AN OVERVIEW OF ANTI-VIRTUALIZATION TECHNOLOGY AND DETECTING VIRTUALIZATION IN USER SPACE

by
SAMBITESH DASH
(Under the Direction of Dr. KANG LI)

ABSTRACT

Virtual Machines or virtualization technology has become very popular in today's computing world. A host of services are being provided on a virtual platform. Hence detecting virtual environment is an important goal for malware attackers and vendors of software services both. In this thesis we study all such attacks, categorize them and observe how Virtual Machine vendors defend against them. Apart from all this we also experiment and find out effectiveness of various categories of attacks on different virtual machine types and hypervisor designs. We also design a software that detects virtualization in user space i.e. without root access/admin rights.

AN OVERVIEW OF ANTI-VIRTUALIZATION TECHNOLOGY AND DETECTING VIRTUALIZATION IN USER SPACE

by

SAMBITESH DASH

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA
2015

AN OVERVIEW OF ANTI-VIRTUALIZATION TECHNOLOGY AND DETECTING VIRTUALIZATION IN USER SPACE

by

SAMBITESH DASH

Major Professor:	Kang Li
Committee:	Ismailcem Budak Arpinar
	Lakshmish Ramaswamy

Electronic Version Approved:

Suzanne Barbour
Dean of the Graduate School
The University of Georgia
December 2015

ACKNOWLEDGEMENTS

I would like to thank my parents without whom my life would not be possible. I would like to thank my Major advisor and advisory committee for their constant guidance and support. They were patient throughout my many errors and slip ups.

- Dr. Kang Li (major advisor)
- Dr. Ismailcem Budak Arpinar
- Dr Lakshmish Ramaswamy

Some of the work in this thesis is partially funded by National Science Foundation (NSF) under award No. 1319115.

In the end I would like to extend my thanks to Khushboo Baghadiya and Nilayan Bhattacharya for their useful inputs while I was writing this thesis.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ABI	Application Binary Interface
GDT	Global Descriptor Table
IDS	Intrusion Detection System
IDT	Interrupt Descriptor Table
LDT	Local Descriptor Table
OS	Operating System
TLB	Translation Lookaside Buffer
TSC	Time Stamp Counter
VM	Virtual Machine
VME	Virtual machine Environment
VMM	Virtual Machine Monitor

Chapter 1

Introduction

Virtual machine concept was first suggested in a 1975 paper by J Donovan [1]. Since then in view of rapid increase in computing speeds, use of virtual machines has become increasingly popular. Virtual machines are used to solve a larger number of computing problems. The most important of which is **Cloud Computing.** Virtualization forms the backbone of cloud computing [2]. Apart from it, Virtual Machines are used for "sandboxing". Sandboxing is a very common technique in analysis of malwares. So it is very important that malwares do not detect that they are running in a virtual environment.

Virtual machine is also gaining popularity in personal use. Since Linux based Operating Systems are getting increasingly popular it is very common to have Linux virtual environment in a Windows system and vice versa. Therefore software vendors are often interested in knowing whether their software is being run in a virtual environment or in native OS.

1.1 Background and related work

One of the most popular anti VM rootkit was introduced by Joanna Rutkowska in Black Hat Conference 2006 LA [3]. It relied on using a single CPU instruction called SIDT to detect the virtual machine environment. This method was developed keeping the old x86 architecture in mind. Since it didn't fully support virtualization [4], certain OS artifacts like *Interrupt Descriptor Table, State Task Register, Global*

Descriptor Table etc. had to be created elsewhere in memory for use by VMs. Rutkowska exploited this to detect a virtual environment.

Other methods soon evolved once architecture started supporting Virtualized environments. Also vendor specific attacks were being implemented by the malwares. In *Virtual PC* a dump of *baseboard class* was taken [5]. *Baseboard class* contains a lot of BIOS information and can be used to get information about the motherboard. By comparing values of *Manufacturer, Serial number, Version* etc it was possible to identify the virtual machine.

Apart from it an inspection of hard disk vendor name, video card vendor name, MAC addresses also revealed enough information to detect virtual machines [7]. Vendor names would have a string containing the name of virtual machine vendor. Virtual machines would often use bogus MAC addresses. A simple mac address lookup [6] can then confirm if it belongs to a real network or a virtual network.

The heart of a Virtual Machine is a Virtual Machine Monitor or Hypervisor. Some VMMs used nonstandard machine instructions [7]. Virtual PC's VMM used non IA32 instructions and this was exploited for detection. VmWare provided mechanism for communication between Host and Guest system[7]. Finding out this backdoor channel can be used to detect a VMWare virtual environment.

Another famous category of attacks are the timing analysis attacks [14]. They are effective even when there is hardware assistance and the Virtual Machine has been customized to avoid detection. One of the earlier form of attacks was to read the **Time Stamp Counter (TSC)** using the CPU instruction called ***rdtsc***. By observing the difference in values in two separate calls to rdtsc a virtual environment can be recognized. Later many virtual machines started implementing their own TSC. So a different approach to timing analysis was taken. There are certain machine level instructions which are privileged for a virtual machine like ***cpuid*** [14]. Any call to this instruction will lead to a trap and hence will take more time to

process in a virtual environment. But an operation like ***noop*** is unprivileged in both host system and guest system. So by running ***noop*** and ***cpuid*** in two separate threads and observing the difference in time taken to process them can belie the virtual environment.

1.2 Problem Statement

Virtual machine detection has been an arms race. The architecture of virtual machines has kept evolving and so has the complexity of attacks. In this thesis we take a look at this arms race. We find out the state of the art in Virtual machine detection. We also identify how each attack is overcome by the vendors. We come up with a Taxonomy for these attacks.

We also study how the architecture of a Virtual Machine affects detection mechanism.

1.3 Research Method

In this thesis we mainly concentrate on hosted VMMs like **VMWare and VirtualBox** and emulated VMMs like **Qemu.**

For some of the anti-virtualization techniques we write **C codes** and test them on above mentioned environments.

1.4 Contributions

I have presented and tested the validity of following four hypothesis in this thesis. They are:-

Hypothesis 1 – Type -2 virtual machines can hide from traditional attacks.

Type 2 virtual machines are those where the Virtual Machine manager is installed on top of a real OS.

From traditional attacks were here refer to the types of attacks discussed in **Chapter 3**. But here is a brief description of all such attacks.

CPU semantic attacks refer to attacks that exploit the mismatch between a real CPU and the virtual CPU. Virtual Machines need virtual CPU to simulate all the functions carried on by a real CPU – memory allocation, control, instruction flow etc. String attacks arise from Virtual Machines writing its own files into memory or drive and in the process belying its existence. There are miscellaneous attacks which look for special instructions or special commands in Instruction Set Architecture.

Hypothesis 2 – Type – 1 virtual machines can hide from traditional attacks.

Type 1 virtual machines are those where the Virtual Machine Manager lies on top of hardware itself. The real OS runs on top of the virtual machine.

Hypothesis 3 – Virtual Machine emulators can hide from traditional attacks.

Emulators are hardware independent that can translate one set of Instruction Set Architecture to another. They are anyways good at hiding from anti-virtualization attacks.

Hypothesis 4 – Timing Analysis can help detect the vendor of a Virtual Machine

Timing analysis relies on difference between running times of a privileged and non-privileged instruction. The degree of this difference should vary vendor to vendor and can be suitably used to detect the virtual machine.

To test the hypothesis I implemented the following :

- Wrote a script for Virtual machine detection that exploited two anti-virtualization techniques which do not exist in any of the existing Virtualization Detection scripts.

- **Timing Analysis**
- **TLB Profiling**

- Made a comprehensive survey of the state of the art in anti-virtualization techniques. This updated the earlier work done by Xu Chen et al. in 2009.

I tested the effectiveness of my script compared to two existing scripts:

virt-what – which comes with fedora distribution.

http://people.redhat.com/~rjones/virt-what/

imvirt – which comes as a part of Ubuntu.

http://linux.die.net/man/1/imvirt

I will next discuss the strength and weakness of anti-virtualization techniques used in my script and *virt-what* and *imvirt.*

Chapter 2

Virtual Machine and its Architecture

2.1 Virtualization and Virtual Machine

Modern computers are a complex machine, composed of many subsystems which need to interact with one another. One way today's computer handle this complexity is through *abstraction.* There are levels of abstraction. These abstraction have well defined interfaces to interact with each other. One advantage of such a structure is that while designing high level components, we can ignore what lies in the lower level. Take for instance the case of a hard disk drive. Internally the disk is divided into sectors, tracks etc. But the Operating System (OS) simply sees it as files of variable size. This is illustrated in the figure below.

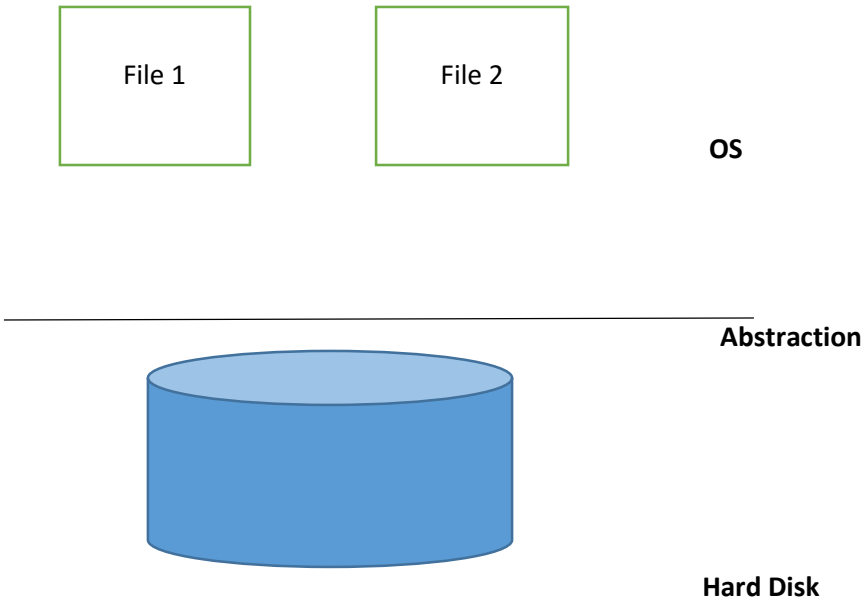


Fig.1 An abstraction of hard Disk

Earlier we mentioned that abstraction has levels of hierarchy. The lower levels are implemented at the hardware level. The properties and rules at this level are defined so as to connect various physical components together. In the higher levels we have software interface and all the rules and properties here at logical level. **Virtualization** is introduced at this boundary of *hardware-software* levels. At this level the software is made independent of the machine it runs on.

One way to achieve this independence is through **Instruction Set Architecture (ISA).** ISA is the *well-defined interface* that decouples the hardware and software development. For e.g. Intel processors come with IA-32 instruction set. People can then code compilers that translate high level programming language to these IA-32 instruction set. This way people involved with developing compilers need not worry about the design of processors.

However, there is still a limit to the mobility introduced by these by these defined interfaces. They limit inter-portability. A binary translated IA-32 instruction set will not run on machine having IBM PowerPC instruction set. Binaries for Windows won't run in Linux OS.

This is where ***Virtualization*** comes into play. When we *virtualize* a system or subsystem, we simply map some interface to an interface of a real system. From the other side it appears that the real system has been transformed to system which we need. More formally

"*Virtualization involves the construction of an isomorphism that maps a guest system to a real host*" (Popek and Goldberg 1974).

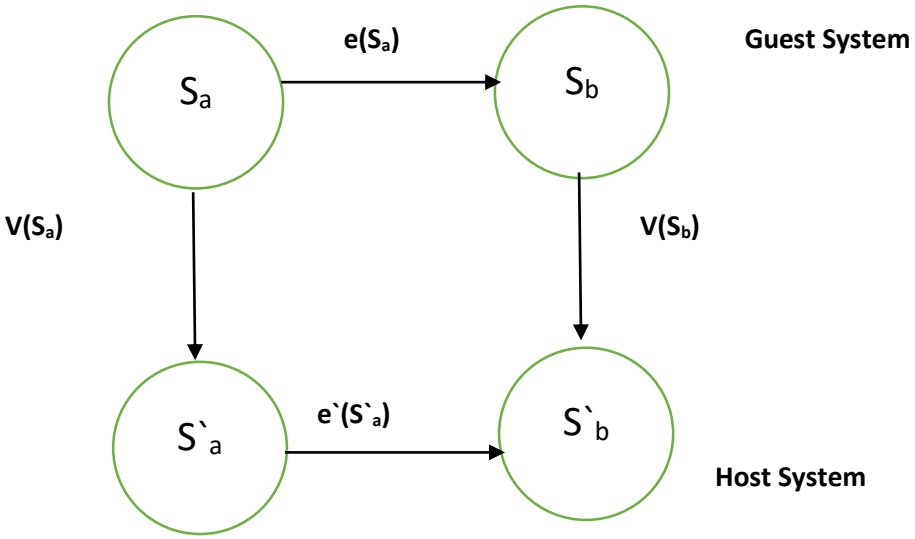


Fig 2 - Isomorphism

The figure above illustrates this isomorphism. For a series of step e that modifies the guest state, there is a corresponding sequence of operations e` that brings about a similar modification in the host system.

When the concept of virtualization is applied to entire system we get a **Virtual Machine (VM)**. A VM is implemented by adding a layer of software to the host machine to transform it into our desired architecture. For instance, we can virtualize a PC (Windows + Intel machine) to run Linux on it. This way we can run all Linux programs too.

Before we further proceed with our discussion on Virtual Machines, it is important to mention about the ***Application Binary Interface*** (ABI). A user level program uses ABI to access services provided by the system and the hardware resources. The ABI has two major components broadly speaking. The first is a set of all user instruction that help interact with system resource. Application level programs access the hardware resources by invoking the operating system through *system calls.* System calls are the other major component of an ABI.

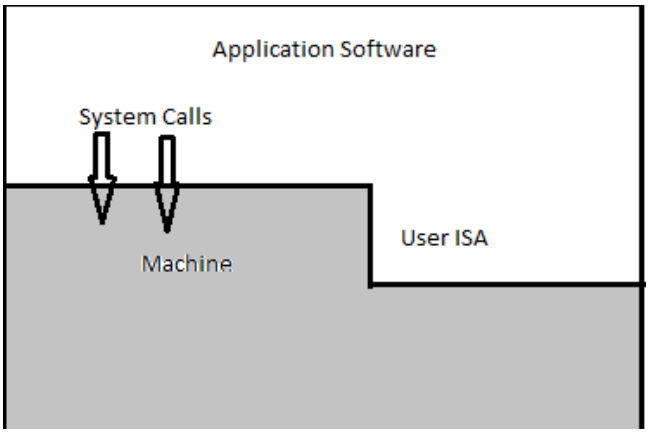


Fig 3 – Major components of ABI

2.2 Types of Virtual Machine

A virtual machine can be categorized into ***Process Virtual Machine*** *and* ***System Virtual Machine***.

Process Virtual Machine - A process virtual machine is capable of giving support to an individual process. The virtualizing software is placed above the OS/Hardware combination, at the ABI interface. In a way the process VM virtualizes the ABI. It emulates both the user ISA and system calls to OS.

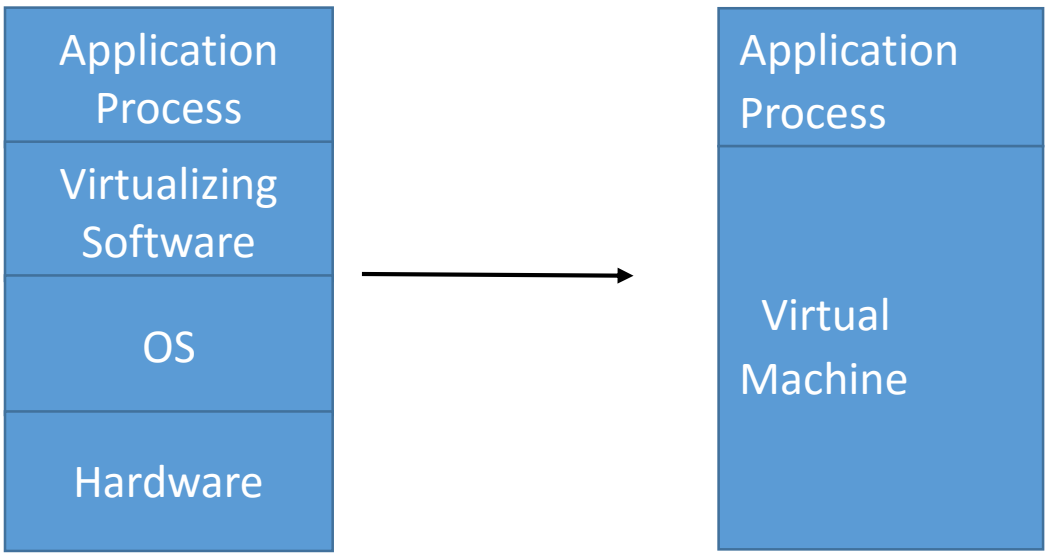
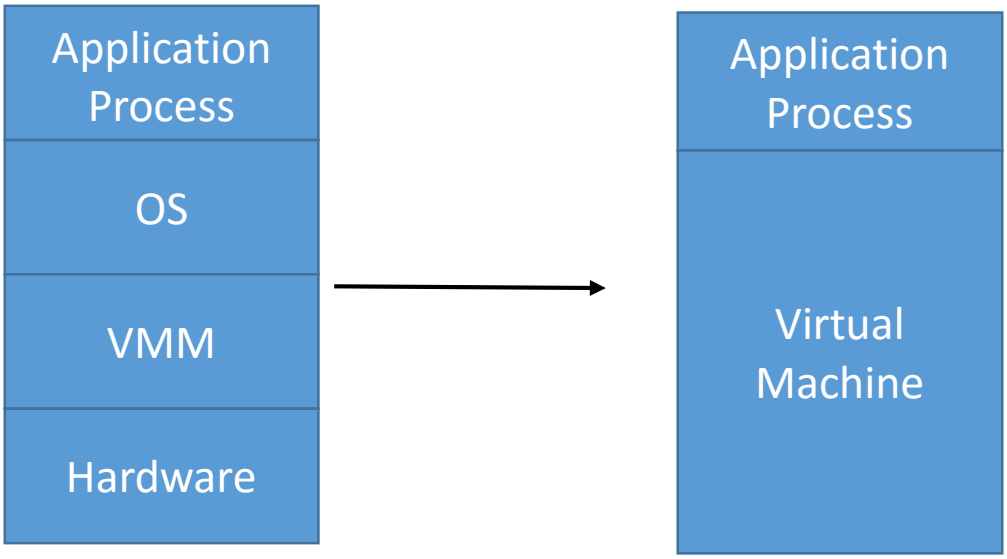


Fig. 4 Process Virtual machine

System Virtual Machine – In a system virtual machine the virtualizing software is placed right above the hardware and the OS lies on top of the virtualizing software. The virtualizing software is also called the **Virtual Machine Monitor (VMM)**.



In this thesis we are dealing with System Virtual Machines. The first Virtual Machines developed in 1960s and 1970s were System Virtual Machines. Today with increase in processing speeds and increased capacity storage devices such as RAMs, there has been renewed interest in System Virtual Machines. Let us take a look at the many different ways a System Virtual Machine is implemented.

2.3 Implementation of System Virtual Machine

- **Classic approach or the Hosted Virtual machine:** Popek and Goldberg, 1974 [20], first suggested the classic approach to VM as shown in figure - . The VMM is installed right on top of the

hardware. And then the Virtual Machine is installed over it. The OSs hosted on top of the VMM are called *guest OS.* The VMM traps all hardware and I/O requests of these guest OS. All this happens in a completely transparent way. This approach provides high efficiency since all guest are serviced in an equitable way. But it has its drawbacks too. If this has to be installed on a Desktop then the entire system has to be wiped clean and the VMM then installed on top of it. Also it has to be made sure that all I/O drivers are installed in the VMM since VMM controls all I/O operations. An alternative to this approach is what is called the *hosted VM*. In this approach the VM is installed in an existing host OS. The VMM relies on this host OS for I/O device drivers and other low level services. Since it introduces an additional software layer for accessing OS services, it is less efficient than the original classic approach.

- **Emulation or Whole System VMs –** In the classic approach all the software systems use the ISA of the underlying hardware, whether it be guest OS or application software. But in real life we often run into cases of different systems implementing different ISA. The most popular example being Macintosh using IBM PowerPC and Windows using the IA-32 ISA. This can lead to problems like sometimes it so happens that a software package for a particular hardware-OS system is no available. This lead to development of VMs where the complete software system – both OS and application programs can be implemented on a host system which implements a different ISAs. They are called whole system VMs because they virtualize every software. They use *emulation* via e.g. *binary translation* to achieve this complete virtualization. A common way to implement whole system VMs is pretty similar to hosted VMs described above. The VMM and guest OS are implemented on top of a host OS running on the hardware. Fig below illustrates it.

As we said earlier that binary translation is used to achieve this emulation. This translated code cannot always take advantage of features provided by the ISA like virtual memory management, trap handling etc. Challenges also arise when it comes to handling differences in hardware resources.

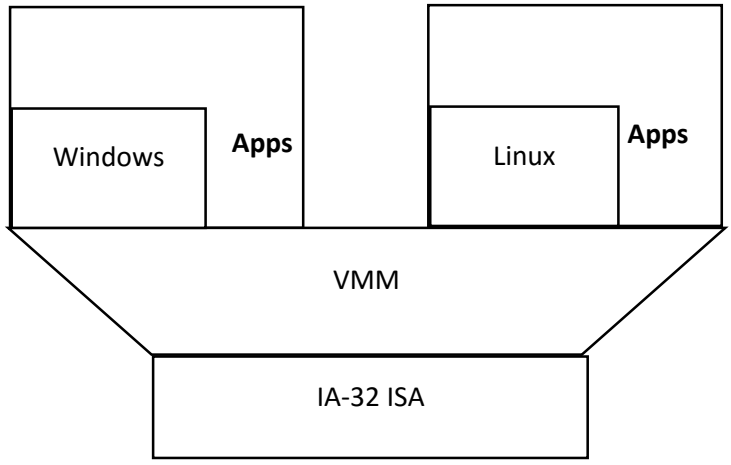


Fig 6.1 – Classic system virtual machine

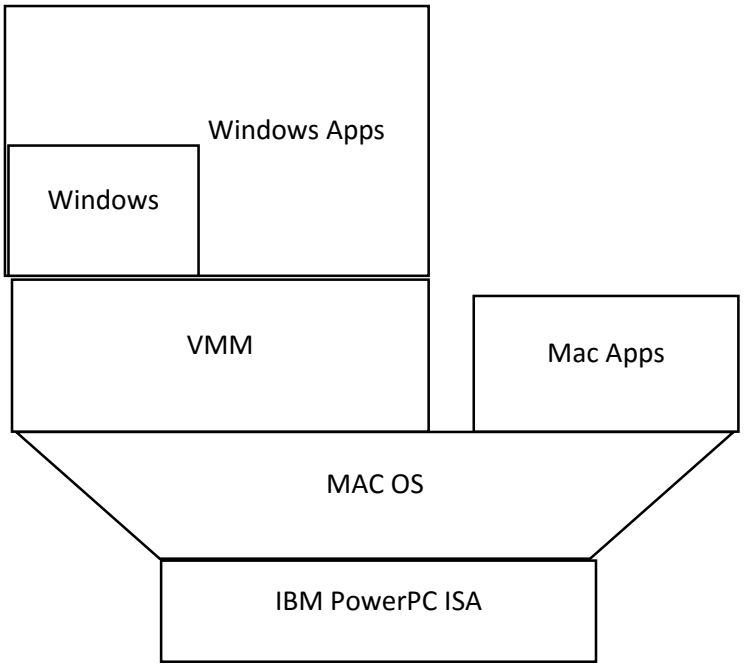


Fig 6.2 – Hosted System Virtual machine

Understanding internal working of VMs can give us key insights into how malware developers attack them. At the heart of the VM lies the virtualizing software or VMM. Popek and Goldberg laid down three conditions for a VMM to be considered a true VMM – efficiency, resource control and equivalence. *Efficiency* means that all non-privileged instruction should be executed natively without requiring any intervention from the VMM. *Resource control* implies that the guest OS should not be allowed to control or change any system resource made available to it like memory or I/O channels. *Equivalence* means that behavior of an application program should be similar while running on the guest OS and also on the host OS on the native hardware. There are few exceptions allowed to this rule though.

- Performance degradation due to emulation through binary translation is allowed
- Availability of resources being limited is allowed. Since VMs share the same disk space for example, so it is expected that they won't be getting as much space as application running on native hardware will have access to.
- An extra layer of software changes the timing relationship e.g. between I/O and processor. Any change in performance arising out of it is allowed too.

However many modern day VMMs falter on these criteria of *efficiency, resource control and equivalence.* In the next section we will take a look at how VMs implement resource virtualization.

2.4 Resource Virtualization

Memory: In a normal machine, a virtual memory gets translated with the help of *Page Tables* and Translation Lookaside Buffer (TLB) directly to a physical memory location. But for a VM, the virtual memory gets translated to a memory address that appears real but is not. We will refer to it as the "real memory" to differentiate it from physical memory. This real memory then translated to an actual physical memory location. The schematics of this translation is shown below.[12]

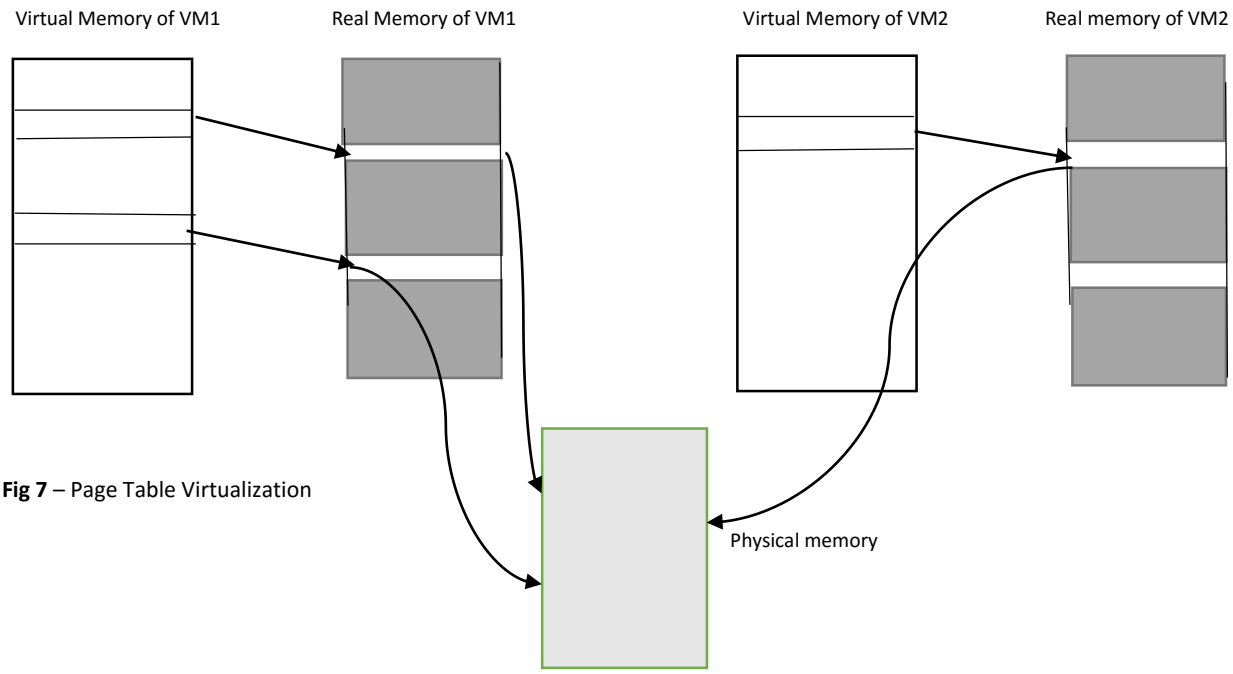


Fig 7 – Page Table Virtualization

Depending on the ISA, either the Page Table is architected or TLB is architected. IF former is the case it means that TLB is invisible to the OS and hardware maintains the TLB. So all VMs share the same TLB too. If the TLB is architected by ISA on other hand then every VM is provided with its own TLB entry too.

I/O – Unlike memory, I/O devices are large in number and continue to grow. Also different I/O devices can be very dissimilar. Virtualization of I/O devices is achieved by creating an abstraction of these I/O devices. They are also called “virtual devices”. A guest OS will make a call to these virtual devices. The VMM will then trap such a request and transfer it to the real device driver in most cases. How each I/O device is virtualized depends on its types. Virtual devices can be broadly divided into 4 broad categories.

- **Dedicated Device -** Dedicated devices like keyboard or monitor cannot be shared among various VMs at once. Also once a guest gets hold of such a device, it can be assumed that it is going to control for some indefinite time. Such devices need to intervention of VMM. Their request can

be directly passed on to the guest OS. But since guest OS is running in non-privileged mode, VMM has to interfere.

- **Partitioned Devices –** Partition devices are like hard disks which can be partially controlled by many VMs. When it comes to partitioned devices, each VM gets a part of the device but it appears that the VM has a complete device. Some sort of mapping is maintained. E.g. in case of hard disks, the disk is partitioned into smaller virtual disks. Then a mapping is maintained that maps the virtual disks to real address space in the physical disk.
- **Shared Devices –** Shared devices are controlled by one VM at a time too but for a definite amount of time. Each VM maintains its virtual state. E.g. if VMs want to get access to the network then they have a virtual network address. The VMM maintains all such virtual network addresses. Any request for network is trapped by VMM and then passed on to physical network. The reply from physical network is again trapped by the VMM and then forwarded to appropriate VM. A different type of shared device is a **Spooled device** like a printer. A printer is always under the control of the program that initiated it till the program signals the release of printer. Even when the program is swapped in and out of memory, the command of the printer should stay with the program (Madnick and Donovan 1974). Such spooled devices are handled by making two level spool table. The first level is maintained by the host OS. The second level is maintained by the VMM. But modern day printers handle spooling internally. In such a case each VM's virtual network address is enough. No multi-level spool table is required.
- **Nonexistent devices –** A virtual device need not always map to a real device. E.g. there can be a virtual networking devices that is only used to network with other VMs in the system. VMware allows clipboard to be shared between the host OS and guest OS. In such cases the drivers for such devices need to be installed in the VMM only.

Virtualization of I/O on a Hosted VM **–** In hosted Virtual Machines, the request from a guest is converted to user level application request by the VMM and passed on to the Native OS. Hence VMM need not have all the device drivers and can simply use the device driver available in the native OS. In hosted VMs, the VM is available in three modes

- **VMM-n (native) -** This runs at the same privilege as the native OS. It may have a small number of device drivers to improve performance.
- **VMM-u (user) –** This runs at non privileged level like a normal application. VMM-u makes request for resources on behalf of native mode VMM.
- **VMM-d (driver) –** This solely exists to facilitate communication between VMM-u and VMM-n.

Processors – Processor virtualization deals with the problem of executing system level and user level guest instructions. Two methods are employed to achieve this. The first is *emulation.* Each of the guest instruction is analyzed. If they are interpreted then the instructions are analyzed repeatedly but if there is a binary translation then the analyzed only once. When the ISA of host and guest system is dissimilar, then emulation is the only way to virtualize the processor. The second method of processor virtualization is *direct native execution* which is possible only when guest and host have similar ISA. But *Popek and Goldberg's* three conditions dictate that not all instructions can be natively executed. So even in case of similar ISAs, the VM has to use part emulation and part direct native execution to implement processor virtualization.

2.5 Performance Degradation in Virtual Machines

It is not possible to run a VM as efficiently as a native system. Performance degradation are often used by malware writers to detect virtualization. Having discussed the implementation of a Virtual Machine, let's analyze the key reason for performance degradation in a VM. [12]

- **Setup –** When a VM is activated, its state is set up. State includes setting up program counter, register, virtual tables etc. VM have their own timing facilities and that needs to be synchronized with the processor too.
- **Emulation –** Emulation adds an extra layer to translation of user instructions and that slows down a VM.
- **Interrupt Handling –** Interrupt generated by the guest OS have to be first trapped by the VMM and then passed on to native OS for handling. This one addition step has its own overhead.
- **State saving –** When VMM takes control, the state of a VM has to be saved first.
- **Time elongation –** As we have seen while discussing implementation of VM that sometimes VMM comes up with extra data structure of its own. So some instruction in VM take longer because they have extra mapping to handle.
- **Bookkeeping –** Some devices have to be managed by VMM and native OS both. For instance networking device in VMs are handled by VMM and OS both. This slows down performance.

2.6 Application of System Virtual Machines

Since earlier days of computing, underutilization of resources has been an issue. Multiprogramming environment with timesharing was created to make optimum use of hardware. VMs took this concept further by creating complete systems that ran on the hardware. VMs have found widespread use in today's computing world. Therefore it has become imperative for many software developers to identify whether they are running in a virtual environment or real. Let's take a look at some of the uses of Virtual machines.[12]

- **Implementing Multiprogramming –** This is from a historical context. Implementing multiple single user VMs that had illusion of being the complete system with complete access to disk and peripherals was one of the trick to implement multiprogramming environment on OS which didn't support timesharing and multiprogramming.
- **Multiple Single application VMs –** This is a common trick applied while testing. When a new application has to be tested it is assigned its own VM. This way any potential bug in the application won't bring down the entire system. Also debugging becomes simpler too.
- **Managed application environment -** A cloud is a good example of managed application environment. The user is provided with a VM with a core set of applications. If he wants to install his own application then he is provided a different VM. Virtual Desktops are a popular concepts these days too. User can login from their browser to a PC system that has popular business tools installed like Office 365. So you don't need to buy an entire PC for your business requirements.

- **Mixed OS environment –** Some OS are more suited for application development. While some OS support a wide variety of productivity tools. So many user would like to install both OS and switch between them effortlessly without any need to logging out of one OS.
- **Support for Legacy Systems –** Sometimes there is need to run old application which have not been optimized for newer version of OS. VMs can be helpful in such cases.
- **Multi-Platform application development –** Software developers often need to develop the same software multiple OS. In such a scenario it is more cost effective to have several VMs running on the same hardware rather than having a dedicated hardware for a single OS.
- **System transition –** When a new OS is released then VMs are an efficient means for system transitioning. All applications can be replicated in the VM running the new OS and once it has been ascertained there are no critical bugs we can discard the old system and move to new OS completely.
- **Operating System Instrumentation –** The VMM can act as an effective tool for instrumenting the OS. Since VM directly interacts with the hardware resource, it can be used to track a particular kind of system resource like program stack and heap or page faults. We can also program the VM to collect all sorts of statistics on such measures. User mode Linux which is used to test fault tolerance of Linux systems is run in a VM environment.
- **Event monitoring –** Shadowing and tracing are two common techniques nowadays in security and bug analysis. VMMs can provide necessary service to aid these techniques. Also VMs provide the feature where they can be replayed from a saved state. So event tracking is easier in a virtual environment.

2.7 Virtual Machine Technology in security

Apart from all these uses, Virtual Machines have become the mainstay of security research. Their ability to provide isolation, save the state and facilitate replay makes them ideal for analyzing threats. We will discuss three ways VMs are used in security research and applications. [12]

Sandboxing

As we have seen earlier that Virtual Machine can run completely independent of one another. One VM crashing will not affect any other VM in the system. Such fault containment make it ideal for *Sandoboxing.* A sample Sandbox architecture is described in the figure below.

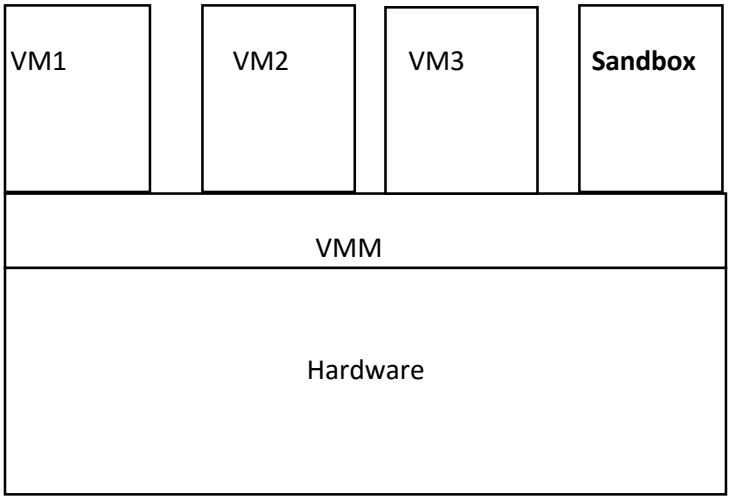


Fig 8 - Sandboxing

(PM Chen and Noble 2001) had stated that attacks can be best understood by replaying them and monitoring the system during the attack. VMs can be saved, cloned, encrypted and restored. Cloning is what makes sandboxing possible. A suspicious packet from network or a suspicious application

from machine can be first sent to a clone machine before being forwarded to intended machine. Similarly while developing an *Intrusion Detection System,* it is safer to test it on a VM than to implement it on a real machine directly.

Monitoring Low Level Activity

We have seen while discussing the architecture of VM that the VMM can form a barrier between the hardware and the OS or other systems installed in the same hardware. This property can be remodeled to make the VMM act as barrier between a system under attack and an *Intrusion Detection System (IDS).* There are two ways to achieve this

- The IDS is separate from VMM. It is either part of the native OS if it's a hosted VM or can be run in a dedicated VM. An interface must be provided between IDS and VMM. This interface must achieve three things – a) Enable message passing between VMM and IDS. IDS can for instance ask VMM to enable instrumentation. b) Enable the IDS to access the physical machine. c) VMM should be able to transfer information about state of a particular VM back to IDS
- The IDS is part of VMM. It runs at the same privilege as the VMM. This was IDS can itself monitor the system being protected. One important criteria for IDS to be part of VMM is that it should be a very secure piece of software just like a VMM is.

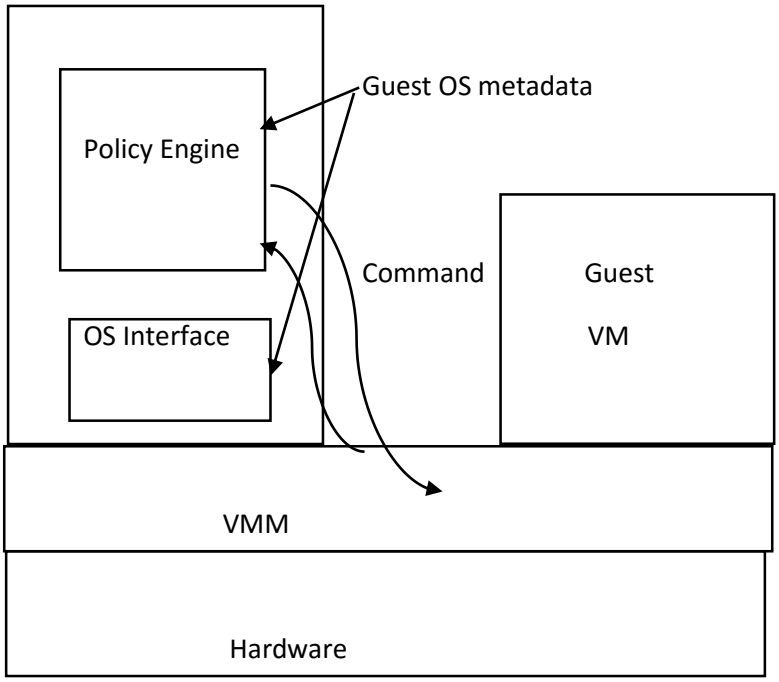


Fig 9. Livewire IDS (Garfunkel and Rosenbaum 2003) [15]

Separating the IDS from the VMM is generally the preferred method. Garfunkel and Rosenbaum, 2003 proposed Liverwire IDS [15] which followed this model. Following is how the Livewire IDS handled various attacks

- **Lie Detector –** Malicious code is often written for a particular OS in mind. VMM has no way to determine the type of guest hosted. Once the malware gets root access, it can start lying and an affected system remains hidden. IDS contains metadata about Guest OS system. So if the VM is lying about the state then it can easily be caught.
- **User Program integrity –** Livewire generates a signature for every code page of a program in memory with the help of VMM. That is than compared against the known "safe" signatures and any discrepancy is flagged as potential attack.
- **Signature detection –** New malwares often use the kernel code of old malwares. Since VMM can see the entire virtual hardware, they can do a complete memory sweep in search of such signature kernel codes.

There are many more such protection modules provided by Livewire IDS.

Secure Logging through Virtual Machines

Logging is an important part of malware analysis. Making a log of access to all critical part of a system can help identify a similar future attack and also identify if the attack has occurred. Unfortunately log data is not reliable once the attack has occurred. But with the help of VMMs this problem has been overcome. There are systems e.g. ReVirt [21] which use the VMM to maintain the integrity of logged data. The specifics of implementation of such a system is beyond scope of this thesis.

Chapter 3

Anti-Virtualization Techniques

In the above discussion of uses of Virtual Machine, we saw that malware analysis is one of the important use of VMs. Apart from that we saw that a host of applications are being installed in VMs. So an application programmer will require the knowledge of his environment before deployment. He might, for instance, forego deployment of certain device drivers if it's a VM the application is being deployed in. So both from programmer and malware analysis point of view, anti-virtualization become important concern. Refer to the table to see how anti-virtualization technique has grown in recent times.

Study Name	**Year**	**Sample Size**	**% of malwares with anti-VM technology**
Chen et al[9]	2008	6222	2.7%
Lindorfer et al[11]	2011	1622	25.6%
Branco et all[10]	2012	4 million	81.4%

Table 1 – Growth of anti-virtualization technology in malwares

Upon detection of a virtual environment malwares can radically change their behavior. They can act benign. They might force the system to crash. They can even escape the VMM. So building countermeasures becomes important. And to do so it is important to understand the anti-virtualization techniques malware authors generally use.

3.1 Taxonomy of anti-Virtualization technology

Anti-virtualization technology can be broadly classified into following 4 categories

- CPU semantic attacks
- String attacks
    - Looking for Virtual Machine Environment (VME) artifacts in process, file system or registry
    - Looking for VME artifacts in memory
    - Looking for VM specific hardware
- Timing analysis
- Miscellaneous attack

Anti-virtualization technology can also be categorized on the basis of abstraction they attack. Following is the categories based on abstraction

- Hardware: artifacts analyzed – device, drivers
- Environment: artifacts analyzed – memory
- Application: artifacts analyzed – installation execution
- Behavioral: artifacts analyzed – timing analysis

We will stick to the first categorization. Let's look in detail each of the categories

3.1.1 CPU semantic attacks

While implementing processor virtualization, not all machine level instructions get emulated correctly. Malware writers target these instruction which behave differently on host machines and the VM. Such instructions are called **Pills.**

A pill is a combination of machine instruction and register/memory value that executes differently in physical machine and in VM.

Some examples of pills are:

- The *CPUID* instruction returns a value of 0 in TSC bit under *Ether* hypervisor. In a physical machine the TSC bit is populated as 1.
- When the value 7fffffffh to register mm1(a floating point register), the st1 register gets populated with signal SNaN in physical machine. In Qemu, it gets populated with random number.

Apart from pills, malware authors can also probe for OS databases like *Global Descriptor Table* (GDT) or *Local Descriptor Table* (LDT). In a hosted VM, the lower memory will be occupied by physical machine data-structures. The GDT and LDT of VMs will be occupying higher address in stack space. One of the first attacks to determine VMs used this fact. Joanna Rutkwoska [16] wrote a very simple code to identify a virtualized environment.

```
int swallow_redpill () {
       unsigned char m[2+4], rpill[] = "\x0f\x01\x0d\x00\x00\x00\x00\xc3";
       *((unsigned*)&rpill[3]) = (unsigned)m;
       ((void(*)())&rpill)();
       return (m[5]>0xd0) ? 1 : 0;
     }
```

The above code returns 0 in a virtual environment. The heart of the code is the *SIDT* instruction that gets you the address of the *Interrupt Descriptor Table*.

3.1.2 String Attacks

When a VMM runs, it keeps making its own entries in memory, processor specific tables etc. Also VMM creates its own virtual devices. String attacks look for such “strings” which can betray the environment.

Let’s take example of the Linux environment.

Checking the dmesg through *cat /var/log/dmesg* can give us lot of information about the environment[19]. Below are some of the sample outputs from going through the dmesg of linux for various virtual machine vendors.

VMWare:

```
# dmesg | grep -i virtual
 VMware vmxnet virtual NIC driver
   Vendor: VMware    Model: Virtual disk      Rev: 1.0
 hda: VMware Virtual IDE CDROM Drive, ATAPI CD/DVD-ROM drive
```

Qemu or KVM:

```
# dmesg | grep -i virtual
 VMware vmxnet virtual NIC driver
   Vendor: VMware    Model: Virtual disk      Rev: 1.0
 hda: VMware Virtual IDE CDROM Drive, ATAPI CD/DVD-ROM drive
```

or

```
dmesg | grep -i virtual
 [    0.000000] Booting paravirtualized kernel on KVM
```

Sometimes VMs can leave BIOS information behind too which can be read using *dmidecode* in Linux environment. BIOS information has many components like vendor name, product information. Some component will betray the virtual environment. [19]

VMWare:

```
# dmidecode | egrep -i 'manufacturer|product'
 Manufacturer: VMware, Inc.
 Product Name: VMware Virtual Platform
```

VirtualPC:

```
# dmidecode | egrep -i 'manufacturer|product'
 Manufacturer: Microsoft Corporation
 Product Name: Virtual Machine
```

Qemu:

```
# dmidecode | egrep -i 'vendor'
 Vendor: QEMU
```

Tracking Virtual hardware can reveal information about the VM too. VMM virtualizes many hardware components like network adapter, USB controller etc. These virtual devices have distinctive “types” or names which can belie the virtual environment. In Linux looking for disk device information can reveal a lot. Below are some of the examples. [19]

VMWare:

```
# cat /proc/ide/hd*/model
VMware Virtual IDE CDROM Drive
# cat /proc/scsi/scsi
Attached devices:
Host: scsi0 Channel: 00 Id: 00 Lun: 00
  Vendor: VMware   Model: Virtual disk     Rev: 1.0
  Type:   Direct-Access                    ANSI SCSI revision: 02
```

VirtualPC:

```
# cat /proc/ide/hd*/model
Virtual HD
Virtual CD
```

Qemu:

```
# cat /proc/ide/hd*/model
QEMU HARDDISK
QEMU DVD-ROM
```

Another popular method to detect a VM environment was to read the MAC address. The list of MAC addresses of popular device is available online. So MAC address can reveal the vendor. If the MAC address of a device belongs to no popular vendor than it probably belongs to a virtual device.

3.1.3 Timing Analysis

We know that sensitive instructions get privileged in VMs. Using this fact we can determine a Virtual Environment. A sensitive instruction will take less time in bare metal system than in a Virtual Machine ideally speaking. But using any external or internal timer becomes difficult. Internal timers like "*rdtsc"* can be skewed. Many VM trap rdtsc instructions and skew it so as to hide the fact that it's running in a virtual environment. They maintain a *TSC_OFFSET*, which they deduct from the real rdtsc value before returning the control to VMM. External timing source like network timing introduce fuzziness and hence it is difficult to come up with reliable statistics to use it for VM detection. Hence we use a counter based timing[14]. In this method we do not use explicit timing. The method is as follows:

- In one thread run a *CPUID* instruction for certain number of times. CPUID is a sensitive instruction but can be called from user space.

- In another thread run *NOP* instruction in loop while maintaining a counter. NOOP is a light weight instruction. When the thread running CPUID terminates, terminate the NOOP thread too while keeping a count of number of times its loop executed.

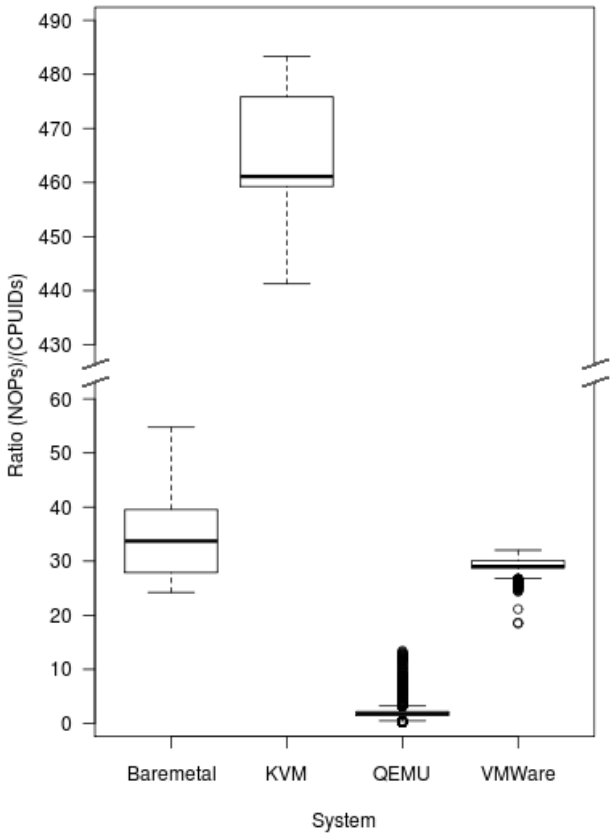


Fig 10 – Timing analysis results – Source – *Christopher Thompson et al. [14]*

The results were interesting. In bare metal system CPUID takes approximately 200 times more clock cycles than NOOP. In Qemu though it takes almost as much time NOOP. This may be because Qemu emulates its own light weight CPUID instruction. While in KVM, CPUID take 450 times more clock cycles when compared to bare metal system. So these statistics can be used to detect a virtual system.

Another trick to implement timing analysis is to use TLB profiling. We can cache some pages to the TLB. Then make a call to a sensitive instruction so that there is a TLB flush. Then we can access the same pages again. There should be noticeable difference in time. VMM uses an extra layer of paging as

discussed previously. So fetching pages from table is even a slower process. So flushing only a few pages should lead to a noticeable time difference. The algorithm for TLB profiling looks something like this.

```
for ( i =0; i<max number ; i++)
{
       p [ i ] = alloc_4k_page ( ) ;
}
Flush TLB ( ) ;
Flush DataCache ( ) ;
for ( i =0; i<max number ; i++) // Fill TLB and find access time
{
       x = p [ i ] [ 0 ] ; t1 = rdtsc ; // 4KB pages in L1 d-TLB
       x = p [ i ] [ 0 ] ; t2 = rdtsc ;
       access_time_1 [ i ] = t2 – t1 ;
}
CPUID ; // Cause #VMEXIT

for ( i =0; i<max number ; i++) // Find access time again
{
       t1 = rdtsc ;
       x = p [ i ] [ 0 ] ; t2 = rdtsc ;
       access_time_2 [ i ] = t2 – t1 ;
}
for ( i =0; i<max number ; i++)
{
       i f ( acces_time_2[i] – access_time_1[i] > threshold )
       {
              printf("VMM exists");
       }
}
```

TLB time profiling

It is also possible to use external timing. But due to fuzziness and latency this method will not always be reliable. A sufficiently difference in timing though can overcome this problem. We can use low resolution timer like NTP that doesn't requires any trap to calculate the timing difference. But to use NTP, the function has to be called a large number of times. Malware authors can afford this though.

There are high precision external timers but hey raise a trap when called. If the VMM catches this trap they can use something like TSC_OFFSET.

```
t1 = GetTimeViaNTP ( ) ;
for ( i =0; i<max counter ; i++)
{
   cupid() ; // Cause #VMEXIT a number of times
}
t2 = GetTimeViaNTP( ) ;

if ( t2 – t1 > threshold )
{
   printf("VMM exists") ;
}
```

Timing analysis through external timing

3.1.4 Miscellaneous attack

There are other varieties of attacks which either take advantage of design flaw of a particular VM or they use properties of OS to detect virtualization. We will take a look at such attacks in this section.

- **TLB Profiling**

  Keith Adams [18] proposed a mechanism of using TLB to detect virtualization that didn't involve any counter based timing analysis. The algorithm he used is shown below.

  Initially the L1 TLB is flushed and completely filled with pages which are all initialized to a certain value say 0x44 in this case. Then a *new_page* is created which is filled with a different value – 0x55

in the pseudocode below. Then page table entries corresponding to these pages is patched to the *new_page.*

At this point a sensitive instruction is executed which either flushes the entire TLB or makes few changes to TLB depending on the architecture of the processor underneath. Then these pages are accessed again and the data value they contain is read. If the value of *new_page* i.e. 0x55 is encountered then it means a page walk took place. So we can conclude that a VMM exists.

```
for ( i =0; i<max number ; i++) // max number = number of entries for
{                                  // 4KB pages in L1 d-TLB
       page[i] = alloc_4k_page( ) ; // Allocate set of pages
}

Page_new = alloc_4k_page( ) ;// Allocate new page

for ( i =0; i<max number ; i++)
{
       memset(page[i],0x44 ,PAGE SIZE) ; // Fill all the
//pages with a data value
}

memset (page new ,0x55,PAGE SIZE );//Fill new page with another  value

for ( i =0; i<max number ; i++)
{
       x = page[i][0] ; // Fill up the TLB completely
}

for(i =0; i<max number; i++)//Remap PTEs to physical address of page
new
{
       Patch_PTE( page[i] , physical_addr(page_new)) ;
}
cpuid() ; // Cause #VMEXIT

for ( i =0; i<max number ; i++)
{
       if(page[i][0]== x55 )
       break ; // No mapping in TLB, so page walk done
}

if( i < max_number )
printf( ”VMM exists \n” ) ;
```

TLB profiling

-

-

- **“Magic value” of VMware**

VMware establishes a “ComChannel” or communication channel between guest and host. This gives certain useful features like allowing sharing of clipboard, drag and drop features, file sharing, time synchronization etc. To enable this, VMware’s VMM tweaks certain machine instructions.

```
mov EAX, 564D5868h ; VMXh
xor EBX, EBX  ; set EBX to anything but 0x564D5868 (in this case 0)
mov CX, 0Ah   ; Backdoor command. 10: Get VMware version
mov DX, 5658h  ; VX
in EAX, DX  ; Read from port VX into EAX
cmp EBX, 564D5868h ; EBX should have the magic number VX is VMware is
                     present. If not, EBX=0
```

VMware magic port detection – Tom Liston, Ed Skoudis [7]

First the magic number “VMxh” is copied to eax and ebx can be set to anything apart from “Vmxh”. Then we load a backdoor command to CX and DX gets the I/O port number which is “VX”. Then an “in” instruction is read from port “VX” into eax. “in” is a privileged instruction and should cause an error in host machine but in VMWare it will return the magic number “VMxh”. This value is read into ebx. In case of an error, ebx reads 0.

Other related works

Dynamic malware analysis sandboxes (DMAS) such as Anubis [27] and CWSandbox [28] have been developed which provide means to automatically execute a sample in a controlled environment and monitor its behavior. Based on the output of a DMAS, malware analysts can figure out the nature of the application under study. [29] Has shown that malware attacks today have a strong economic motivation. So they are only going to get worse in future. Blue Pill [43] had made the first attempt to make a hypervisor completely transparent but [31, 32] posited

that building a completely transparent sandbox is fundamentally unfeasible. Even if one were possible then installation configuration of such a sandbox will belie its existence. [33, 34, 35, 36] propose ways to detect malwares from deviation in behavior. [38] Explores multiple execution paths in Anubis to provide information about triggers for malicious actions.

Chapter 4

Behavior Comparison Results

Before we discuss results we will discuss the experiment set up and what is a positive case and what is a negative case.

Experiment Environment: We tested all the scripts in latest versions of Qemu, Xen, VmWare and VirtualBox.

Qemu - QEMU version 2.2.1

VmWare – VmWare Player 7.0

VirtualBox - VirtualBox 4.3.26

Xen - xen-hypervisor-4.1-amd64 package from Ubuntu 14.04

The host operating system was Ubuntu 12.04 and Ubuntu 14.04.

Positive result – If the script detects virtual environment in spite of changing virtual machine environment parameters, rootkits or Joanna Rutwoska's Subvirt: Blue Pill then it's a positive result.

Negative Result – If the script fails to detect the virtual environment due to changing virtual machine environment parameters, rootkits or Joanna Rutwoska's Subvirt: Blue Pill then it is a negative result.

Rootkit – From rootkit [22, 23, 24] here we generally mean Loadable Kernel Modules (LKMs) mostly and sometimes a simple script running with root privileges. Below we state result of various categories of attack.

	String Attack			
	virt-what[17]	imvirt[18]	dmesg [9]	dmidecode[9]
VMWare	X	X	X	X
VirtualBox	X	X	X	X
Qemu	X	X	X	X
Xen	X	X		

Table 4.1

String Attacks – In above table a cross represents a successful detection. We have covered only the prominent String attacks. *Im-virt and virt-what* are two scripts with a host of string attacks in them. But all these attacks took place with root access. Without root access these attacks didn't run.

	CPU Semantics Attack			
	CPUID[11]	RedPill[7]	SnoopY[23]	Scooby[24]
VMWare	X	X	X	X
VirtualBox	X	X	X	X
Qemu				
Xen	X			

Table 4.2

CPU Semantics attack – We didn't need any special rootkit to hide CPU semantic attack and ran it on bare virtual machines. We chose two old scripts SnoopY and ScoobY for it apart from classical redpill attack and getting the CPUID information.

	Cache Profiling	
	TLB Profiling[21]	Return Address Cache Profiling[30]
VMWare	X	X
VirtualBox	X	X
Qemu	X	X
Xen	X	X

Table 4.3

Cache Profiling – There was no way to stop cache profiling at all. They detected all sorts of VM environments.

	Timing Analysis	
	Graphics performance[29]	Thread Race[14]
VMWare		X
VirtualBox		X
Qemu		X
Xen		X

Table 4.4

I could not come up with reliable parameters to use graphic performance to detect virtual environment. Thread Race on other hand was highly effective.

The below table summarizes all the above attacks.

Attack type	**Type of VMM affected**	**Countermeasure**	**Comments**
CPU semantic attacks	Type1, Type 2, Emulated	Easy	False positive cases high
String Attack	Type 1, Type 2 and Emulated	Easy for Type 1, medium for type 2 and emulated	Many such attacks require root access
Timing analysis	Type 1, Type 2 and Emulated	Difficult.	Possible to write rootkit but rootkit will end up harming performance a lot.
Miscellaneous	Type 1, Type 2 and Emulated	-	When it comes to TLB profiling, countermeasures are very difficult.

Table 4.5

Xu Chen et al summarized the results from second categorization of anti-virtualization technology. The below table is from their paper for compare and contrast.

Abstraction	Artifact	Accuracy	Access Level	Complexity	Evasion	Imitation
Hardware	device	high	local	medium	easy	easy
	driver	high	local	low	medium	easy
Environment	memory	high	local	medium	hard	medium
	system	high	local	high	hard	hard
Application	installation	high	local	low	easy	easy
	execution	high	local	low	medium	easy
Behavioral	timing	medium	network	medium	medium	medium

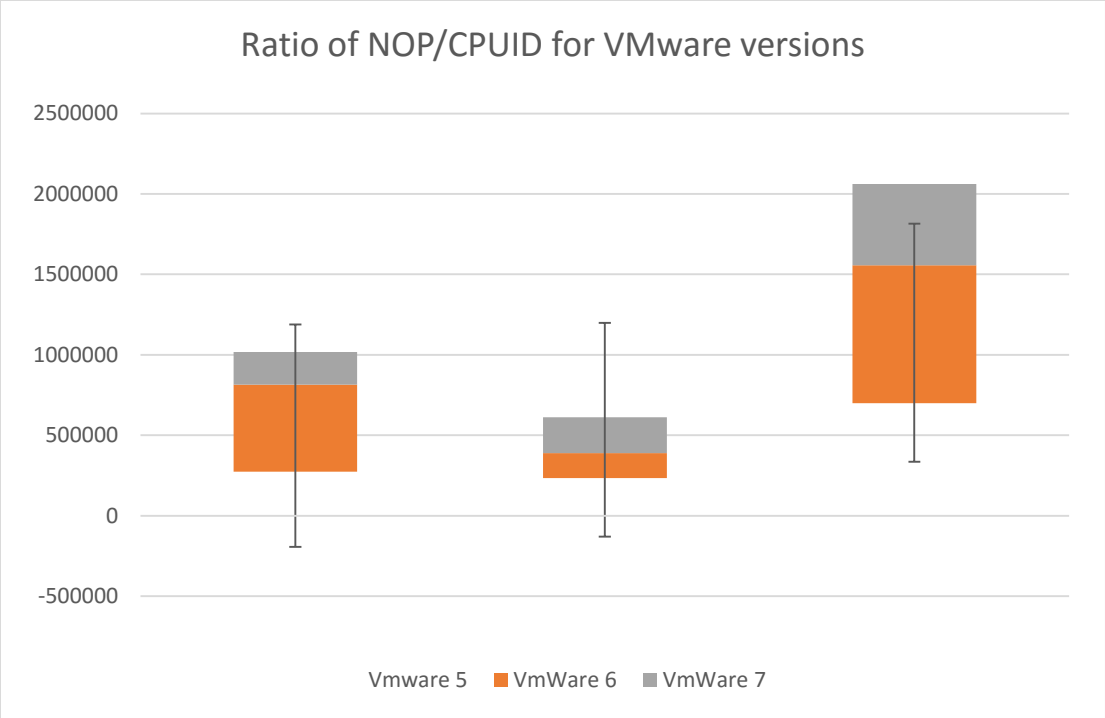
Table 4.6

Here Type 1 VMM stands for the classic VMM and Type 2 VMM stands for the hosted VMM.

We see a digression from *Xu Chen* et al, we have shown that timing attacks is possible locally too. We don't necessarily need network timing to carry it out. A counter based timing attack neither depends on local timing or external timing.

4.1 Detection of Vendor

Detection of Vendor is difficult from Timing attacks and mostly impossible from TLB profiling. We carried out an experiment similar to that by *Christopher Thompson et al[14]* for three different generation of **Vmware Player** and the timing analysis gave vastly varied result as evident from the graph below. So ratio is not a good measure to detect the Vendor.



So to sum it all up, we have the following.

My script – "detect"

Technique 1 – *Timing Analysis* – Run two threads. One with *NOP* instructions. Other with a sensitive instruction like *CPUID.* CPUID instruction will not lead to a context switch in a host machine but on a virtual machine it will. Context switches are more expensive and hence the relative timing difference between execution of *NOP* and *CPUID* will help detect a virtual machine.

Strengths –

I showed it reliably detects a VM environment.

It is easy to implement. Also requires no root access.

I also showed that countermeasures are very difficult.

Weakness –

Doesn't determine the vendor of VM accurately.

Technique 2 – *TLB profiling* - Executing a sensitive instruction also leads to TLB flush. Profiling the pages present in TLB before and after a sensitive instruction can also reveal a virtual environment.

Strengths-

Accurately detects virtual environment. Needs no root access

I showed that the countermeasures are difficult too.

Weakness-

Difficult to implement. Since *cache* management differs processor to processor. Also it is not at all useful to detect the vendor.

virt-what

Technique 1 – ***String Attack*** - Uses "dmidecode" and "dmesg" and searches for string within them. Also looks for strings in /proc/cpuinfo etc.

Strength –

Easy to implement.

Reliably detects VM.

Detect vendors accurately.

Weakness –

Countermeasures are easy. I was able to write a rootkit for VMWare which hid files in /proc/cpuinfo. Dynamic Binary translation can trap dmidecode and dmesg hide all references to virtualization. Needs root access.

Technique 2 – *String attack on CPUID –* Values returned by CPUID can have VM environment information sometimes.

Strength-

Easy to implement and reliably detects a VM.

Weakness-

Countermeasure is easy since Dynamic Binary Translation can easily mimic the CPUID of host machine.

Imvirt

Imvirt uses “dmidecode” like virt-what above. But it also has Vendor specific attacks when not having root access. All those attacks come under ***CPU Semantic attacks*** and I showed that when compared to timing analysis and TLB profiling, the countermeasures

for CPU semantic attacks are easier. So the below table sums up the comparison between three scripts.

Script Name	**Root access? Y/N**	**VM detected reliably? Y/N**	**Vendor detected reliably? Y/N**	**Countermeasures**
detect	N	Y	Y for only certain vendors	Difficult
virt-what	Y	Y	Y	Easy
imvirt	N	Y	Y	Yes

Chapter 5

Conclusion

5.1 Main Contributions

In this thesis, we have attempted two things. First we categorized the new age anti-virtualization technologies. We saw how they have improved over the years, getting more sophisticated. This is important for two broad reasons. First reason being that virtualization has become backbone of security analysis. We saw how a host of security features from firewall to Intrusion detection is implemented using virtual machines. If the virtualization is detected then the malware analysis can be compromised.

The second reason was that thanks to new and improved hardware specifications virtual machines are finding increased use. We can now have two very different OS running side by side. This makes it necessary to detect the virtual environment. While installing applications or patches it becomes necessary to know the type of environment we are in. Keeping that in mind we wrote a script to detect virtualization.

5.2 Future Works

Automated testing of Virtual Machine Monitor (VMM) to detect all types of bugs that can be exploited is an area that warrants attention. VMM are the heart of virtualization and securing them should be our foremost aim. Also we saw that TLB profiling and thread race conditions are difficult to block without paying severe performance penalties. So efforts have to be made towards observing behavior of malwares in a variety of environments instead of going for complete transparency. Based on behavior of malwares that implement aggressive timing attacks we can come up with a pattern. This pattern hen can be fed to *Anubis[23]* and the instrumented code can then monitor the behavior of any malware

running in the system. Ay malware trying to detect the virtual environment looks for specific areas in memory which a general purpose application doesn't. So any attempt to access such sources of information can be used to flag malware activity too. [45, 46, 47] shed light on how anti-virus programs monitor changes made to Kernel Data Structures by rootkits. This research is useful in monitoring in malwares exploiting the hypervisor too.

Bibliography

[1] JOHN J DONOVAN, Use of Virtual Machines in Information Systems

http://dspace.mit.edu/bitstream/handle/1721.1/60458/EL_TR_1975_010.pdf

[2]MICHAEL ARMBRUST,ARMANDO FOX,REAN GRIFFITH,ANTHONY D. JOSEPH,RANDY H. KATZ,ANDREW KONWINSKI,GUNHO LEE,DAVID A. PATTERSON,ARIEL RABKIN,ION STOICA,MATEI ZAHARIA,Above the Clouds: A Berkeley View of Cloud Computing

https://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf

[3] KING, S. T.; CHEN, P. M. (2006). "SubVirt: implementing malware with virtual machines". 2006 IEEE Symposium on Security and Privacy (S&P'06). pp. 14 pp.

[4] http://www.informit.com/articles/article.aspx?p=2233978

[5] http://social.technet.microsoft.com/wiki/contents/articles/942.hyper-v-how-to-detect-if-a-computer-is-a-vm-using-script.aspx

[6] http://www.coffer.com/mac_find/

[7] TOM LISTON, ED SKOUDIS, On the Cutting Edge: Thwarting Virtual Machine Detection

http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf

[8] HAO SHI, ABDULLA ALWABEL, JELENA MIRKOVIC, Cardinal Pill Testing of system virtual machine

[9] CHEN, X., ANDERSEN, J., MAO, Z., ET AL. Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware. In IEEE International Conference on Dependable Systems and Networks with FTCS and DCC (DSN) (2008)

[10] BRANCO, R. R., BARBOSA, G. N., AND NETO, P. D. Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies. In Black Hat (2012).

[11] LINDORFER, M., KOLBITSCH, C., AND MILANI COMPARETTI, P. Detecting Environment-Sensitive Malware. In Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID) (2011).

[12] JIM SMITH, RAVI NAIR, Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design) – June 3, 2005

[13] FERRIE , P. 2007. Attacks on more virtual machine emulators. Symantec Technology Exchange

[14] CHRISTOPHER THOMPSON, MARIA HUNTLEY, CHAD LINK, Virtualization Detection: New Strategies and Their Effectiveness

http://www.cs.berkeley.edu/~cthompson/papers/virt-detect.pdf

[15] T. GARFINKEL AND M. ROSENBLUM, "A virtual machine introspection-based architecture for intrusion detection," in Proceedings of the Network and Distributed Systems Security Symposium, 2003, pp. 1 9 1-206.

[16] J. RUTKOWSKA, "Redpill," http: //invisiblethings.org/papers/ redpill.htrnl.

[17] XU CHEN, JON ANDERSEN, Z. MORLEY MAO, MICHAEL BAILEY, JOSE NAZARIO, Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware, in Proceedings of International Conference on Dependable Systems &Networks: Anchorage, Alaska, June 24-27 2008

[18] http://x86vmm.blogspot.com/2007/07/bluepill-detection-in-two-easy-steps.html

[19] http://www.dmo.ca/blog/detecting-virtualization-on-linux/

[20] GERALD J. POPEK, ROBERT P. GOLDBERG, Formal Requirement for Virtualizable Third Generation Architectures

[21] GEORGE W. DUNLAP, SAMUEL T. KING, SUKRU CINAR, MURTAZA A. BASRAI, PETER M. CHEN, ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay, Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)

[22] Chkrootkit - rootkit detection tool. http://www.chkrootkit.org/.

[23] Rootkit hunter - rootkit detection tool. http://www.rootkit.nl/.

[24] System virginity verifier. http://www.antirootkit.com/software/System-Virginity-Verifier.htm.

[25] F-secure blacklight. http://www.f-secure.com/blacklight/.

[26] Klister rootkit detector. http://www.rootkit.com/project.php?id=14.

[27] ULRICH BAYER, CHRISTOPHER KRUEGEL, AND ENGIN KIRDA. TTAnalyze: A Tool for Analyzing Malware. In [28] Proceedings of the 15th European Institute for Computer Antivirus Research (EICAR) Annual Conference, 2006.

[29] JIANWEI ZHUGE, THORSTEN HOLZ, CHENGYU SONG, JINPENG GUO, XINHUI HAN, AND WEI ZOU,STUDYING Malicious Websites and the Underground Economy on the Chinese Web. In Proceedings of the 7th Workshop on the Economics of Information Security (WEIS), 2008

[30] TAL GARFINKEL, KEITH ADAMS, ANDREW WARFIELD, AND JASON FRANKLIN. Compatibility is Not Transparency: VMM Detection Myths and Realities. In Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS), 2007.

[31] ULRICH BAYER, IMAM HABIBI, DAVIDE BALZAROTTI, ENGIN KIRDA, AND CHRISTOPHER KRUEGEL. A View on Current Malware Behaviors. In Proceedings of the 2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET), 2009.

[32] DAVIDE BALZAROTTI, MARCO COVA, CHRISTOPH KARLBERGER, CHRISTOPHER KRUEGEL, ENGIN KIRDA, AND GIOVANNI VIGNA. Efficient Detection of Split Personalities in Malware. In Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS), 2010.

[33] XU CHEN, JON ANDERSEN, ZHUOQING MORLEY MAO, MICHAEL BAILEY, AND JOSE NAZARIO. Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware. In Proceedings of the 38th Annual IEEE International Conference on Dependable Systems and Networks (DSN), 2008.

[34] NOAH M. JOHNSON, JUAN CABALLERO, KEVIN ZHIJIE CHEN, STEPHEN MCCAMANT, PONGSIN POOSANKAM, DANIEL REYNAUD, and DAWN SONG. Differential Slicing: Identifying Causal Execution Differences for Security Applications. In Proceedings of the 32nd IEEE Symposium on Security and Privacy, 2011

[35] MIN GYUNG KANG, HENG YIN, STEVE HANNA, STEVE MCCAMANT, AND DAWN SONG. Emulating emulation-resistant malware. In Proceedings of the 2nd Workshop on Virtual Machine Security (VMSec), 2009.

[36] ANDREAS MOSER, CHRISTOPHER KRUEGEL, AND ENGIN KIRDA. Exploring Multiple Execution Paths for Malware Analysis. In Proceedings of the 28th IEEE Symposium on Security and Privacy, 2007.

[37] PETER FERRIE. Attacks on Virtual Machine Emulators. Technical report, Symantec Research White Paper, 2006.

[38] ROBERTO PALEARI, LORENZO MARTIGNONI, GIAMPAOLO FRESI ROGLIA, AND DANILO BRUSCHI. A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT), 2009.

[39] AMIT VASUDEVAN AND RAMESH YERRABALLI. Cobra: Fine-grained Malware Analysis using Stealth Localized-executions. In Proceedings of the 27th IEEE Symposium on Security and Privacy, 2006.

[40] GÁBOR PÉK, BOLDIZSÁR BENCSÁTH, AND LEVENTE BUTTYÁN. nEther: In-guest Detection of Outof-the-guest Malware Analyzers. In Proceedings of the 4th ACM European Workshop on System Security (EUROSEC), 2011.

[41] PHILIPP TRINIUS, CARSTEN WILLEMS, THORSTEN HOLZ, AND KONRAD RIECK. A Malware Instruction Set for Behavior-Based Analysis. Technical Report 07–2009, University of Mannheim, 2009.

[42] DAVIDE BALZAROTTI, MARCO COVA, CHRISTOPH KARLBERGER, CHRISTOPHER KRUEGEL, ENGIN KIRDA, AND GIOVANNI VIGNA. Efficient Detection of Split Personalities in Malware. In Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS), 2010.

[42] ULRICH BAYER, ENGIN KIRDA, AND CHRISTOPHER KRUEGEL. Improving the Efficiency of Dynamic Malware Analysis. In Proceedings of the 25th ACM Symposium on Applied Computing (SAC), 2010.

[43] Joanna Rutkwoska, The Blue Pill, http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html

[44] C. CURTSINGER, B. LIVSHITS, B. ZORN, AND C. SEIFERT. Zozzle: Low-overhead mostly static JavaScript malware detection. In Proceedings of the Usenix Security Symposium, Aug. 2011.

[45] C. KOLBITSCH, B. LIVSHITS, B. ZORN, AND C. SEIFERT. Rozzle: Decloaking Internet malware. Technical report, Microsoft Research, Sept. 2011.

[46] NICK L. PETRONI JR., TIMOTHY FRASER, JESUS MOLINA, AND WILLIAM A. ARBAUGH Copilot - a coprocessor-based kernel runtime integrity monitor. In Security ’04: Proceedings of the USENIX Security Symposium, San Diego, CA, August 2004.

[47] BRYAN D. PAYNE, MARTIM CARBONE, MONIRUL I. SHARIF, AND WENKE LEE. LARES: An architecture for secure active monitoring using virtualization. In SP ’08: Proceedings of the 2008 IEEE Symposium on Security and Privacy, Oakland, CA, May 2008.

[48] JR. NICK L. PETRONI AND MICHAEL HICKS Automated detection of persistent kernel control-flow attacks. In CCS ’07: Proceedings of the 14th ACM Conference on Computer and Communications Security, Alexandria, VA, October 2007.