally verified.

4.3.5 Avoid Imposing Overwhelming Overhead

In our experiments, Maitland induced an overhead to programs being monitored. In experiment 2 (Section 4.2.2) we recorded a worst-case performance hit on process execution time. In experiment 3 (Section 4.2.3) we recorded an exponential growth in performance penalty as process memory size grows exponentially. The worst-case overheads seem high but we believe these can be significantly reduced by adopting the optimizations discussed in Section 5.1.1.

4.4 Additional Properties

4.4.1 Multi-Round Packed Executables

Maitland's implementation should theoretically detect and bypass multi-round packed executables provided it is capable of handling the method used in each round individually. Although Maitland can handle this case, the current configuration will see all packing rounds as one big operation because it cannot distinguish between one or more layers. It is however, certainly possible to add this feature using a known

method which can approximate when each packing stage has occurred. We discuss this method in Section 5.1.2.

4.5 Summary

In this chapter we analyzed Maitland's run-time performance. We explained our analysis and how our experiments were orchestrated. We then presented our results and provided an initial explanation of the outcome. We continued by comparing the final implementation with the high-level requirements we defined in Chapter 1, discussing each in relation to Maitland. Finally we talked about a few interesting properties which arise and how they relate to certain decisions we made. In the concluding chapter, we look at many pieces of future work and how they will help Maitland meet our high-level requirements more closely.

Chapter 5

Future Work and Conclusions

In this chapter, we look at ways Maitland can be improved based on our analysis and also wrap up some items we did not directly address. Finally, we provide our conclusions based on our experience working on Maitland.

5.1 Future Work

Maitland is an early implementation of this design. Although Maitland meets our requirements, there is still potential work to be done. In this section we discuss the possible continuation of work regarding Maitland.

5.1.1 Optimizations

5.1.1.1 Improved Fault Filtering

As was demonstrated in Section 4.2.2, Maitland's current configuration results in excessive report requests. The result is that Maitland halts the process in question to create a full report far too often, significantly affecting execution time. The frequency of report requests can be reduced by modifying Maitland to be more strict about which page faults it chooses to act upon. There are a number of ways this can be approached, but the most obvious is to filter not only more aggressively, but in a smarter way. By this we mean, taking other factors into account which Maitland currently does not. An example might be taking into account the history of the running program. Another is to consider additional system state information. Patagonix, for example, achieves a low induced overhead by tracking [60] which pages have already been

checked and are trusted. As long as the page is unmodified, it is skipped. A method like this would be relatively simple to add to Maitland. There are a number of known optimization techniques which could be easily added to Maitland.

5.1.1.2 Unchanged Page Caching

As was demonstrated in Section 4.2.3, Maitland's performance suffers with a growing memory allocation size of the program. One approach to mitigate this would be to improve Maitland's reporting system to cache a local copy of a report and use page hashes to keep track of page changes. The idea being that with some minor changes, Maitland could greatly reduce the number of pages it must make available to the analysis tool and perhaps even skip traversing segments of the process' page table structures.

5.1.1.3 Hardware Improvements

Pure software systems like Maitland can provide some insight into what parts of the process should be moved to hardware implementations. What is gained is a better understanding of what can be simply optimized and what must be moved out of software. The addition of several virtualization support features by CPU manufacturers shows this is not an unreasonable approach [53,54]. Therefore, moving features which create a significant bottleneck in pure software systems to hardware is not unreasonable. Obviously direct hardware support is not the answer for everything, but systems like Maitland help us decide where it might be best exploited.

5.1.2 Multi-Round Packing and Stage Completion Approximation

As Maitland itself does not perform analysis, the support for different variations of packing falls on the analysis tool. Multi-round packing is one such variation. A packer using this strategy will repeatedly pack the same data in a nested fashion so as to make it harder to analyze. With a relatively simple analysis tool, Maitland can aid in detecting these instances. Our initial implementation does not include an analysis tool sophisticated enough to distinguish between single and multi-round packing. It will however still expose both of these with similar effectiveness, without being aware of how many layers were involved.

Maitland currently uses a naive approach to detect unpacking stages, intentionally resulting in worst-case performance results. This means that whenever a possible unpacking operation occurs, Maitland invokes the analysis tool. Some tools like OmniUnpack [42] utilize a heuristic to approximate the end of an unpacking stage, allowing the analysis stage to only be invoked once at the end. With Maitland, we could greatly improve performance with the use of such a heuristic. The ability to count unpacking stages might also be valuable and would be available with the addition of a heuristic.

Using a heuristic technique to do such a task, however, implies the possibility of false negatives. If a system were to rely too heavily on such a heuristic, the potential to miss important system state information exists. If the system incorrectly approximated the end of the packing stage, it could result in reduced detection ability. In addition, malware can take advantage of the use of such a heuristic, tampering with the data used as input for the heuristic. The result would be that malware could influence when Maitland requests a report, potentially causing it to miss events critical for the detection of that malware.

So we see a tradeoff between an additional feature and improved security. The addition of such a feature would rely on whether or not Maitland's focus is to aid the defense of a system from malware, or to examine and learn about malware.

5.1.3 Additional Operating System Support

As previously explained, Maitland uses standard Xen facilities to function. Any OS which functions on the Xen hypervisor is capable of running Maitland. The availability of more OSes besides Linux would be a useful addition to Maitland's functionality. Although Xen supports several OSes in a paravirtualized mode, others are supported using only hardware virtualized mode. The Xen project currently supports Windows XP, Vista and 7 only in hardware virtualized mode. Although there are paravirtualized Xen drivers for the Windows platform offered by some organizations like *Citrix* [82], they are proprietary and are not a part of the Xen project itself. Because it tends be the most largely effected by malware attacks, Windows support would allow a much larger sample base on which to test and improve Maitland.

5.1.4 Remote Analysis Component

Because actual analysis is done outside of Maitland and in user-space, it is also possible to implement a proxy in the privileged VM. One application for this might be to provide a simpler, local analysis component with a cache of known signatures which forwards any heavy-duty analysis to a network service. Such a system might look like the one shown in Figure 5.1. An incredibly useful application of such a system might be for a cloud provider. A network service would be able to do upfront work, forwarding hashes and other meta data down to caches on the monitored VMs. This would make it easy for groups of VMs to quickly respond to anything which has already been detected by banning that process from being run.

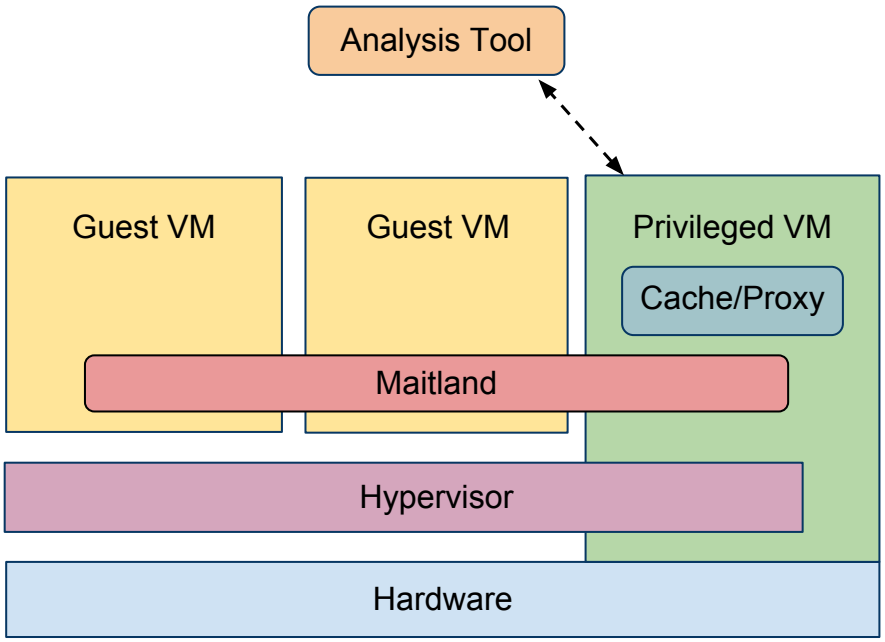

Figure 5.1: Hypothetical extension for Maitland's architecture.

Malware analysis is often a computationally intensive task. Depending on the type of analysis, heavy CPU, memory, or IO load can occur. For this reason, it makes sense to have the ability to pull that work outside of the VM being monitored so as not to cause outages or slowdowns to services. Using this mentality, the obvious thing would be to have a dedicated VM responsible for analysis of other guests sharing a hypervisor. Maitland's design encourages this. Maitland serves as a specialized light-weight monitoring layer between VMs which can funnel interesting events from

monitored guests directly to a dedicated analysis VM. Within this dedicated VM, one can have all the tools for analysis in one centralized place. The benefit is that a system can react very quickly in the context of a single hypervisor. If a program on one machine is unpacked and determined to be malicious, Maitland could potentially ban that program from being executed on other VMs in it's jurisdiction. Propagating this information quickly would protect the local VMs from that particular instance of malware. This information could also be sent to a network service to propagate it to other hypervisor instances.

5.1.5 Interpreters

Interpreters are an interesting case. An interpreter causes the same symptoms that a packer might cause while it interprets and executes bytecode. Our initial design and implementation of Maitland has no solution for dealing with interpreters. However, because Maitland runs partially inside the guest VM OS kernel, it is possible to build an extension to interface with an interpreter. An extension or custom interpreter could provide assistance in examining interpreted programs within the interpreter's environment. Hypothetically, one could have a trusted interpreter to provide extensions into it's internals for Maitland to utilize, allowing Maitland to monitor programs running in the interpreter. This could be applied to a standalone interpreter, a web browser, or a JIT compiler.

5.1.6 Tradeoffs

What has been the ongoing theme through much of our research is the idea of managing tradeoffs. Where one system specializes in providing highly granular information at the cost of performance, another may use heuristics to approximate and improve performance at the cost of granularity. These variables define what the resulting system specializes in, detection and protection versus analysis and examination.

Our goal for Maitland was to build a system which would support as many variables as possible and allow anyone to choose which trade-offs suited them best rather than requiring them to build their own system to meet those needs. We envisioned other researchers, organizations or communities making use of Maitland by adding additional features, options and variables. These variables allow someone like a cloud provider to tune and set the trade offs in a way that is beneficial for them.

5.2 Conclusions

In this thesis, we looked at the current state of the art in the detection and exposure of packed and encrypted malware. Detailed background was provided in many of the technical areas, helping to better understand the scope of the problem.

Although the problem is by no means solved, there are relatively effective solutions shown by previous research. That is not to say there is not more work to be done however. The baked-in, integrated nature of the predominant solutions leaves much to be desired for achieving adoption by infrastructure providers. Additionally, the push to virtualization requires a solution that is geared to the unique circumstances of such a system.

We presented Maitland, our attempt at a solution which focuses on simple deployment in a virtualized environment. Maitland's goal to achieve what previous work has, is done using a virtualization platform as a tool but without requiring changes or any non-standard features normally found in such a system.

Our experiments showed promising results in detection rates of our sample packed malware in a virtualized environment. We provided an analysis of Maitland, both regarding worst-case performance and malware sample detection rates. This analysis resulted in explanations as well as suggestions for improvements and future work.

Maitland demonstrates a practical approach to the problem of detecting and exposing packed and encrypted malware in a virtualized environment.

Bibliography

[1] K. Timm, "Malware Validation Techniques," 2010. `http://blogs.cisco.com/security/malware_validation_techniques` [Accessed Oct. 5, 2011].

[2] C. Economics, "2005 Malware Report: Executive Summary," 2006. `http://www.computereconomics.com/article.cfm?id=1090` [Accessed Oct. 7, 2011].

[3] U. Bayer, C. Kruegel, and E. Kirda, *TTAnalyze: A Tool for Analyzing Malware*. PhD thesis, Technical University of Vienna, 2006.

[4] A. Moser, C. Kruegel, and E. Kirda, "Exploring Multiple Execution Paths for Malware Analysis," *2007 IEEE Symposium on Security and Privacy SP 07*, vol. 0, pp. 231–245, 2007.

[5] P. Morley, "Processing virus collections," *VIRUS*, vol. 129, no. September, pp. 129–134, 2001.

[6] N. Idika and A. Mathur, "A survey of malware detection techniques," *Purdue University*, 2007.

[7] M. R. Chouchane and A. Lakhotia, "Using engine signature to detect metamorphic malware," *Proceedings of the 4th ACM workshop on Recurring malcode (WORM '06)*, p. 73, 2006.

[8] I. Santos, Y. Penya, J. Devesa, and P. Bringas, "N-Grams-based file signatures for malware detection," in *Proceedings of the 11th International Conference on Enterprise Information Systems (ICEIS)*, pp. 317–320, 2009.

[9] A. Sung, J. Xu, P. Chavez, and S. Mukkamala, "Static Analyzer of Vicious Executables (SAVE)," *20th Annual Computer Security Applications Conference*, pp. 326–334, 2004.

[10] D. Bruschi, L. Martignoni, and M. Monga, "Using code normalization for fighting self-mutating malware," in *Proceedings of International Symposium on Secure Software Engineering*, IEEE, 2006.

[11] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha, "An architecture for generating semantics-aware signatures," in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, (Berkeley, CA, USA), USENIX Association, 2005.

[12] G. Hu and D. Venugopal, "A Malware Signature Extraction and Detection Method Applied to Mobile Networks," *2007 IEEE International Performance, Computing, and Communications Conference*, pp. 19–26, Apr. 2007.

[13] K. Griffin, S. Schneider, X. Hu, and T.-c. Chiueh, "Automatic generation of string signatures for malware detection," in *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, RAID '09, pp. 101–120, Springer-Verlag, 2009.

[14] M. Fossi, D. Turner, E. Johnson, T. Mack, T. Adams, J. Blackbird, S. Entwisle, B. Graveland, D. McKinney, J. Mulcahy, and C. Wueest, "Symantec Global Internet Security Threat Report: Trends for 2009," vol. XV, Apr 2009.

[15] M. Fossi, G. Egan, K. Haley, E. Johnson, T. Mack, T. Adams, J. Blackbird, L. Mo King, D. Mazurek, D. McKinney, and P. Wood, "Symantec Internet Security Threat Report: Trend for 2010," vol. XVI, Apr 2011.

[16] S. M. Tabish, M. Z. Shafiq, and M. Farooq, "Malware Detection using Statistical Analysis of Byte-Level File Content Categories and Subject Descriptors," *Architecture*, pp. 23–31, 2009.

[17] J. Z. Kolter and M. A. Maloof, "Learning to Detect and Classify Malicious Executables in the Wild," *Journal of Machine Learning Research*, vol. 7, pp. 2721–2744, 2006.

[18] J. Lee, K. Jeong, and H. Lee, "Detecting Metamorphic Malwares using Code Graphs," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pp. 1970–1977, ACM, 2010.

[19] R. Lyda and J. Hamrock, "Using Entropy Analysis to Find Encrypted and Packed Malware," *IEEE Security And Privacy*, vol. 5, no. 2, pp. 40–45, 2007.

[20] D. Quist and L. Liebrock, "Visualizing Compiled Executables for Malware Analysis," in *Proceedings of the Workshop on Visualization for Cyber Security*, pp. 27 –32, IEEE, Oct 2009.

[21] M. Christodorescu and S. Jha, "Testing Malware Detectors," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, vol. 29 of *ISSTA '04*, p. 34, ACM, 2004.

[22] C. Willems, T. Holz, and F. Freiling, "Toward Automated Dynamic Malware Analysis Using CWSandbox," *IEEE Security and Privacy Magazine*, vol. 5, pp. 32–39, Mar. 2007.

[23] M. D. Preda, M. Christodorescu, S. Jha, and S. Debray, "A Semantics-Based Approach to Malware Detection," *ACM Transactions on Programming Languages and Systems*, vol. 30, pp. 1–54, Aug. 2008.

[24] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware Analysis via Hardware Virtualization Extensions," in *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pp. 51–62, ACM, 2008.

[25] A. Moser, C. Kruegel, and E. Kirda, "Limits of Static Analysis for Malware Detection," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pp. 421–430, IEEE, Dec 2007.

[26] U. Bayer, P. P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering," in *Network and Distributed System Security Symposium (NDSS)*, Feb 2009.

[27] A. Stepan, "Improving proactive detection of packed malware.," *Virus Bulletin*, vol. 1, no. March, 2006.

[28] G. Taha, "Counterattacking the packers," *McAfee Avert Labs, Aylesbury, UK*, 2007. `http://yaokai.org/dl/avar2007/data/22_Counterattackingthepackers/Counterattackingthepackers.pdf` [Accessed Oct. 3, 2011].

[29] P. Bustamante, "Mal(ware)formation statistics," 2007. `http://research.pandasecurity.com/malwareformation-statistics` [Accessed Oct. 5, 2011].

[30] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee, "Eureka: A framework for enabling static malware analysis," in *Computer Security - ESORICS 2008*, vol. 5283 of *Lecture Notes in Computer Science*, pp. 481–500, Springer Berlin / Heidelberg, 2008.

[31] Kaspersky, "Virus.Win32.Sality.bh," 2011. `http://www.securelist.com/en/descriptions/15312802/Virus.Win32.Sality.bh#doc1` [Accessed Oct. 5, 2011].

[32] J. Larimer, "An inside look at Stuxnet," 2009. `http://blogs.iss.net/archive/papers/ibm-xforce-an-inside-look-at-stuxnet.pdf` [Accessed Oct. 8, 2011].

[33] F. Guo, P. Ferrie, and T. Chiueh, "A study of the packer problem and its solutions," in *Recent Advances in Intrusion Detection*, pp. 98–115, Springer, 2008.

[34] W. Yan, Z. Zhang, and N. Ansari, "Revealing packed malware," *IEEE Security and Privacy Magazine*, vol. 6, no. 5, pp. 65–69, 2008.

[35] "Annual Report PandaLabs 2009," 2009. `http://www.pandasecurity.com/img/enc/Annual_Report_Pandalabs_2009.pdf` [Accessed Nov. 2, 2011].

[36] M. Oberhumer, L. Molnar, and J. Reiser, "UPX: the Ultimate Packer for eXecutables (2007)," 2007. `http://upx.sourceforge.net` [Accessed Oct. 5, 2011].

[37] Oreans Technologies, "Themida." `http://www.oreans.com` [Accessed Oct. 5, 2011].

[38] ASPack Software, "ASPack." `http://www.aspack.com` [Accessed Oct. 8, 2011].

[39] PEid, "PEid - Packer Detector." `http://www.peid.info` [Accessed Oct. 5, 2011].

[40] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant, "Semantics-Aware Malware Detection," *2005 IEEE Symposium on Security and Privacy (S&P'05)*, pp. 32–46, 2005.

[41] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware," *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pp. 289–300, Dec. 2006.

[42] L. Martignoni, M. Christodorescu, and S. Jha, "OmniUnpack: Fast, Generic, and Safe Unpacking of Malware," *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pp. 431–441, Dec. 2007.

[43] M. G. Kang, P. Poosankam, and H. Yin, "Renovo : A Hidden Code Extractor for Packed Executables," in *Proceedings of the 2007 ACM workshop on Recurring malcode*, WORM '07, pp. 46–53, Oct 2007.

[44] Q. Zhang, *Polymorphic and metamorphic malware detection.* PhD thesis, North Carolina State University, 2008.

[45] A. Walenstein, R. Mathur, M. Chouchane, and A. Lakhotia, "Normalizing metamorphic malware using term rewriting," in *Source Code Analysis and Manipulation, 2006. SCAM'06. Sixth IEEE International Workshop on*, pp. 75–84, IEEE, 2006.

[46] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *13th Proceedings of USENIX Security*, pp. 255–270, 2004.

[47] A. G. Gleditsch, "Linux Cross Reference." `http://lxr.linux.no/linux+v2.6.32` [Accessed Oct. 2, 2011].

[48] J. Blanchette, M. Summerfield, M. Gorman, A. J. Massa, and N. Mcfarlane, *Understanding the Linux Virtual Memory Manager.* Prentice Hall, 2004.

[49] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pp. 164–177, ACM, 2003.

[50] "XenOverview," 2011. `http://wiki.xensource.com/xenwiki/XenOverview` [Accessed Oct. 1, 2011].

[51] D. Chisnall, "The definitive guide to the xen hypervisor," *Journal of the Electrochemical Society*, vol. 129, 2007.

[52] J. Pfoh, C. Schneider, and C. Eckert, "A formal model for virtual machine introspection," *Conference on Computer and Communications Security*, pp. 1–10, 2009.

[53] Intel, "Intel Virtualization Technology (Intel VT)." `http://www.intel.com/technology/virtualization/technology.htm` [Accessed Oct. 2, 2011].

[54] AMD, "AMD-V Nested Paging," *Advanced Micro Devices*, 2008.

[55] J. Rutkowska and A. Tereshkin, "Bluepilling the Xen Hypervisor," *Invisible Things Lab*, 2008. `http://invisiblethingslab.com/bh08/part3.pdf` [Accessed Oct. 3, 2011].

[56] S. King, P. Chen, C. Verbowski, H. Wang, and J. Lorch, "SubVirt: Implementing malware with virtual machines," *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pp. 314–327, 2006.

[57] J. Rutkowska, "Redpill," 2004. `http://www.invisiblethings.org/papers/redpill.html` [Accessed Oct. 3, 2011].

[58] D. Quist and V. Smith, "Detecting the Presence of Virtual Machines Using the Local Data Table," *Offensive Computing*, 2006.

[59] R. Wojtczuk, "Subverting the Xen hypervisor," *Black Hat USA*, 2008.

[60] L. Litty, H. A. Lagar-Cavilla, and D. Lie, "Hypervisor support for identifying covertly executing binaries," in *Proceedings of the 17th conference on Security symposium*, (Berkeley, CA, USA), pp. 243–258, USENIX Association, 2008.

[61] L. Litty and D. Lie, "Manitou: a layer-below approach to fighting malware," in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pp. 6–11, ACM, 2006.

[62] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor : A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes," in *Proceedings of 21st ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pp. 335–350, ACM, 2007.

[63] J. Franklin, A. Seshadri, N. Qu, S. Chaki, and A. Datta, "Attacking, Repairing, and Verifying SecVisor: A Retrospective on the Security of a Hypervisor," *Technical Report, Carnegie Mellon University*, 2008.

[64] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003)*, vol. 37, (New York, NY, USA), pp. 193–206, ACM, Oct 2003.

[65] A. M. Nguyen, N. Schear, H. Jung, A. Godiyal, S. T. King, and H. D. Nguyen, "Mavmm: Lightweight and purpose built vmm for malware analysis," in *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pp. 441–450, IEEE Computer Society, 2009.

[66] R. Russell, "lguest : Implementing the little Linux hypervisor," *Proceedings of the Linux Symposium*, vol. 2, 2007.

[67] Amazon, "Amazon EC2 FAQs," 2011. `http://aws.amazon.com/ec2/faqs`.

[68] StratusLab, "EC2, Xen and Paravirtualization." `http://stratuslab.eu/doku.php/ec2_xen_and_paravirtualization` [Accessed Oct. 5, 2011].

[69] P. J. Salzman, M. Burian, and O. Pomerantz, "Talking to Device Files (writes and IOCTLs)." `http://linux.die.net/lkmpg/x892.html` [Accessed Oct. 2, 2011.

[70] "Intel 64 and IA-32 Architectures Software Developers Manual, Combined Volumes," *Intel Corporation*, Oct 2011. `http://download.intel.com/products/processor/manual/325462.pdf` [Accessed Oct. 5, 2011].

[71] N. Aki, "Enabling User Provided Kernels in Amazon EC2." `https://forums.aws.amazon.com/servlet/JiveServlet/download/30-51562-194272-3595/user_specified_kernels.pdf`.

[72] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, vol. 34, (New York, NY, USA), pp. 2–13, ACM, Oct. 2006.

[73] O. Whitehouse, "An analysis of address space layout randomization on Windows Vista," *Symantec Advanced Threat Research*, pp. 1–14, 2007. `http://www.mobile-download.net/tools/Miscellaneous/Address_Space_Layout_Randomization.pdf` [Accessed Oct. 10, 2011].

[74] T. Klein, "ScoopyNG." http://www.trapkit.de/research/vmm/scoopyng/index.html [Accessed Oct. 5, 2011].

[75] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the Linux virtual machine monitor," in *Proceedings of the Linux Symposium*, vol. 1, pp. 225–230, 2007.

[76] SafeNet, "The Sentinel HASP Envelope." http://www.intech.ro/hasp/uploads/whitepaper/The-Sentinel-HASP-Envelope-WP-EN-web.pdf, 2011.

[77] S. Talpalaru, "exepak," 2005. http://exepak.sourceforge.net [Accessed Oct. 5, 2011].

[78] T. Ooura, "Ooura's Mathematical Software Packages," 2006. http://www.kurims.kyoto-u.ac.jp/~ooura [Accessed Oct. 5, 2011].

[79] J.-l. Gailly and M. Adler, "The gzip home page." http://www.gzip.org [Accessed Oct. 6, 2011].

[80] B. Fitzgibbons, "The linux slab allocator," 2000. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.26.9588.

[81] ClamAV, "ClamAV." http://www.clamav.net/lang/en [Accessed Oct. 14, 2011].

[82] Citrix, "3.2. Windows paravirtualized drivers." http://docs.vmd.citrix.com/XenServer/4.0.1/guest/ch03s02.html [Accessed Nov. 24, 2011].

Appendix A

Test Environment

A.1 Test Machine Hardware

Related output from *dmidecode* command.

```
BIOS Information
Vendor: IBM
Version: IBM BIOS Version 1.49-[PME149AUS-1.49]-

System Information
Manufacturer: IBM
Product Name: 622310U

Base Board Information
Manufacturer: IBM
Product Name: MSI-9151 Boards

Processor Information (x2)
Family: Xeon MP
Manufacturer: Intel Corporation
Version: Intel(R) Xeon(TM) CPU 3.40GHz
Voltage: 1.3 V
External Clock: 200 MHz
Max Speed: 3600 MHz
Current Speed: 3400 MHz
```

Memory Array Mapped Address
Range Size: 2560 MB

A.2 Test Machine Software

Kernel: Debian 2.6.32.27 SMP x86_64 GNU/Linux
Xen 4.0.1 - From Source

A.3 Test VM Environment

Kernel: Debian 2.6.32.27 SMP x86_64 GNU/Linux Paravirt Tools
1 VCPU
128MB Memory