Virtual Machine Detection Through Central Processing Unit (CPU) Detail Anomalies

by

David Mettrick

A thesis submitted to the
School of Graduate and Postdoctoral Studies in partial
fulfillment of the requirements for the degree of

Master of Science in Computer Science

Faculty of Business and IT

University of Ontario Institute of Technology (Ontario Tech University)

Oshawa, Ontario, Canada

December 2022

THESIS EXAMINATION INFORMATION

Submitted by: **David Mettrick**

Master of Science in **Computer Science**

Thesis title: Virtual Machine Detection Through Central Processing Unit (CPU) Detail Anomalies

An oral defense of this thesis took place on November 23, 2022 in front of the following examining committee:

Examining Committee:

Chair of Examining Committee	Miguel Vargas Martin
Research Supervisor	Patrick Hung
Examining Committee Member	Amirali Salehi-Abari
Thesis Examiner	Khalil El-Khatib

The above committee determined that the thesis is acceptable in form and content and that a satisfactory knowledge of the field covered by the thesis was demonstrated by the candidate during an oral examination. A signed copy of the Certificate of Approval is available from the School of Graduate and Postdoctoral Studies.

ABSTRACT

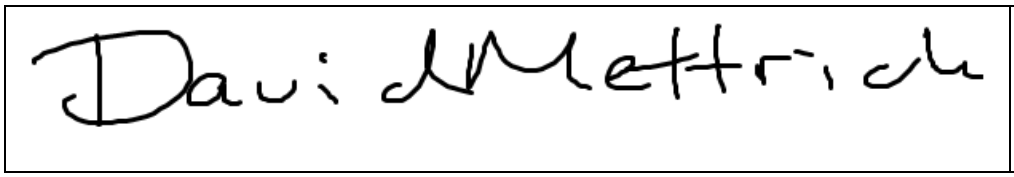
Malware analysts commonly use virtual machines to provide safe environments to study malware. Malware authors in response, include virtual machine detection functions in their malware so it changes its behavior should a virtual machine be detected. It is therefore important for researchers to continuously uncover new virtual machine detection methods that may be exploited by criminals. This thesis explores a method of virtual machine detection that looks for inconsistencies in the following Central Processing Unit (CPU) details: the CPU model, the number of physical cores, the number of logical cores and the cache capacities. Should inconsistencies be detected, a virtual machine is present. We explore our method in scenarios where all CPU cores are assigned to the test virtual machines to determine if inconsistencies exist. In our tests, many of the hypervisors tested possessed inconsistencies that could be used to deduce the presence of a virtual machine.

Keywords: Virtualization; Hypervisor; Security; Malware; Virtual Machine

AUTHOR'S DECLARATION

I hereby declare that this thesis consists of original work which I have authored. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize the University of Ontario Institute of Technology (Ontario Tech University) to lend this thesis to other institutions or individuals for the purpose of scholarly research. I further authorize the University of Ontario Institute of Technology (Ontario Tech University) to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research. I understand that my thesis will be made electronically available to the public.



STATEMENT OF CONTRIBUTIONS

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication. I have used standard referencing practices to acknowledge ideas, research techniques, or other materials that belong to others. Furthermore, I hereby certify that I am the sole source of the creative works and/or inventive knowledge described in this thesis.

ACKNOWLEDGEMENTS

I would like to thank my professor Patrick Hung for supporting me throughout my experiences in graduate school. I dedicate this paper to my sister, Laura Mettrick. You are enough.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS AND SYMBOLS

CPU	Central Processing Unit
VM	Virtual Machine
NIC	Network Interface Card
MMU	Memory Management Unit
TLB	Translation Lookaside Buffer
RAM	Random Access Memory
API	Application Programming Interface
IP	Internet Protocol
ICMP	Internet Control Message Protocol
MAC	Media Access Control
RDP	Remote Desktop Protocol
VNC	Virtual Network Computing
RAAS	Ransomware As A Service

Chapter 1. Introduction

Virtualization has become an important aspect of the information technology world today. Cost-effective, scalable, and with the ability to be moved to other machines, virtualization has been widely deployed in both cloud and desktop environments alike. Virtual machine usage is especially popular in the malware analysis field, as they provide a safe environment where malicious binaries can be run and studied [1]. In addition, virtual machines possess “snapshot technology” that allows for users to save the current state of a virtual machine and revert back to it later, allowing a malware analyst to infect a virtual machine with malware and revert back to a clean state at a later time. This saves the analyst the trouble of having to reinstall the operating system and other necessary software. Unfortunately, many malware authors have noticed this and started to design their malware to detect virtualized environments. If detected, the malware will typically not execute its primary payload. It will instead execute some benign-looking commands before self-terminating so as to not raise suspicions from security systems present [2]. Such behavior from Virtual Machine (VM) sensitive malware complicates analysis for malware analysts, providing an incentive to develop “hardened” virtual machines that are more difficult to detect by malware and, therefore will allow malware to execute their malicious payloads. Malware analysts and malware authors ultimately find themselves in an arms race – one side trying to harden their analysis environments from VM detection, while the other seeks new methods of VM detection to stay hidden longer [3].

Most methods revolving around the topic of VM detection depend on searching for files and system artifacts that would be present inside a virtualized environment but not in a bare metal environment. With this in mind, researchers have developed solutions to hide these known artifacts to trick some VM-sensitive malware programs into executing their main payload, allowing for further analysis. A notable example is “Ether” [4] developed by Dinaburg et al., which contains various solutions to known methods of detecting hypervisor presences such as VMware Workstation and VirtualBox. Another less popular means of detecting VM presence

is using specific Central Processing Unit (CPU) instructions to deduce the presence of VMs. Most of the instruction-based detection methods are no longer functional today. However, this method was used during the mid to late 2000s when the research was being more heavily conducted in this field. A piece of research would be the “Red pill” [5] VM detection method authored by Joanna Rutkowska.

From the malware analyst’s point of view, it is vital to consistently find new ways that malware authors could abuse to detect a virtual machine’s presence. Should a piece of malware utilize a new method of detection that is not known by existing analysis environments, the malware could go undetected for longer, giving the author more opportunity to cause damage or commit various other cybercrimes. Furthermore, hypervisor software vulnerabilities have been found in the past [6], and security researchers have demonstrated proof of concept exploit software to execute arbitrary code on the host machine, effectively allowing an attacker to “escape” the virtual machine. The existence of these dangerous vulnerabilities furthers the importance of doing active research to find new methods criminals could use to detect virtual environments.

In this paper, we aim to address the following questions:

Is it possible for an unprivileged adversary to detect a VM’s presence by looking for inconsistencies in the CPU details exposed? If so, which hypervisors are affected?

This thesis aims to document a new method of detecting virtual machine presence by seeking out inconsistencies in CPU details exposed to the VM. The main premise of our detection method involves checking if the correct number of physical cores, logical cores and caching capacities exist within a system according to the CPU reportedly in use. We assume a VM is present if incorrect values are found relative to the CPU model in use. Assigning a VM any number of CPU cores less than the maximum number of cores available in the CPU will consequently give a VM an incorrect number of physical or logical cores, making our method work independently of the hypervisor in use. We also address the

possibility of users assigning all physical and logical cores to a VM as a means of attempting to defeat our proposed method. We explored 5 virtualization products: VMware Workstation, VirtualBox, HyperV, XEN and KVM/QEMU on 10 different CPUs and assigned all physical and logical cores of the CPU to the testing VMs. Despite our efforts, inconsistencies were found in many of the test cases. We highlight the patterns we observed for each hypervisor along with what CPUs may be more optimal to evade our proposed method.

In summary, our contributions are as follows:

- We propose a means of VM detection by checking for inconsistencies in the number of CPU physical cores, logical cores and cache capacities.
- We document CPU inconsistencies observed in VMs assigned all CPU physical and logical cores among 5 different virtualization products. Our results show that our method is effective at detecting VM presence and is a cause for concern. It should be addressed by malware analysts and those who use “hardened” VMs regularly.

We will structure the rest of the paper as follows. We will cover the technical analysis of the method, along with requirements, limitations and countermeasures researchers can use to detect binaries using this technique. Chapter 2 will cover the background information regarding virtual machines and their functions and usage. Chapter 3 will cover the related works, including in-depth documentation of the known methods of VM detection. Chapter 4 will provide further detail about our proposed method and how we implemented it. Chapter 5 will cover our testing methodology, scope and experiments we did on different CPUs and virtualization platforms. Chapter 6 will provide an analysis of the results from the experiments along with known limitations to our propositions. Finally, Chapter 7 will wrap up the thesis with conclusions and future works.

Chapter 2. Background

2.1 Introduction

Virtualization is the technology that allows for the splitting of resources of a single system to create individual environments referred to as “virtual machines” (VMs). A virtual machine could be thought of as a computer that is entirely run on software and does not rely on physical hardware, unlike a traditional computer. The concept of virtualization and virtual machines was first pioneered in the early 1960s when IBM was faced with the challenge of not fully utilizing the resources available in its expensive mainframe systems [7]. The result of this problem gave birth to IBM’s Control Program (CP), which partitioned the resources of the mainframe to create virtual machines, each running a copy of IBM’s CP/CMS operating system [7]. This revealed a large breakthrough in computer technology, as businesses could share the resources of a single machine instead of requiring several hardware machines for different tasks, thus making the best use of their investments in computer technology. Furthering this discovery, Goldberg and Popek published the paper “Formal Requirements for Virtualizable Third Generation Architectures” [8] in 1974, outlining conditions regarding computer architecture required to practically support virtualization. The topic of virtualization, however became obscured during the 1980s and early 1990s [7] and didn’t become more mainstream with businesses until 1999 when the company VMware developed the software product VMware Workstation, which was capable of virtualizing the x86 CPU architecture [9]. In 2005, Intel and AMD each released CPUs with hardware-accelerated virtualization capabilities to assist with the speed and efficiency of virtualization products such as XEN [10]. Today, virtualization is widely utilized by data centers, cloud computing, malware analysis, academic communities, and individuals. Furthermore, many cloud computing companies exist today, such as Amazon Web Services (AWS), Microsoft Azure, Google CoLab, and DigitalOcean.

2.2 Malware Landscape

Before we further detail virtualization, we believe it is important to further explain the finer details of the current malware landscape. This section intends to explain

what malware is, the different types of malware that exist, the different techniques used by malware to spread, and the current trends with security evasion being used by malware today.

2.2.1 What is Malware?

Malware, the concatenation of the words “malicious” and “software,” is a computer program designed to cause disruption to a computer system or network, usually without a user’s consent or knowledge [11]. Malware is one of the key vehicles used by cybercriminals today to facilitate online crimes and poses a grave threat to businesses and individuals alike. According to AVTest, a respected malware statistics website, over 450,000 new malware programs are released daily, totaling to more than 1 billion malware programs recorded on the internet to date [12]. Further, cybercrime has been estimated to have cost $6 trillion USD globally in 2021 and is forecasted to cost the world $10.5 trillion USD by 2025 [13].

2.2.2 Types of Malware

There exist various types of malware, such as computer viruses, worms, Trojan horses, ransomware, spyware and adware - each designed with a particular goal in mind.

2.2.2.1 Viruses

Computer viruses exist with the intention of replicating and maliciously destroying data on executing systems. Similar to biological viruses, computer viruses require a host machine to run on to replicate and destroy. While not as popular anymore, the creation and spreading of computer viruses were much more commonplace back in the 1990s to early 2000s, and a notable virus was the “CIH” virus, infamous for its ability to overwrite a computer’s BIOS with garbage code rendering the system unbootable [14].

2.2.2.2 Worms

Worms are similar to computer viruses, except they are designed to spread to other computer systems to maximize the damage they cause. Commonly using a network to propagate, worm authors make use of a variety of means to spread their worm creations, such as port scanning and attempting to log in to servers

using weak passwords. A notable example of a worm program was the “Mydoom” worm which spread via Peer to Peer (P2P) networks such as Kazaa and caused system slowing and denial of service attacks on systems worldwide [15].

2.2.2.3 Trojan Horses

Trojan horses, often compared to the ancient Greek Trojan horse that led to the destruction of the city of Troy, refers to any malicious program that is typically packaged with another application to misleading the user of its intention, such as a benign-looking email. Trojan horses rely on social engineering tactics to convince the user(s) to run or install the Trojan horse thinking it is benign when it is actually malicious. A notable Trojan horse is the “Zeus” malware which was believed to have spread via phishing emails, scam campaigns and drive-by downloads [16]. “Zeus” became such a nuisance that law enforcement agencies from various affected countries around the world collaborated in a combined effort to stop the “Zeus” malware code-named “Operation Tovar” [16].

2.2.2.4 Ransomware

Ransomware is a type of malware that intends to abuse cryptography to maliciously encrypt a user’s files and demand a ransom to be paid before decrypting the files. With the advent of anonymous cryptocurrencies such as Monero [17], ransomware has seen an increase of over 151% since 2020, making it one of the biggest malware threats of 2021, according to SonicWall, a cybersecurity company [18]. Such malware has become so popular with cyber criminals that it has led to the rise of ransomware-as-a-service businesses (RAAS), which helps cyber criminals with the creation of ransomware binaries as well as the backend maintenance required for successful ransomware campaigns [19]. With the average ransom demanded in 2021 being $5.3 million USD, ransomware has proven to be an extremely profitable means of monetizing computer hacking and other cybercrimes by criminals. Furthermore, criminals have made use of recently released software exploits as a means of spreading ransomware in hopes of a more successful campaign. A notable example of this is the “WannaCry” ransomware, which made use of the leaked NSA “EternalBlue” exploit [20] which

allowed an attacker unfettered remote access to Windows XP and 7 machines in the spring of 2017 [21].

2.2.2.5 Spyware

Spyware, the concatenation of the words “spy” and “software,” refers to any software program that intends to covertly collect information about a user or network for an attacker to harvest. Such software intends to violate the privacy of the affected party. However, not all spyware software intends to be rogue in nature. “Benign” spyware software exists, such as in the form of parental control software that intends to monitor a child’s use of a computer system. An example of such software is “Net Nanny,” which intends to provide parents with a means of tracking children’s Internet usage and protecting against inappropriate websites [22].

2.2.2.6 Adware

Adware, the concatenation of the words “advertisement” and “software, is a piece of software that intends to generate advertising revenue by maliciously displaying ads to victims. Adware typically operates in a few methods to generate revenue for the authors [23]:

1. Pay-Per-Click
    - Authors receive payment every time a link is clicked.
2. Pay-Per-View
    - Authors receive payment every time an ad is shown.
3. Pay-Per-Install
    - Authors receive payment every time a piece of software is installed by the user.

Adware can be both legitimate and illegitimate. Developers of free software can use ads as a means of receiving compensation for their work instead of charging a fee for their work upfront. However, illegitimate adware can take the form of maliciously bombarding the user with ads in an attempt to take advantage of the user for greater ad revenue. A notable example of adware is the “fireball” adware that hijacks web browsers to maliciously show ads and possibly download additional malware without the user’s consent [24].

2.2.3 Methods of Malware Spread

Malware developers have come up with various creative methods of spreading their creations from system to system to better ensure more successful infection campaigns. Most infections can be categorized into 3 different categories:

2.2.3.1 Social Engineering

Social engineering refers to the act of attempting to influence a victim to take some action that would be beneficial to the social engineer but less so for the victim. Within the context of malware and cybersecurity, social engineering can be applied to influencing targets to take actions that could eventually lead to a system or network compromise. The following are examples of social engineering attacks:

- Phishing emails with malicious email attachments have proven to be an effective entry point for malware to enter a victim's network. Typically, the authors of such emails try to make the email appear enticing to encourage potential victims to open malicious attachments. Such malicious attachments may be delivered in various file formats such as ".zip," ".pdf" and Microsoft Office file formats such as ".docx" and ".pptx." Other times, such files may be named to appear to be a benign file format, such as a Microsoft word document, when they are actually a disguised malicious executable file. The "Emotet" ransomware found success utilizing this method of spread [25].
- Criminals can utilize USB flash drives loaded with malware as a vector for attacking organizations [26]. Employees and individuals not trained in proper cybersecurity practices can be enticed to insert unknown USB flash drives into their computers that are connected to an internal network, allowing a potential attacker to gain access and pivot around the local network.

2.2.3.2 User Error

In scenarios of user error, a user might make a mistake, misuse, or otherwise make a poor judgment of a scenario that could lead to a security breach. The following are examples of user error within the context of malware and cybersecurity:

- Weak passwords for various servers can be a decent way for criminals to enter networks unauthorized. Remote connection servers such as Remote Desktop Protocol (RDP) or Virtual Network Computing (VNC) can be configured with poor passwords that can be easily guessed by automated software that attempts to connect to various servers for attackers to possibly begin a malware infection. Configuring a strong password will minimize this method of malware infection.
- Pirated software and “cracked” software usually available through various download websites and peer-to-peer (P2P) networks, will commonly be bundled with various pieces of malware. “Crackonosh,” a crypto mining malware, was distributed via cracked video games and reportedly yielded $2 million USD in Monero cryptocurrency [27].
- Users can provide processes with elevated system privileges to provide the process with more accessibility to the system in which it is executing. In doing so, it opens the possibility for the process to cause damage to a system with its elevated system privileges granted willingly by the user.

2.2.3.3 Software Vulnerability Exploitation

A software vulnerability refers to a programming mistake made in a software’s programming that can lead to security consequences if exploited, such as arbitrary code execution, security bypasses, or privilege escalation. Malware can take advantage of software vulnerabilities to help it spread from system to system. Despite software patches being developed to alleviate known software vulnerabilities, such patches may not immediately be installed, giving criminals an opportunity to attack unpatched systems. The following are example scenarios of software vulnerability exploitation:

- Drive-by downloads refer to downloads through a web browser without the user’s consent. Minimal interaction is required for drive-by downloads to occur – simply visiting a malicious website is enough for system infection. Such downloads can be made using malvertising (malicious advertising)

which typically makes use of exploiting software vulnerabilities in web browsers that allow attackers to install malware [26].

- Network propagation involves vulnerabilities of remote systems that attackers exploit to spread malware. Should an attacker successfully propagate throughout a network, entire organizations can be crippled. An unfortunate example is the “WannaCry” ransomware which managed to not only propagate across several networks throughout the world but also managed to cripple hospitals in the UK, where treatments and surgeries were reportedly delayed as a result [21].

2.2.4 Mitigations

As a response to the threat malware poses for businesses and individuals, mitigations have been developed to make successful malware infections more challenging for cyber criminals.

2.2.4.1 Antivirus Software

Antivirus Software is a piece of software designed to detect, block and remove malware programs from a system. Most antivirus software today employ various different strategies to detect malware:

- Definition Detection
    - Antivirus engines rely on definitions to locate new malware. Definitions refer to patterns and signatures that pertain to new malware to make detecting them possible.
- Heuristic Based Detection
    - This method of detection is coupled with definition detection for the best results. Heuristic detection is used when no hardcoded definition exists for a piece of malware, yet the engine is suspicious of a particular binary. Suspicious binaries are executed within a controlled sandbox under close supervision of the antivirus engine. Should the engine deem the behaviors of the binary to be malicious, the engine will flag the binary as malware and will quarantine the potential threat to prevent it from being run.

- Real Time Protection
    - Antivirus software typically embeds itself into the operating system itself to both proactively protect the user's system but also as a means of self-defense against malware attempting to turn off or otherwise defeat the antivirus engine. All files that are written to the disk are intercepted by the antivirus and are scanned for malware before being released onto the disk. Likewise, network traffic is also monitored by antivirus engines to stop threats before they become a problem.

2.2.4.2 Network Segregation and "Air Gapping"

Network segregation, the splitting up of a network into smaller, more manageable ones, can be useful to mitigate the spread of possible infections. Additionally, limiting the flow of traffic that enters or exits sections of a network can greatly reduce the attack surface a piece of malware could take advantage of. By extension, such preventative measures could be taken as far as "air gapping" machines, that is, keeping highly sensitive machines off the network entirely. It has been demonstrated to be possible to infect "air gapped" machines, however. The infamous "Stuxnet" malware was able to infect various "air gapped" machines inside Iran's nuclear facilities controlling sensitive infrastructure via infected USB flash drives [28].

2.2.5 Security Evasion Trends

Malware authors are well aware of the existence of antivirus engines, and the other methods of defense users employ to protect against malware and attempt to employ methods of their own to subvert these means of defense. In 2015, many malware programs analyzed were found to make use of techniques to attempt to make analysis more difficult [29]. A number of these techniques can include:

- Virtual machine detection methods
    - Virtual machines provide malware analysts with a temporary execution environment that can be reverted to a clean state easily, making it easier to maintain an analysis environment. Malware

authors are aware of this, so they incorporate virtual machine detection functions to make analysis more challenging. Should a malware program detect it is running within a virtualized environment, it will usually execute some benign-looking instructions before self-terminating to avoid detection.

- Malware sandbox detection methods
    - With the huge amount of malware being released daily [12], malware analysts make use of automated malware analysis sandboxes such as “cuckoo sandbox” [30] to collect information about suspicious binaries. These sandboxes run the malware binaries and record activity made by the binary in question. Similarly to virtual machines, certain behaviors of these sandbox environments can be different from non-sandboxed environments, which malware authors can try to detect to make an analysis of their malware more difficult with sandboxes. Likewise, to virtual machine-sensitive malware, should a piece of malware detect it is running within a sandbox, it usually will self-terminate before executing its main payload.
- Obfuscation
    - Obfuscation in the context of software and malware refers to the efforts to make a piece of software more difficult to understand while still being able to function as originally intended by the authors. Malware authors may make use of self-decrypting payloads or code virtualizers such as “VMProtect” [31] which transforms executable binaries into difficult-to-understand binaries that are very challenging and time-consuming to reverse engineer. A solution to this problem is to simply run the suspicious binary within a virtualized environment and observe the behaviors of the binary, further exemplifying the usage of virtualization in the malware analysis field.
- Fileless Malware
    - Fileless malware are highly advanced malware programs that do not require a file to run. This is to make it more difficult for forensic

analyzers to find evidence of the malware infection, as no file was written to the system's hard drive. Fileless malware runs entirely within a computer's memory, making it very challenging to detect.

2.2.6 Malware Analysis

As a response to criminals' large usage of malware, security researchers and malware analysts put forth malware analysis efforts to understand how malware works within contexts of incidence response, academic research, or searching for indicators of compromise. Various methods and techniques exist used to analyze malware, each with its own strengths and weaknesses. This section intends to document some of the well-known methods.

Malware analysis is separated into two categories, static malware analysis, and dynamic malware analysis.

2.2.6.1 Static Malware Analysis

Static malware analysis involves the analysis of malware without running the malware sample. Some examples of static malware analysis could include the following:

- Locating Hardcoded Strings
    - Software programs may contain various strings that can be read using a program that collects strings from files, such as "strings". Such strings could include various APIs the malware could call, referenced files, interesting IP addresses, DNS queries, and passwords.
- Finding Embedded Files
    - Some programs can have files embedded in them that are extracted during runtime. The static analysis could allow the extraction and analysis of these files without running the malware sample.
- Locating Virtualization or Sandbox Detection Functions
    - Virtualization and isolated sandbox analysis environments are heavily used in the malware analysis fields. Malware authors are

aware of this fact and commonly include functionality in their malware to detect and self-terminate their malware should virtualization or an analysis sandbox be detected. Static analysis of samples can reveal such methods to an analyst, allowing one to bypass what the samples are looking for to detect virtualization or sandboxes.

- Obfuscation Detection
    - To make analysis more difficult or to help evade detection, malware authors employ obfuscation in their malware creations. By looking at the entropies of the portable executable sections, an analyst can tell if a sample uses likely obfuscation. A number of analysis tools exist that can perform this task, such as “exeinfo” [32].
- Disassembly
    - This involves using disassembler software such as IDA Pro [33] or Ghidra [34] to analyze the CPU instructions that make up a malware sample. Disassemblers also allow analysts to “step through” the CPU instructions to make it easier for analysts to follow along with the functionality of a sample.
- Decompilation
    - In cases where a program makes use of an interpreted language such as Python, it is possible to recover the source code from the executable. In cases where programs are written in compiled languages such as C, some analysis tools such as Ghidra come with built-in “pseudo decompilers” which attempt to compile CPU instructions into high-level C code.

Static analysis allows an analyst to comb through samples with utmost precision, allowing for an understanding of all possible functionalities of a target binary. While very capable, manually disassembling and analyzing binaries can be greatly time-consuming and impractical in cases of larger, more complicated malware samples. Furthermore, should a sample be obfuscated, it may be even more taxing for an analyst to understand the sample’s functionality.

2.2.6.2 Dynamic Malware Analysis

Sometimes referred to as behavioral analysis, dynamic malware analysis involves the running of a malware sample and observing a sample's behavior to determine its functionality. Given that analysis takes place during the runtime of the suspicious binaries, dynamic malware analysis provides a practical solution to obfuscated samples that would otherwise be very challenging to analyze via static analysis. A typical implementation of dynamic malware analysis would be malware "sandboxes" - isolated environments designed to run, monitor and record malware behavior. This, however leads to a different problem. Malware authors are also aware of these automated sandboxes and have implemented functions in their malware to attempt to detect the presence of these sandboxes to make analysis more difficult. Similar to virtual machine-sensitive malware, should a malware program detect the presence of a sandboxed environment, it will likely not execute its primary payload before self-terminating, not allowing the sandbox to record any of its malicious acts, defeating the purpose of the sandbox altogether. While dynamic malware analysis allows an analyst a means of quickly understanding roughly how a malware sample functions, it will not give an analyst a full understanding of all the functionalities a binary may be capable of. Performing static analysis with a disassembler would have an advantage in this regard.

2.2.7 Methods of Malware Detection

Malware detection is the act of distinguishing a malicious binary from a benign one. Much research has been done in this field, and multiple different methods have been proposed. All such methods of detecting malware rely on extracted "features", pieces of information extracted through malware analysis that are used to deduce a binary to be malicious or benign. This section intends to detail some of the methods of malware detection.

2.2.7.1 Signature Based Detection

Signature based malware detection relies on features that pertain to a malware's software structure to determine maliciousness. Binaries are searched for certain structures known to be related to known malware and are flagged as malicious

should they possess such structures. Such a method has been largely adopted by commercial security systems such as antivirus programs [33] as it offers a fast means of detecting known malware. Unknown malware or malware that employs obfuscations, however, can defeat this method of detection. Some of the structures that are used in detecting malware can include:

- Printable strings
    - Looking for strings that are unique to known malware binaries can help security systems determine if a sample is benign or malicious.
- File hashes
    - Antivirus engines will record the cryptographic hashes of known malicious binaries. File hashes will identify exact matches to known malicious binaries.
- CPU instruction sequences
    - Some malware binaries can have unique byte sequences that can identify the malware sample to an antivirus engine or intrusion detection system.

2.2.7.2 Behavioral Based Detection

Behavior based malware detection involves the observation of a binary's actions to deduce maliciousness. Such a method of detection typically uses virtual machine or sandbox environments where binaries can be run without the risk of the binary causing damage to sensitive system resources or data. This method of detection intends to provide an answer to the challenges of detecting unknown malware where signature based detection falls short; malware binaries with altered structures can evade signature based detection, but if the behavior of the binary is similar to the original unmodified binaries, behavioral detection can flag the binary as malicious. While this method does provide a solution to the shortcomings of signature based detection, not all malware will run properly within the virtual machine or sandbox environments which can be problematic for this method of detection. As a result, some binaries may be flagged as benign when they are

actually malicious. Some of the behaviors that are monitored by sandboxes can include the following:

- Patterns in system API calls
    - Certain malware families can possess unique sequences of system API calls that could be used to identify a sample as malicious.
- File changes
    - Malware may add, modify or attempt to hide files on infected systems. File names, file contents and hashes of the files created could all be used to deduce a particular malware family binary.
- Windows registry changes
    - Some malware may make changes to the windows registry to help hide its presence or disable certain system features.
- Network activities
    - Malware families may generate specific DNS queries or contact certain remote servers, which could be used as an indicator of maliciousness.

2.2.7.3 Heuristic Based Detection

Heuristic based malware detection is a variation of behavioral based detection that makes use of different techniques, such as rules and machine learning classifiers [34]. This method of detection offers high accuracy in detecting unknown malware, but has difficulty with complex malware that may make use of advanced techniques such as zero day exploits or novel means of hiding or system persistence [33]. For this method to work, a very large database of rules will need to be maintained, which can make this method of detection laborious for researchers.

2.3 Virtualization Design

Virtualization involves the creation of virtual machines where the hardware resources exposed to the virtual machine are also virtualized. Modern virtualization follows the 3 requirements outlined by Goldberg and Popek's paper "Formal Requirements for Virtualizable Third Generation Architectures" [8] which included the following.

1. The virtual machine must provide an “essentially identical” environment to an original machine. An “essentially identical” environment refers to an environment where a program executing will operate in an identical way as if operated on a bare metal machine.
2. Processes running inside the virtual machine must execute their instructions directly on the processor of the host machine.
3. Virtualization must provide control over the resources provided to the virtual machines to protect against changing memory or data of the host machine and the other virtual machines also running. The hypervisor should have full control over the resources provided, while the processes running inside the virtual machine should not be able to access any resource not allocated to it by the hypervisor. The hypervisor should also be able to regain control over resources already allocated.

This section will further detail virtualization design, including the components that make up a typical virtual machine, challenges faced by virtualization, and the techniques used throughout history to virtualize various pieces of hardware.

2.3.1 Virtual Machine Components

A typical virtual machine is made up of the following components, as shown in Figure 1.

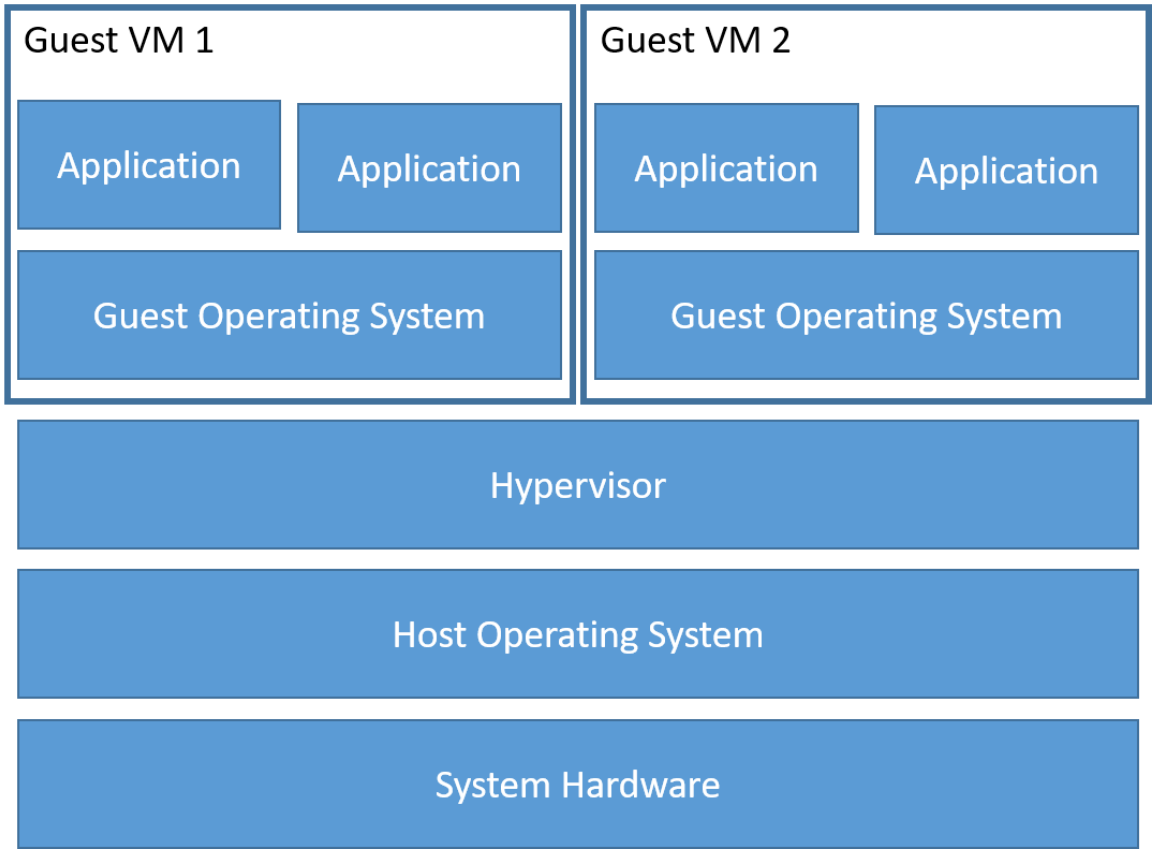


Figure 1: Virtual Machine Diagram

2.3.1.1 Host Machine

The host machine is the physical system that will be hosting the virtual machines (guest machines). A typical host machine is made up of system hardware, a host operating system and a hypervisor. The host machine is responsible for providing the resources necessary for the virtual machines (guest machines).

2.3.1.2 Virtual Machine

Also referred to as the guest machine, the virtual machine is the virtualized environment that acts like a computer, except it is not reliant on hardware like a physical system is. The host machine shares the resources necessary for the virtual machine to function.

2.3.1.3 System Hardware

System hardware refers to any hardware in the host machine that is to be shared or partitioned with the virtual machines. Some examples of such hardware could include the CPU, memory and hard drive storage.

2.3.1.4 Host Operating System

The host operating system is the operating system running on the host machine. The operating system is responsible for memory management and process management. The significance of the operating system comes into play when classifying the hypervisor.

2.3.1.5 Hypervisor

The hypervisor sometimes referred to as the virtual machine monitor (VMM), is responsible for the allocation of hardware resources from the host machine to the virtual machines. Acting as a software layer between the virtual machines and the hardware resources of the host machine, the hypervisor also acts as a boundary, allowing multiple separate virtual machines to run concurrently on the host machine. Depending on the implementation of the hypervisor, it can be classified in two ways:

- Type 1 hypervisors have direct access to the hardware and do not require an operating system for hardware access. Some examples of type 1 hypervisors include VMware ESXi, XEN, HyperV and Kernel-based Virtual Machine (KVM).
- Type 2 hypervisors require the operating system to gain access to the underlying hardware. Due to this type of hypervisor being reliant on the operating system, there is a greater inefficiency in performance when using this type of hypervisor. Some examples of type 2 hypervisors include VMware Workstation and VirtualBox.

2.3.2 Challenges facing Virtualization

Virtualizing the x86 CPU architecture was originally seen as a great challenge due to several technical aspects of the architecture and its application. This section summarizes these limitations.

2.3.2.1 Hardware Resource Sharing

Operating systems using this architecture assume they have full access to all hardware available on the system, making sharing resources awkward and difficult [9]. The challenge rested in the fact that the guest machine should not be able to modify resources to which the host machine has access. Giving the guest machine direct access to the hardware resources would interfere with the host machine, hence requiring the need for virtualized hardware that only the guest machine can access from the hypervisor.

2.3.2.2 x86 Privilege Rings

The x86 architecture operates with different levels of privilege, typically referred to as “rings” of privilege [11]. Each ring, ranging from 0 to 3, possesses varying levels of privilege to access memory and hardware and execute certain privileged instructions [35]. Ring 3 provides the least privilege and is where most applications execute. Applications at this ring are provided with no direct memory or hardware access and cannot run privileged instructions. Ring 0 on the other hand, is the most privileged ring where privileged instructions can be executed and direct memory and hardware access is granted. This aspect of the x86 architecture presented a problem for virtualization, as the privilege rings would need to be enforced by the hypervisor. A solution to this problem was to keep the hypervisor at ring 0, so it could still allocate the system resources necessary for the virtual machines, but the virtual machines were placed at ring 1, therefore allowing the hypervisor to intercept CPU instructions being executed by the virtual machines, while not giving it ring 0 privileges [9].

2.3.2.3 Privileged CPU Instructions

The challenges provided to virtualization by privileged CPU instructions go hand in hand with the challenges of enforcing the x86 privilege rings. Some instructions require ring 0 access to run, which is problematic, as this may give an opportunity for the guest machine to interfere with the host machine’s kernel. Binary translation, a proposed solution for this problem, involves the hypervisor trapping and translating these privileged instructions before they are executed [9]. Trapped

instructions are converted to different instructions so they can return values relevant to the original privileged instruction attempting to be executed. However, the hypervisor also typically resides in ring 0, which causes additional problems. Having both virtualized processes running privileged instructions and the hypervisor at ring 0 would make it difficult for the hypervisor to trap these privileged instructions. Furthermore, early implementations of binary translation were heavily taxing on guest machine performance due to the additional overhead it would cause for the hypervisor.

2.3.3 Virtualization Techniques

To address the numerous challenges that faced the virtualization of the x86 architecture, different techniques for virtualization were developed.

2.3.3.1 Full Virtualization

Full virtualization involves virtualizing a system without any modifications made to the virtual machine's operating system. The virtual machine will not be aware it is being virtualized as it is unmodified and will be provided all services it expects from a real physical system by the hypervisor [36]. For performance increases, virtual machine ring 3 processes are executed directly on the host machine's processor without any intervention from the hypervisor. As an answer to the challenges of virtual machines attempting to execute privileged CPU instructions, VMware has implemented binary translation into their virtualization products [9]. Figure 2 depicts a visual diagram of full virtualization that also makes use of binary translation.

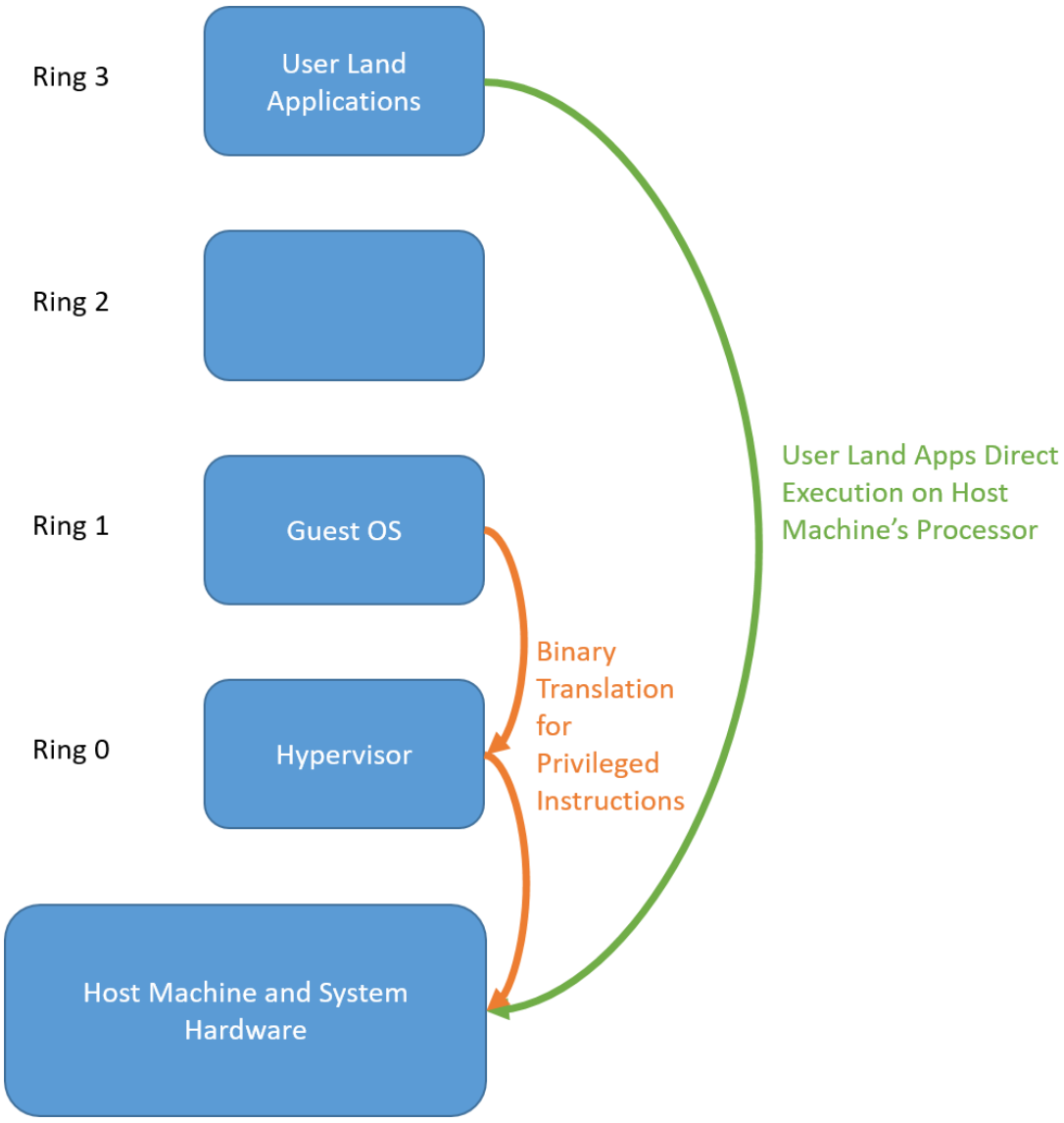


Figure 2: Full Virtualization with Binary Translation

2.3.3.2 Paravirtualization

Paravirtualization, also known as operating system-assisted virtualization, refers to the virtualization of an operating system that has been modified specifically to eliminate the need for binary translation for the privileged CPU instructions. The privileged instructions are instead replaced with calls to the virtualization layer that takes care of the sensitive kernel operations such as memory management and interrupt handling [9]. As a result, paravirtualization has less overhead and greater performance compared to full virtualization, which must perform binary translation. However, paravirtualization suffers from poor compatibility and portability due to

the kernel modifications of the guest operating system. XEN, an open-source hypervisor project, makes use of paravirtualization that utilizes a custom Linux kernel and device drivers to increase performance. Figure 3 shows a visual representation of how paravirtualization works at the ring level.

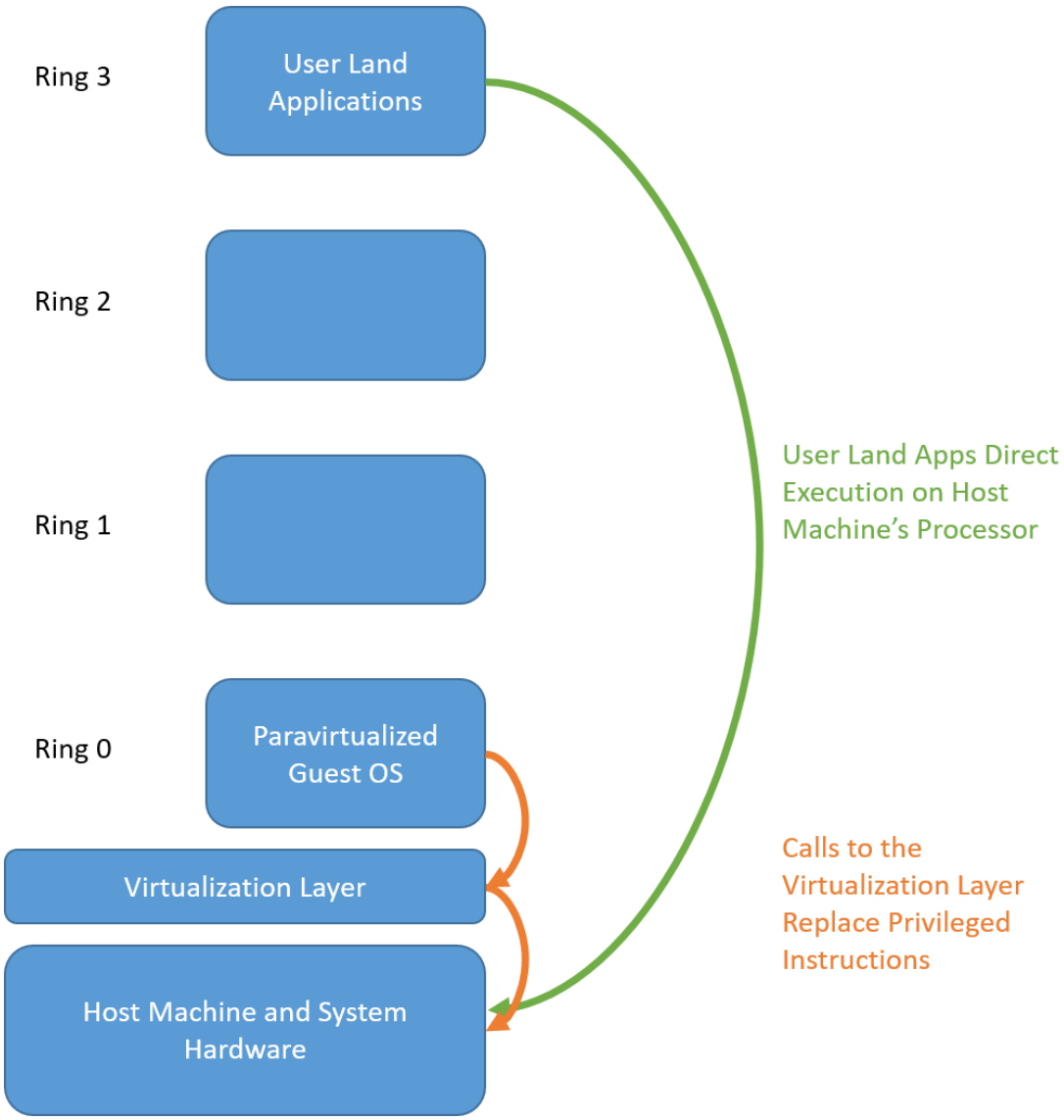


Figure 3: Paravirtualization

2.3.3.3 Hardware Assisted Virtualization

With the growing popularity of virtualization in the early 2000s, the hardware manufacturers Intel and AMD began shipping processors with virtualization extensions built in. Intel developed "Intel Virtualization Technology" (VT-x) [37]

while AMD developed “AMD Virtualization” (AMD-V) [38]. Both aim to provide a hypervisor at a ring level below ring 0. Privileged CPU instructions are trapped at the CPU hardware level, eliminating the need for binary translation in full virtualization or kernel modification in paravirtualization. These hardware-assisted virtualization technologies take care of additional hypervisor tasks such as memory management, memory address translation, and nested paging [9]. Figure 4 shows an example of how hardware-assisted virtualization could be implemented at the ring level.

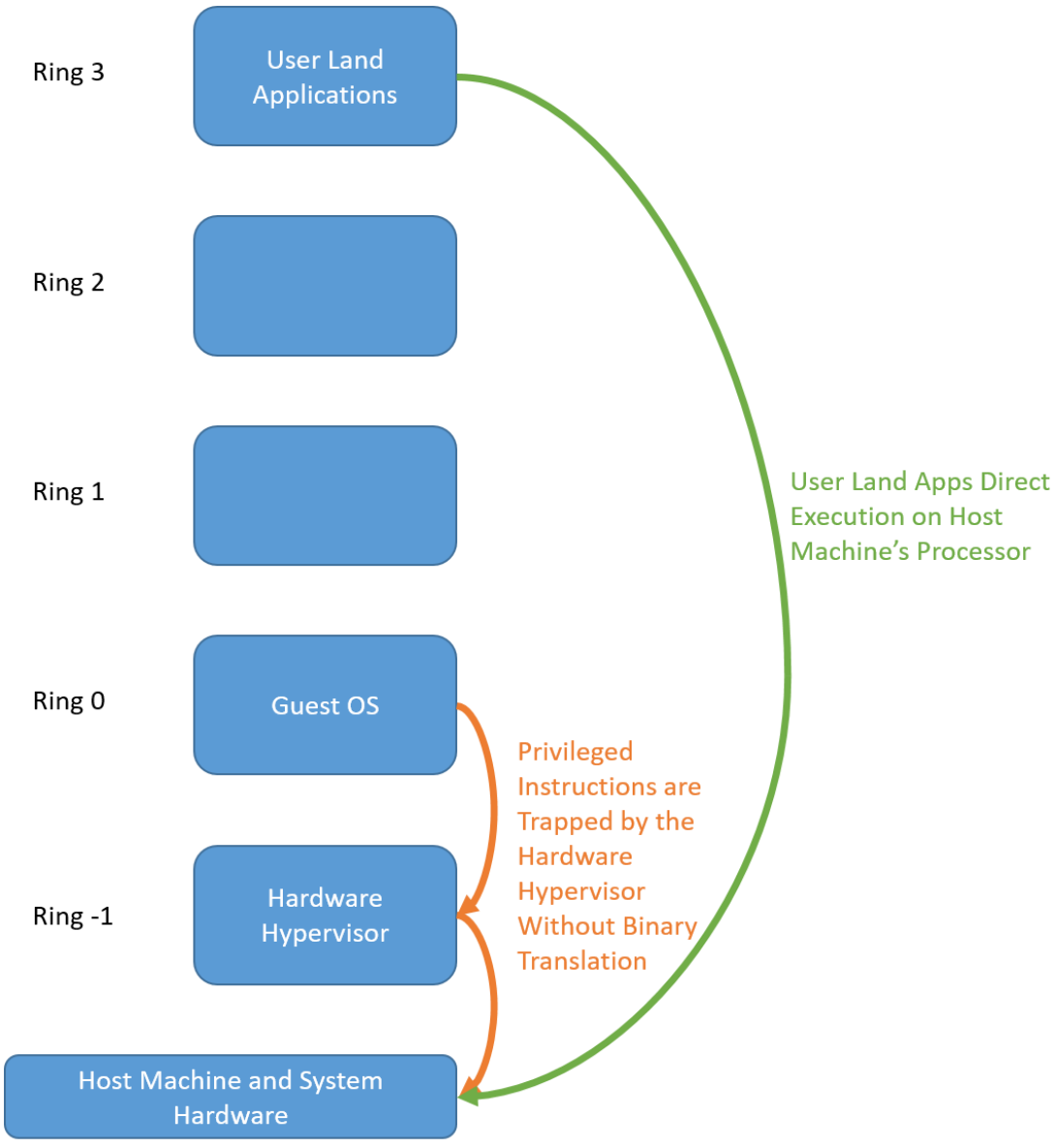


Figure 4: Hardware Assisted Virtualization

2.4 Memory Virtualization

Memory virtualization involves the sharing of system memory from the host machine to the guest machine. The hypervisor is responsible for allocating the set amount of system memory to a specified virtual machine. The system memory is managed by the operating system as virtual memory, and this virtual memory is, in turn, converted to actual physical memory addresses in random access memory (RAM) with the help of the Memory Management Unit (MMU) and the Translation Lookaside Buffer (TLB) hardware built into the CPU [39]. The MMU is responsible for the mapping between virtual and physical memory addresses, while the TLB is responsible for acting as a cache of recently used memory mappings for performance purposes. While bare metal operating system machines may have access to these physical aspects of the CPU, virtual machines do not. Given this, the virtual machine is expected to have access to actual physical RAM, but to not conflict directly with the host machine, it cannot be given direct access to the host machine's memory. The hypervisor, therefore must virtualize both the MMU and TLB along with managing the allocated memory from the host machine's operating system to provide virtual memory that the virtual machine will think is physical memory, thus "virtualizing" memory for the virtual machine. Figure 5 shows a visualization of how memory virtualization could be modeled.

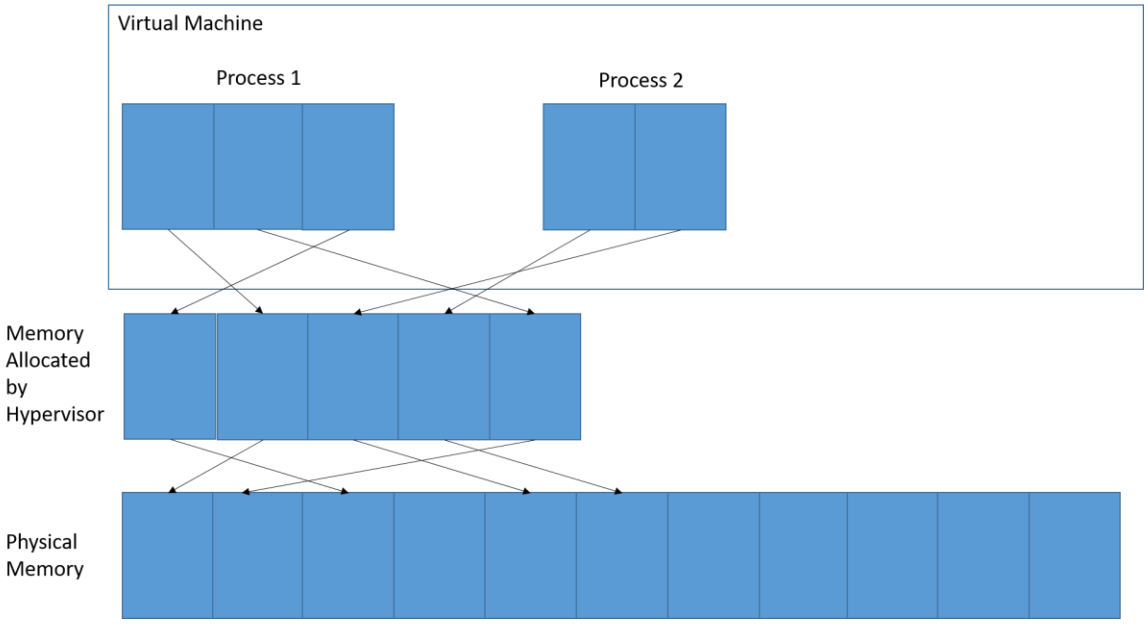


Figure 5: Memory Virtualization

Due to the additional layer of address translations imposed by the hypervisor, there is additional overhead with this schema. Optimizations were therefore proposed in the form of shadow page tables. Shadow page tables involve the hypervisor making use of the CPU's TLB hardware to translate the virtual memory used by the virtual machine to allow for a direct memory lookup and therefore reduce the overhead imposed by the hypervisor [9]. Should there be changes made in the virtual memory to physical memory mappings, the hypervisor updates the shadow page tables. Figure 6 shows how memory virtualization works with shadow page tables implemented.

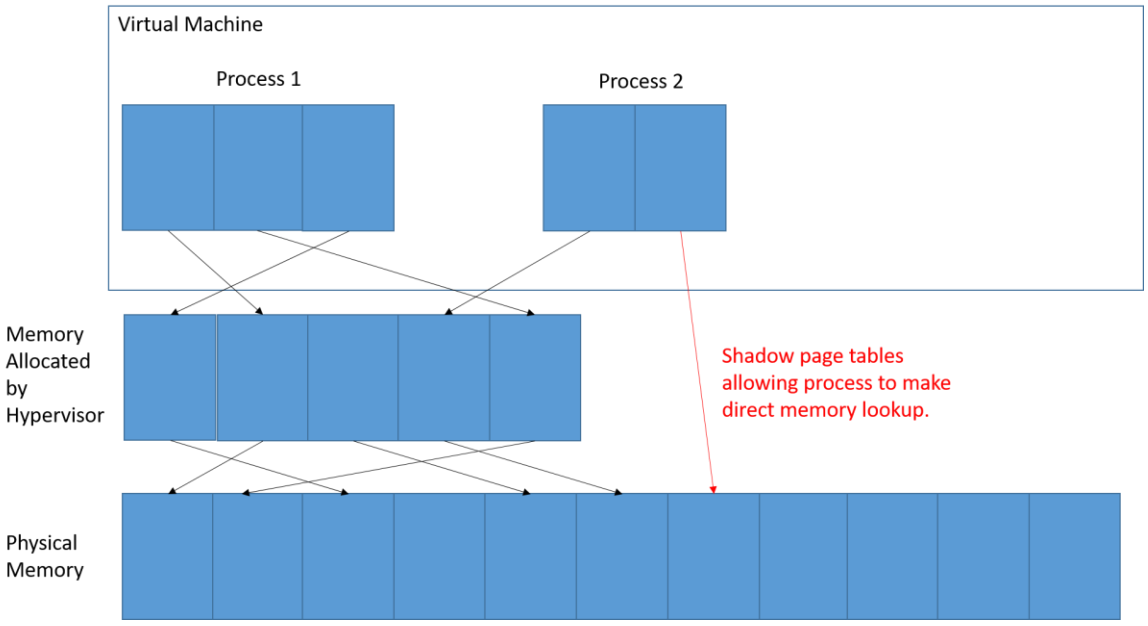


Figure 6: Memory Virtualization with Shadow Page Table Optimizations

2.5 Network Virtualization

Network virtualization refers to creating networks in software instead of physical network infrastructure. These virtualized logical network devices support sharing information and I/O operations as they would in physical form. As part of virtual machine isolation, logical devices such as network interface cards (NICs) can be created per virtual machine by the hypervisor, allowing for isolated networks created entirely in software [9]. This also allows for the reduction of networking

equipment needed within contexts such as cloud computing that typically involve many virtual machines running concurrently.

2.6 Benefits of Virtualization

Modern day virtualization offers many flexible benefits to users that include the following:

- Migration Capabilities
    - Virtual machines usually are bundled with settings of how the hypervisor is to treat the virtual machine and what hardware is to be virtualized for the VM. Such VMs are encapsulated into a number of files that can be transferred and run from one system to another, allowing VMs to be easily migrated to new systems by users [40]. This also allows for VMs to have the quality of hardware independence, as the VMs make use of virtualized hardware administered by the hypervisor.
- Snapshot Technology
    - Modern hypervisors allow for the saving of “snapshots” of VMs, which involves saving the current state of the VM as a file that can be reverted back to at a later time [41]. This technology can be used as a means of system backup in the event of system failure or can be used in scenarios where users may benefit from having an environment where they wish to revert changes they make to the VM, such as in malware analysis.
- System Isolation and Security
    - While VMs share the same hypervisor and system resources, each VM remains isolated from the other [40]. This allows for additional stability in the event that a VM fails, the others will remain functional. This feature can also be seen as beneficial in scenarios where users do not wish for changes in one VM to spread to others, such as malware infections.
- Software Licensing

    - Businesses may make use of proprietary software that requires a license in order to use it fully. With the usage of a virtual machine with proprietary software installed, businesses can migrate the virtual machine to different physical machines, allowing for greater flexibility in the software's usage.
- Better Usage of Computing Resources
    - By segmenting out the hardware resources of a single system, business owners can more effectively make use of the investment into expensive computing resources over having to purchase multiple single machines for each individual computing task. This leads to significant savings and increases in productivity for businesses. Coupling this with the above-mentioned advantages of virtualization, the global application virtualization market revenue is forecasted to increase from $2.09 billion USD in 2018 to $5.76 billion USD in 2026 [42].

2.7 Related Topics to Virtualization

Two topics that relate closely to virtualization are containerization and emulation. This section intends to briefly detail each of these topics.

2.7.1 Containerization

Containerization revolves around packaging a software application and its dependencies into a standalone file to be mounted and managed by a container engine. A visualization of a typical container implementation can be seen in Figure 7.

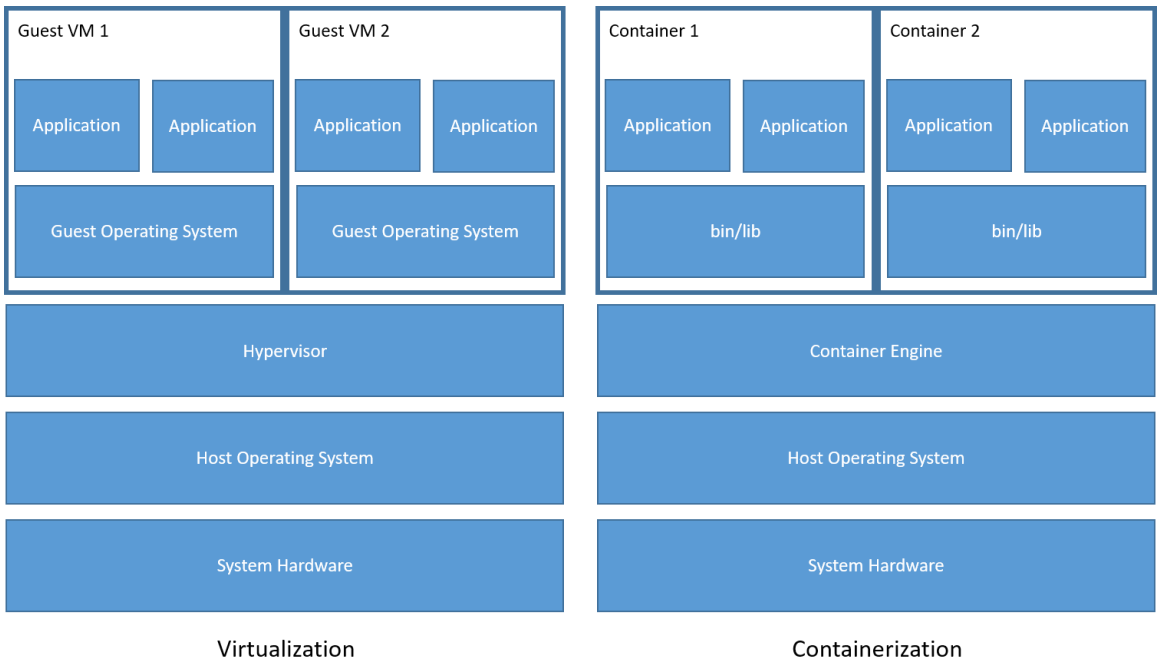


Figure 7: Containerization Vs. Virtualization

Unlike virtualization, containerization does not make use of a guest operating system or a hypervisor. Instead, it shares the kernel resources of the host operating system with the help of a container engine such as Docker [43], giving containerization the advantage of requiring less resource usage during runtime due to not having to support an additional kernel. Virtual machines support a kernel, system drivers, software applications, and installed libraries, while a container only supports software applications and the necessary libraries, allowing for containers to also have a much smaller file size compared to virtual machines [44]. With smaller container file sizes, containerization offers the advantage of easier portabilization of software packages and necessary dependencies without having to transport virtual machines, which can be very large in size. Containerization software such as Docker [43] and Kubernetes [45] have seen popularity in machine learning and AI research as it allows for the easy transfer of programming environments to different systems [46].

While containerization offers advantages in less resource usage and greater portabilization over virtualization, containers can pose a greater security risk than virtualization. Due to the fact that containers utilize the same kernel resources as

the host machine, should a mounted container exploit or otherwise gain arbitrary code execution at the kernel level, an attacker could compromise the security of the entire host machine. Unlike that of virtualization, if an attacker were to compromise a virtual machine, they would be confined to the virtual machine and would need a means of escaping the VM to compromise the host machine. Users of containerization should take proper precautions when mounting containers, such as restricting containers from root privileges and properly securing access, authorization, and Application Programming Interface (API) server access [44].

2.7.2 Emulation

Emulation within the context of computing refers to using software or hardware to allow for a system to behave like a different system. A typical application of emulation would be running binaries or operating systems that are compiled in a CPU architecture different to the system currently in use. The emulator is responsible for translating the CPU instructions of the binaries to ones that are understood by the system in use, along with emulating any hardware peripherals that the binaries would expect present as if they were run on actual legitimate pieces of hardware. A popular application of emulation is in the video game emulation field, where users are capable of running various video games on desktop computers and mobile phones that would normally require a video game console to play [47].

Emulation and virtualization have a number of similarities and differences that distinguish them from each other. A visualization comparing the two can be seen in Figure 8. Virtualization makes use of a hypervisor to share the hardware resources of the host machine with the VMs. While virtualization can emulate various adapters and pieces of hardware for a system, such as network adapters, USB controllers, and sound cards, all CPU instructions are executed natively on the processor of the host machine. Virtual machines using virtualization must be in the same architecture as the host machine. Emulation on the other hand, not only emulates all the hardware peripherals exposed to the guest machine but also emulates the CPU used for the guest [48]. This property of emulation allows for

the running of binaries of CPU architectures different to the architecture of the host machine. Unlike virtualization, emulation does not have direct access to hardware resources. Instead, it requires the emulator to act as a software “bridge” to the hardware resources the host machine has access to. It should be noted that due to the heavy reliance on the emulator to provide resources to the guest, emulators can be very slow when compared to the performance of virtualization.

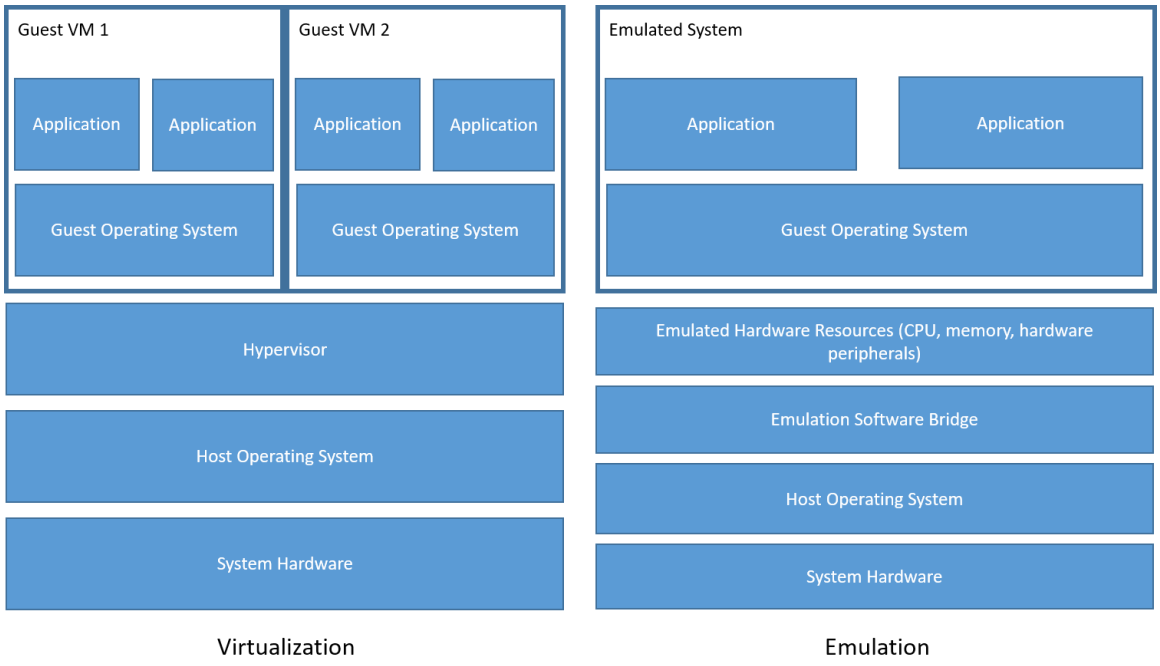


Figure 8: Virtualization vs. Emulation

Chapter 3. Related Works

This section of the thesis intends to document the known methods of virtual machine detection and the related research that has been done in this field. Malware is known to have abused some of these methods of detection, such as the “cryptowall” ransomware, which looked for the presence of certain system artifacts that were unique to a virtual machine. Given the fact that malware is run within virtual machines by malware analysts to observe their behavior safely, malware authors intend to obstruct analysis efforts by detecting the presence of virtual environments and altering the behavior of their malware should a virtual machine be detected. As for the “cryptowall” ransomware, if a virtual machine is detected, the malware will not decrypt the main ransomware payload and will instead cease execution to make analysis more laborious for malware analysts. The primary payload, in this case, included information such as the domain generation algorithm used to contact the command and control servers and the methods of system persistence, which are important pieces of information to an analyst intending to stop a malware campaign. From the perspective of the malware analyst, it is therefore important to understand the existing methods of detection and build solutions around these methods to make it harder for malware to detect their virtualized environments.

In this chapter, we have separated the methods of detection into three classifications based on the themes of the detections: artifact based detection, instruction based detection and miscellaneous detection methods. We intend to summarize the details of each classification along with assessing the use cases and the applicable history of each associated method. We will end the chapter with a summary of each classification and a comparison of each to our proposed method of detection. To distinguish the advantages of our method over the others that will be detailed in this chapter, we are focusing on aspects that we believe would of interest to the malware author. A malware author planning on implementing a means of virtual machine detection in their malware will be seeking a robust means of detecting virtual machines. We decided to assess the detection methods on the following aspects:

- No Elevated System Privileges Needed
    - Requiring elevated system privileges is an extra step that can complicate the job of the malware author. The malware author is ideally looking for a way of detecting virtual machines without needing to elevate their system privileges.
- Hypervisor Independence
    - Many methods of virtual machine detection are specific to particular hypervisors. This can make it laborious for the malware author to keep track of the many different details necessary to detect different hypervisors. We believe methods of virtual machine detection that are independent of the hypervisor are a plus for malware authors.
- Inconspicuous
    - Methods of detecting virtual machines should not attract unwanted attention from security systems or network administrators. Malware authors will want to keep their malware undetected for as long as possible to profit as much as they can from their creations. Activity such as excessive network traffic or opening of arbitrary network ports is considered suspicious to firewalls and antivirus security systems and may alert admins of such activity. On the other hand, activity such as searching for various files on a system or executing individual CPU instructions is considered inconspicuous as security systems typically do not alert admins regarding this activity.

3.1. **Artifact Based Detection**

Virtualized operating systems typically possess certain files, directories and other system artifacts that bare metal machines will not. Malware programs can look for the presence of these artifacts to determine if they are running inside a virtual machine. This simple method of detecting virtual machines is effective and inconspicuous, but some system artifacts may require enhanced system privileges to check for. In addition, artifacts can differ from one virtualization product to another, and to complicate things further, some of these artifacts can be removed altogether from the virtual machine. Finally, there exist countermeasures such as

“Ether” [4] which intend to be deployed on virtual machines to hide the presence of their system artifacts from malware. The following subsections are known artifacts for VMware Workstation, VirtualBox, HyperV, XEN, and KVM/QEMU at the time of writing.

3.1.1 Files and Directories

Various hypervisors typically install drivers and other software on guest virtual machines to allow guest machines to make use of features offered by the hypervisor. A VM-sensitive program could search for the presence of these files, directories or running processes that are known to be associated with a virtual machine.

VMware

- C:\Program Files\VMware\
- C:\Program Files\VMware\VMware Tools\
- C:\Program Files\VMware\VMware Tools\vmtoolsd.exe
- C:\Program Files\VMware\VMware Tools\vmwaretray.exe
- C:\Program Files\VMware\VMware Tools\vmwareuser.exe
- C:\Program Files\VMware\VMware Tools\VGAuthService.exe
- C:\Program Files\VMware\VMware Tools\vmacthlp.exe
- C:\Windows\System32\vm3dc003.dll
- C:\Windows\System32\vm3ddevapi64-debug.dll
- C:\Windows\System32\vm3ddevapi64-release.dll
- C:\Windows\System32\vm3ddevapi64-stats.dll
- C:\Windows\System32\vm3ddevapi64.dll
- C:\Windows\System32\vm3dgl64.dll
- C:\Windows\System32\vm3dglhelper64.dll
- C:\Windows\System32\vm3dservice.exe
- C:\Windows\System32\vm3dum64-debug.dll
- C:\Windows\System32\vm3dum64-stats.dll
- C:\Windows\System32\vm3dum64.dll
- C:\Windows\System32\vm3dum64_10-debug.dll

- C:\Windows\System32\vm3dum64_10-stats.dll
- C:\Windows\System32\vm3dum64_10.dll
- C:\Windows\System32\vm3dum64_loader.dll
- C:\Windows\System32\vmGuestLib.dll
- C:\Windows\System32\vmGuestLibJava.dll
- C:\Windows\System32\vmhgfs.dll
- C:\Windows\System32\VMWSU.DLL
- C:\Windows\System32\vsocklib.dll
- C:\Windows\System32\drivers\vm3dmp.sys
- C:\Windows\System32\drivers\vm3dmp_loader.sys
- C:\Windows\System32\drivers\vm3dmp-debug.sys
- C:\Windows\System32\drivers\vm3dmp-stats.sys
- C:\Windows\System32\drivers\vmnet.sys
- C:\Windows\System32\drivers\vmmouse.sys
- C:\Windows\System32\drivers\vmusb.sys
- C:\Windows\System32\drivers\vmci.sys
- C:\Windows\System32\drivers\vmhgfs.sys
- C:\Windows\System32\drivers\vmmemctl.sys
- C:\Windows\System32\drivers\vmx86.sys
- C:\Windows\System32\drivers\vmrawdsk.sys
- C:\Windows\System32\drivers\vmusbmouse.sys
- C:\Windows\System32\drivers\vmkdb.sys
- C:\Windows\System32\drivers\vmnetuserif.sys
- C:\Windows\System32\drivers\vmnetadapter.sys

VirtualBox

- C:\Program Files\Oracle\VirtualBox Guest Additions\
- C:\Program Files\Oracle\VirtualBox Guest Additions\VBoxTray.exe
- C:\Windows\System32\vboxdisp.dll
- C:\Windows\System32\vboxhook.dll
- C:\Windows\System32\vboxmrxnp.dll

- C:\Windows\System32\vboxogl.dll
- C:\Windows\System32\vboxoglarrayspu.dll
- C:\Windows\System32\vboxoglcrutil.dll
- C:\Windows\System32\vboxoglerrorspu.dll
- C:\Windows\System32\vboxoglfeedbackspu.dll
- C:\Windows\System32\vboxoglpackspu.dll
- C:\Windows\System32\vboxoglpassthroughspu.dll
- C:\Windows\System32\vboxservice.exe
- C:\Windows\System32\vboxtray.exe
- C:\Windows\System32\VBoxControl.exe
- C:\Windows\System32\drivers\VBoxGuest.sys
- C:\Windows\System32\drivers\VBoxSF.sys
- C:\Windows\System32\drivers\VBoxVideo.sys
- C:\Windows\System32\drivers\VBoxMouse.sys

KVM

- C:\Windows\System32\drivers\balloon.sys
- C:\Windows\System32\drivers\netkvm.sys
- C:\Windows\System32\drivers\pvpanic.sys
- C:\Windows\System32\drivers\viofs.sys
- C:\Windows\System32\drivers\viogpudo.sys
- C:\Windows\System32\drivers\vioinput.sys
- C:\Windows\System32\drivers\viorng.sys
- C:\Windows\System32\drivers\vioscsi.sys
- C:\Windows\System32\drivers\vioser.sys
- C:\Windows\System32\drivers\viostor.sys
- C:\Program Files\Virtio-Win\

QEMU

- C:\Program Files\qemu-ga
- C:\Program Files\SPICE Guest Tools

3.1.2 Devices

In Windows Virtual Machines, “Devices” exist as a means of addressing hardware resources such as peripheral devices and system processors. Virtual machines typically possess uniquely named “devices”, which could be indicators of the presence of a Virtual Machine.

VMware

- \\\\.\\HGFS
- \\\\.\\vmci

VirtualBox

- \\\\.\\VBoxMiniRdrDN
- \\\\.\\VBoxGuest
- \\\\.\\pipe\\VBoxMiniRdDN
- \\\\.\\VBoxTrayIPC
- \\\\.\\pipe\\VBoxTrayIPC

3.1.3 Shared Folders

Shared folders are a mechanism introduced in virtual machines for easy file transfer between the guest and host. Checking for the presence of known folders associated with shared folders can be used to deduce the presence of a virtual machine.

VirtualBox

- “VirtualBox Shared Folders”

3.1.4 MAC Addresses

Hypervisors typically provide virtual machines with virtualized Network Interface Cards (NICs) that possess a MAC address exclusive to the vendor of the hypervisor. The following are listings of known MAC addresses. The X’s in the listed Media Access Control (MAC) addresses can be any arbitrary hexadecimal value.

VMware

- 00:50:56:XX:XX:XX
- 00:1C:14:XX:XX:XX
- 00:0C:29:XX:XX:XX
- 00:05:69:XX:XX:XX

VirtualBox

- 08:00:27:XX:XX:XX

Hyper-V

- 00:15:5D:XX:XX:XX

XEN

- 00:16:3E:XX:XX:XX

3.1.5 BIOS Strings

Virtualization products typically ship with a Basic Input/Output System (BIOS), which contains strings that reveal the presence of a virtual machine. This method typically makes use of checking specific Windows registry values.

VMware

- HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System contains the value “SystemBiosVersion” which contains the string “VMware, Inc.”
- HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System\BIOS contains various data values that are unique to VMware machines.
    - “BIOSVendor” will be “VMware, Inc.”
    - “BIOSVersion” will start with the character “VM.”
    - “SystemManufacturer” will be “VMware, Inc.”
    - “SystemProductName” will contain the string “VMware.”
- The same information shown in the bullet above is also available in HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\SystemInformation

VirtualBox

- HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System contains various data values that are unique to VirtualBox machines.
    - “SystemBiosDate” will be “06/23/99.”
    - “SystemBiosVersion” contains the string “VBOX.”
    - “VideoBiosVersion” contains the string “Oracle VM VirtualBox.”

3.1.6 SMBIOS Strings

The System Management BIOS (SMBIOS) is a series of data structures that are read by the BIOS for management purposes. It is possible for a piece of software to search the SMBIOS for strings exclusive to certain hypervisors. The following are known strings.

VMware

- Will contain the string “VMware”

VirtualBox

- Will contain “VirtualBox,” “vbox” or “VBOX”

QEMU

- Will contain “qemu” or “QEMU”

3.1.7 ACPI Strings

Advanced Configuration and Power Interface (ACPI) are interfaces used by an operating system for communication with various hardware components through the BIOS or UEFI. Virtual machines possess strings that will give away their presence.

VMware

- Will contain the string “VMware”

VirtualBox

- Will contain “VirtualBox”, “vbox” or “VBOX”

QEMU

- Will contain “BOCHS” or “BXPC”

3.1.8 Miscellaneous Windows Registry Keys

The Windows registry is littered with clues that malware and other VM sensitive software may use to detect a virtual machine’s presence. The registry keys listed in this section exclude the ones mentioned to be associated with BIOS information.

VMware

- The following keys contain the string “VMWARE”
    - HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\Logical Unit Id 0
    - HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\Scsi\Scsi Port 1\Scsi Bus 0\Target Id 0\Logical Unit Id 0
    - HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\Scsi\Scsi Port 2\Scsi Bus 0\Target Id 0\Logical Unit Id 0
- HKEY_LOCAL_MACHINE\SOFTWARE\ VMware, Inc. \VMware Tools

VirtualBox

- HKEY_LOCAL_MACHINE\HARDWARE\ACPI\DSDT\VBOX__
- HKEY_LOCAL_MACHINE\HARDWARE\ACPI\FADT\VBOX__
- HKEY_LOCAL_MACHINE\HARDWARE\ACPI\RSDT\VBOX__
- HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\VBoxGuest
- HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\VBoxMouse
- HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\VBoxService
- HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\VBoxSF
- HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\VBoxVideo

KVM

- HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\vioscsci
- HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\viostor
- HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\VirtIO-FS Service

- HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\VirtioSerial
- HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\BALLOON
- HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\BalloonService
- HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\netkvm

QEMU

- The following keys contain the string “QEMU”
    - HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\Logical Unit Id 0
    - HKEY_LOCAL_MACHINE\HARDWARE\ Description\System

3.1.9 Running Processes

Some virtual machines make use of certain processes running in the background that are specific to a particular virtualization product. Typically, these processes will act in a way that provides convenience to the user, such as auto-configuring the screen resolution and configuring networking settings.

VMware

- Vmtoolsd.exe
- Vmwaretray.exe
- Vmwareuser.exe
- VGAuthService.exe
- Vmacthlp.exe

VirtualBox

- Vboxservice.exe
- Vboxtray.exe

XEN

- Xenservice.exe

QEMU

- Qemu-ga.exe
- Vdagent.exe
- Vdservice.exe

3.1.10 Driver Objects

Driver objects act as a means of processing to communicate with driver software loaded in kernel memory. Some driver names are exclusive to virtual machine-based drivers and, therefore can be used as a means of identifying the presence of a virtual machine.

HyperV

- \\Driver\VMBusHID
- \\Driver\Vmbus
- \\Driver\Vmgid
- \\Driver\IndirectKmd
- \\Driver\HyperVideo
- \\Driver\hyperkbd

3.1.11 Global Objects

A global object in the context of the Windows operating system refers to any system object or resource that can be used by any process or thread running in the current session. Some global objects are exclusively used by certain virtual machines hypervisors which can be used to identify the presence of a virtual machine.

HyperV

- \\GLOBAL??\VMBUS#
- \\GLOBAL??\VDRVROOT
- \\GLOBAL??\VmGenerationCounter
- \\GLOBAL??\VmGid

3.2. Instruction Based Detection

Instruction based virtual machine detection hinges on the fact that some CPU instructions, when executed inside a virtual machine, will return different values than what they would return if they were executed on a bare metal machine. A malware program could execute these instructions and attempt to deduce the presence of a virtual machine based on the return values. While this presents a method of detection that is minimalistic and inconspicuous to security systems, few instruction based detection methods work today. Furthermore, these methods are specific to certain hypervisors and therefore will not work universally. Nonetheless, we still believe it is important to list the instruction based methods that have been used to detect virtual machine presence throughout computing history. The following are known instructions that have been used for virtual machine detection in history at the time of writing.

3.2.1 IN

The IN command reads a value from a specified I/O port to a specified memory address. I/O ports serve as a means of communication between the operating system and I/O devices such as disk controllers and sound devices. VMware based virtual machines possess an I/O device at port number 0x5658 that allows for communication between the hypervisor and virtual machine [49]. Ordinarily, running the IN command in unprivileged user mode will raise a privileged instruction exception. In VMware however, no such exception is raised. The company VMware has even acknowledged this as an official means of detecting VMware hypervisors, making this a highly reliable means of detecting VMware based hypervisors [50]. A typical implementation of such detection can be seen in Figure 9.

```
bool isVmwarePresent()
{
    __try
    {
        _asm
        {
            mov eax, 0x564d5868
            mov ebx, 0
            mov cx, 1
            mov dx, 0x5658
            in  eax, dx
        }
        return true;
    }
    __except (GetExceptionCode() == EXCEPTION_PRIV_INSTRUCTION)
    {
        return false;
    }
}
```

Figure 9: Test for VMware's I/O Port

The implementation checks to see if a privileged instruction exception is raised when attempting to run the IN command. If an exception is raised, we are not inside a VMware virtual machine. Otherwise, we assume we are inside a VMware virtual machine. Testing this code on VMware, this method is still a valid way of detecting VMware hypervisors. It should be noted that VMware has provided an interface to allow users to “turn off” this method, providing users with an easy countermeasure to this method of detection.

3.2.2 SIDT

SIDT is an Intel x86 command that was used to detect virtual machine presence on older single-core processors. The usage of this command to detect virtual machine presence was most notably used by Tobias Klein [51] in his tool “ScoopyNG” and later by Joanna Rutkowska [5] in her tool “Redpill”. The command stores the address of the interrupt descriptor table register (IDTR) in a specific memory address. The interrupt descriptor table is a vital operating system data structure used when handling interrupt commands such as those from hardware like keyboards and mice. The usage of this command in the context of virtual machine detection depends on a hypervisor moving the address of the virtualized operating system’s interrupt descriptor table to a predictable address unique to the

hypervisor. Klein [51] and Rutkowska [5] both found that VMware products at the time stored the interrupt descriptor table at address 0xFFXXXXXX, allowing for simple yet effective detection of the VMware hypervisor on single core processors. Since the introduction of multicore processors, however, this technique of virtual machine detection has become less reliable as each core inside the processor possesses a unique interrupt descriptor table, each with a different address. Furthermore, when we tested the SIDT command on a virtual machine running under a VMware Workstation hypervisor, we found that the address of the table to never be at an address of 0xFFXXXXXX, even when only a single processor core was assigned to the virtual machine. Quist and Smith [52] proposed two possible solutions to this problem. The first solution was to run the SIDT command in a loop and keep track of the interrupt descriptor table addresses returned to profile the processor. This was found to be heavily taxing on the processor and not always functional. The second solution involved calling the "SetThreadAffinityMask" Windows API function [53], which allows a process to specify which processors it is to run on, eliminating the problem of getting multiple different interrupt descriptor table addresses. This was found to be problematic, however, as the hypervisor can still schedule virtual machines to run on varying processor cores, making this solution fail. Despite Quist and Smith's efforts, there are no currently known consistent indicators returned from this command and therefore, this method of virtual machine detection is no longer functional or used today.

3.2.3 SGDT

SGDT is an Intel x86 command that returns the address of the Global Descriptor Table Register (GDTR) to a specific memory address. The global descriptor table is a data structure assigned to a processor and is used to define details regarding process memory, such as base address, size and memory permissions [54]. This method of virtual machine detection was notably used by Tobias Klein's tool "ScoopyNG". Similar to the SIDT command, this command will return a value of 0xFFXXXXX when run inside a VMware based virtual machine. This command also suffers from the same problem SIDT experienced with the introduction of multicore processors. Testing this method of detection with VMware Workstation

has yielded returned values that are not 0xFFXXXXXX. Therefore, this is no longer a viable option for virtual machine detection.

3.2.4 SLDT

Store Local Descriptor Table (SLDT) is an Intel x86 command that returns the location of the local descriptor table register [55] to a specified memory address. The Local Descriptor Table (LDT) holds information regarding process memory segments such as the code, data and heap segments [56]. When this command is executed within a virtualized environment, a non-zero value will be returned, while 0 is returned in non-virtualized environments. Notable usage of this method of virtual machine detection was used by Tobias Klein's "ScoopyNG" tool [51] and the "Conficker" malware in 2009 [57]. We tested this command on VMware Workstation, VirtualBox, XEN, HyperV and KVM/QEMU virtual machines and found that all consistently returned a value of 0. This method of detection is no longer effective.

3.2.5 STR

STR is an Intel x86 command that returns the address of the Task State Register (TSR) of the currently executing process to a specific memory address. The task state register is a data structure used by the operating system for task switching, which involves the storing of a process' threads and registry information to be stored and later resume execution by the processor, controlled by the operating system's scheduler. Klein [51] found that this command returns an address of the format 0x0040XXXX when run within a virtual machine. Testing this command today on VMware Workstation, VirtualBox, XEN, HyperV and KVM/QEMU machines has shown that the command no longer returns an address of the format found by Klein. This method of detection is no longer used.

3.2.6 SMSW

SMSW is an Intel x86 command that returns a machine status word to a specific memory address. Information about the machine status word is pulled from the CR0 register, and it pertains to various system flags used by the processor to indicate if certain features are enabled, such as memory paging. Quist [58]

discovered that running this command inside a VMware virtual machine with EAX containing the value 0xCCCCCCCC will consistently yield a value of 0x8001XXXX. It should be noted that Quist's research dates back to 2006, and CPUs have greatly changed since then. As such, testing this command inside and outside a VMware virtual machine both returned a value of 0x8005XXXX instead of the 0x8001XXXX value claimed by Quist. There is no known distinguishable difference between running this command within a virtual machine or on a baremetal machine, making this method of detection nonfunctional.

3.2.7 CPUID

CPUID is an x86 instruction that obtains various pieces of information about the CPU it is executed on. The instruction works by taking the value in the EAX register to determine what information to return in the EAX, EBX, ECX and EDX registers.

Calling CPUID with an EAX value of 0 will return a 12-character manufacturer ID string in the order of the EBX, EDX and ECX registers. Intel CPUs return "GenuineIntel" [59] and AMD CPUs after the K5 model return "AuthenticAMD" [60]. This manufacturer ID is replaced by custom strings that are exclusive to certain virtual machines when run inside virtual machines. Table 1 shows the known virtual machine strings.

Vendor	**String**
QEMU	"TCGTCGTCGTCG"
KVM	" KVMKVMKVM "
VMWARE	"VmwareVmware"
VIRTUALBOX	"VBoxVBoxVBox"
XEN	"XenVMMXenVMM"
HYPERV	"Microsoft Hv"
PARALLELS	" Prl hyperv "
PARALLELS	" lrpepyh vr "
BHYVE	"bhyve bhyve"
QNX	"QNXQVMBSQG"

Table 1: Virtual Machine Manufacturer Strings *[61]*

Testing this on VMware Workstation, VirtualBox, XEN, HyperV and KVM/QEMU hypervisors did not return the expected strings however, and instead returned the appropriate strings associated with Intel and AMD, making this method of virtual machine detection no longer functional.

Calling CPUID with an EAX value of 1, will return information regarding the processor's features, such as supported instructions by the processor. The 31st bit of the value returned in ECX determines if a hypervisor is present. This value will be 0 on non-virtualized environments and 1 inside virtual machines. Testing this on our testing hypervisors, this method successfully determined the hypervisor's presence. This method of virtual machine detection is well known and has been abused widely by malware [62]. It should be noted that it is possible for some administrators of virtual machines to spoof CPUID results, which can defeat the aforementioned CPUID detection methods.

3.2.8 VM "Synthetic Instructions"

Traut [63] obtained a patent in 2003, which described a method of virtual machine detection by using "synthetic CPU instructions." These instructions are only supported by virtual machines and otherwise raise invalid instruction exceptions when executed on bare metal machines. Due to these being invalid CPU instructions, these instructions are not documented in Intel's or AMD's programmer manuals [64] [55]. Table 2 shows the instructions listed by the patent.

Mnemonic	**Opcode**
VMCPUID	0F C6 28 01 00
VMGETINFO	0F C6 28 00 00
VMSETINFO	0F C6 28 00 01
VMDXDSBL	0F C6 28 00 02
VMDXENBL	0F C6 28 00 03
VMHLT	0F C6 28 01 01
VMSPLAF	0F C6 28 01 02

VMPUSHFD	0F C6 28 02 00
VMPOPFD	0F C6 28 02 01
VMCLI	0F C6 28 02 02
VMSTI	0F C6 28 02 03
VMIRETD	0F C6 28 02 04
VMSGDT	0F C6 28 03 00
VMSIDT	0F C6 28 03 01
VMSLDT	0F C6 28 03 02
VMSTR	0F C6 28 03 03
VMSDTE	0F C6 28 04 00

Table 2: Synthetic Instructions

These instructions follow the format “0FC7C8XXXX”, where the X’s are replaced with hex values that correspond to a specific synthetic instruction. One of the most notable of these instructions is the “VMCPUID” command, which was used by the “Necurs” botnet malware to check for the presence of virtual machines [65]. We tested this command on VMware Workstation, VirtualBox, XEN, HyperV and KVM/QEMU hypervisors, and the code used can be seen in Figure 10.

```
bool vmcpuid_supported()
{
    bool supported = true;
    void* mempool;
    char vmcpuid[] = "\x0F\xC7\xC8\x01\x00"; // VMCPUID OPCODE
    mempool = malloc(sizeof(vmcpuid) );
    memcpy(mempool, &vmcpuid, sizeof(vmcpuid));
    DWORD prevAccess;
    VirtualProtect(mempool, sizeof(vmcpuid), PAGE_EXECUTE_READ, &prevAccess);

    _try
    {
        ((void(*)())mempool)();
    }
    _except(EXCEPTION_EXECUTE_HANDLER)
    {
        supported = false;
    }
    return supported;
}
```

Figure 10: VMCPUID Usage

Due to the fact that the VMCPUID command is invalid, most compilers will not recognize it. By setting a memory pool, copying the VMCPUID opcode to it, and changing executable permissions, we can run the command in a try, except for block. If the command is supported, no exception will be raised and we can assume we are inside a virtual machine. Otherwise, the raised exception will be caught and the software will determine it is outside a virtual machine. None of the hypervisors we tested this with supported the VMCPUID command. The command was instead misinterpreted as two commands instead of one:

- CMPXCHG8B EAX
- ADD DWORD PTR DS:[EAX],EAX

On a related note, we were able to find a report that claims this command worked under VirtualPC hypervisors [66], but we were unable to test the validity of these claims due to VirtualPC being discontinued and unavailable at the time of writing. Regardless, this is no longer a reliable means of detecting virtual machine environments.

3.2.9 VPCEXT

VPCEXT is a command used to detect the presence of the VirtualPC hypervisor [67]. Similar to the “synthetic CPU instructions" mentioned in Table 2, this command is also considered an invalid instruction [68] and is therefore, not documented in Intel’s or AMD’s programmer manuals [64] [55]. The following are known VPCEXT opcodes.

- 0F 3F 05 XX
- 0F 3F 05 XX
- 0F 3F 07 XX
- 0F 3F 0D XX
- 0F 3F 10 XX

If the execution of this command does not raise an illegal instruction exception, a process can determine it is running within a VirtualPC virtual machine. Due to

VirtualPC being depreciated by Microsoft, this is no longer a viable means of detecting virtual machines.

3.3. Miscellaneous Detection Methods

The methods listed in this section of the chapter hold no similarity to each other unlike the previously mentioned methods of VM detection. We, therefore, listed these as “miscellaneous” methods. We will address each of these methods separately.

3.3.1 Thermal Zone Temperature

A novel method of detecting virtual machine presence by checking for system thermals was first seen by the malware “gravityRAT” [69]. By using a powershell script in Figure 11 [70], it is possible to easily check to see if a machine supports an “MSAcip_ThermalZoneTemperature” object. This provided the method to be hypervisor independent, inconspicuous, and not require any elevated privileges to work.

```
function Get-AntiVMwithTemperature {
    $t = Get-WmiObject MSAcpi_ThermalZoneTemperature -Namespace "root/wmi"

    $valorTempKelvin = $t.CurrentTemperature / 10
    $valorTempCelsius = $valorTempKelvin - 273.15

    $valorTempFahrenheit = (9/5) * $valorTempCelsius + 32

    return $valorTempCelsius.ToString() + " C : " + $valorTempFahrenheit.ToString() + " F : " + $valorTempKelvin + "K"
}
```

Figure 11: Thermal Zone Temperature Script

If this script returns no results, it is assumed that the machine is within a virtual machine, as virtual machines do not support thermal monitoring by default. Testing this method on multiple bare metal systems has shown that this method is unreliable due to inconsistencies in the results. Thermal zone sensors can be disabled and enabled within a system’s BIOS or UEFI, which explains the inconsistencies. Overall, this is not a very reliable means of detecting virtual machines.

3.3.2 IP Timestamp Patterns

This method of virtual machine detection was discovered by Noorafize et al. [71] and involved the inability of virtual machines to keep accurate time like that of a

bare metal machine. Virtual machines are typically time synced by time sharing with the host computer to attempt to disguise themselves as a physical machine, but inconsistencies arise in Internet Protocol (IP) timestamps originating from virtual machines. It was found that the timestamp information received from virtual machines was slightly different from what the proper timestamp data should have been, given that virtual machines possess greater delay due to having to interact with the hypervisor, unlike a bare metal machine. For this method of detection to work, attackers would need to send a sizable number of IP/Internet Control Message Protocol (ICMP) packets at target systems to obtain enough information to deduce whether the target machine is virtualized or not. Furthermore, an attacker would need the ability to send packets to the machine in question, unlike the previously mentioned methods, which only require code execution on the target machine itself.

3.4. **Comparisons with Our Proposed Method**

The below table summarizes the known methods of virtual machine detection listed in this chapter. We included our proposed method at the bottom of the table.

VM Detection Method	**No Elevated Privileges?**	**Hypervisor Independence?**	**Inconspicuous?**
Artifacts	No	No	Yes
VMware “Backdoor”	No	No	Yes
SIDT, SGDT, SLDT, STR and SMSW Instructions	Yes	No	Yes
CPUID Hypervisor Bit	Yes	No	Yes
Synthetic CPU Instructions	Yes	No	Yes
Thermal Zone Check	No	Yes	Yes
IP Timestamp Method	Yes	Yes	No

Our Proposed Method (Not all CPU cores assigned to VM)	**Yes**	**Yes**	**Yes**

Table 3: Comparison of Methods of VM Detection

As seen in the table above, the existing methods of virtual machine detection have distinct strengths and weaknesses with their uses. Some offer inconspicuous means of detection, for instance, but may require elevated system privileges to function at all. Our proposed method offers virtual machine detection that does not require elevated privileges, is inconspicuous to security systems, and is independent of the hypervisor. To the best of our knowledge, at the time of typing, no other methods of virtual machine detection make use of CPU core counts and CPU caching capacities as a means of detecting virtual machines, making this both unique and inconspicuous to security systems. Given that our method involves looking at CPU core counts and caching capacities, our method is independent of the hypervisor in use. For these reasons, we believe our method of detection is both robust and novel compared to the existing methods of detection and may be of interest to malware authors.

Chapter 4. Proposed Method

This section addresses details of our proposed method of virtual machine detection. It is common for malware to include virtual machine detection functions to evade the detection of security systems such as antivirus engines. Given this, it is of utmost importance that malware analysts and security researchers locate and block new methods of virtual machine detection methods that malware authors could abuse to keep their malware undetected for longer.

4.1 Overview

Our VM detection method works by comparing information gathered about the CPU with information already known about the CPU from the manufacturer. If inconsistencies are found, we assume a virtual machine is present. We chose to look at the following pieces of information.

- Number of Physical Cores
- Number of Logical Cores/Threads
- Total Cache L1 Capacity
- Total Cache L2 Capacity
- Total Cache L3 Capacity

We separate our method into 3 steps. We will expand on each step in the next sections.

1. Collect CPU Information
2. Lookup CPU Information
3. Determine if VM is Present

The overall procedure can be summed up in this flowchart:

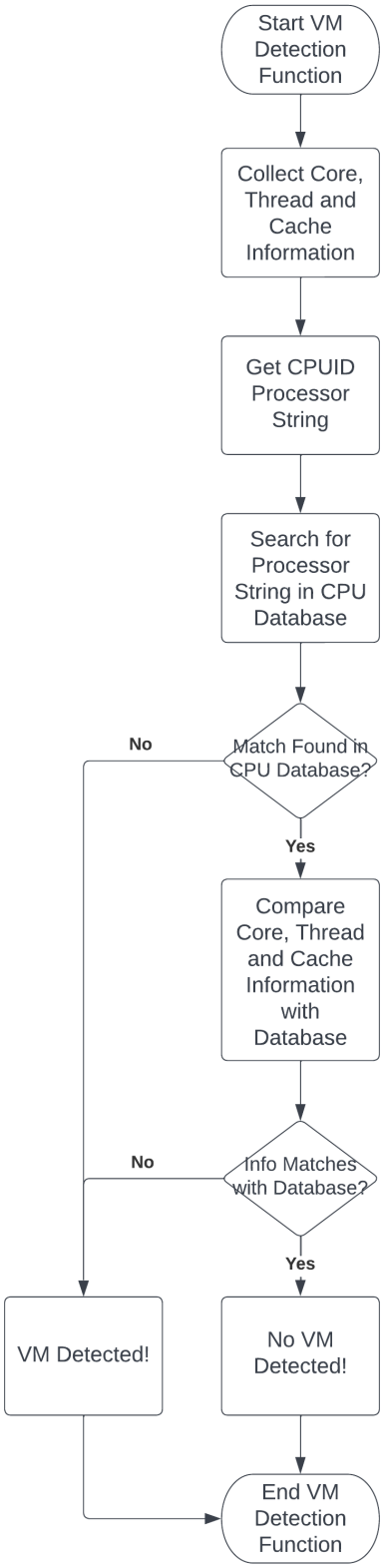


Figure 12: Proposed VM Detection Flow Chart

4.2 Collect CPU Information

We begin by collecting information about the CPU model and manufacturer. We collect this information so we can identify what CPU to compare to in the known CPU database.

We identify this information using the assembly command CPUID. This command works in a similar fashion to a function call, where the command takes in parameters and returns an output based on the parameters. CPUID reads the values inside the EAX register and sometimes the ECX register and can return values in the registers EAX, EBX, ECX and EDX. This specific EAX and ECX register values are documented inside the official programming manuals written by Intel and AMD.

The CPU model information is collected by executing the following commands.

- CPUID(EAX = 0x80000002)
- CPUID(EAX = 0x80000003)
- CPUID(EAX = 0x80000004)

Each of these commands returns a total of 16 bytes in little-endian format in the EAX, EBX, ECX and EDX registers, respectively.

As an example, we executed these commands on a test machine with an AMD Ryzen 9 5950X CPU installed and received the string “AMD Ryzen 9 5950X 16-Core Processor”. The following are the string values received for each command executed.

Command	**Return Value**	**Little Endian ASCII**	**Big Endian ASCII**
CPUID(EAX = 0x80000002)	EAX = 0x20444D41	“ DMA”	“AMD “
	EBX = 0x657A7952	“ezyR”	“Ryze”
	ECX = 0x2039206E	“ 9 n”	“n 9 “
	EDX = 0x30353935	“0595”	“5950”

CPUID(EAX = 0x80000003)	EAX = 0x36312058	“61 X”	“X 16”
	EBX = 0x726F432D	“roC-“	“-Cor”
	ECX = 0x72502065	“rP e”	“e Pr”
	EDX = 0x7365636F	“seco”	“oces”
CPUID(EAX = 0x80000004)	EAX = 0x20726F73	“ ros”	“sor”
	EBX = 0x20202020	“ “	“ “
	ECX = 0x20202020	“ “	“ “
	EDX = 0x00202020	“ “	“ “

Table 4: CPUID Processor String Example

For collecting the relevant CPU details, such as the number of cores and caching information, we assumed that we were interfacing with the Windows operating system and opted to use the Windows system API function “GetLogicalProcessorInformation” [72]. We also assumed that we are interfacing with versions of Windows that are compatible with this function, which includes Windows XP Service Pack 3, to the current Windows 11 operating system [72]. Collecting specific CPU information, such as caching information, typically uses the CPUID assembly function. Still, the parameters for CPUID to get this information can be inconsistent. It can differ depending on the CPU manufacturer, so this Windows API function simplifies our data collection process.

4.3 Lookup CPU Information

In this step, we consult the database that will contain information about the commercial CPUs available. We decided to use a sqlite3 database to contain all the CPU information for ease of usage of API availability for our experiments.

We primarily built our database of known CPUs from “cpubenchmark.net” [61], which hosts a comprehensive list of commercially available CPUs. We noticed that

the listed CPUs on this website excluded caching information, so we obtained this information from the following sources:

- https://en.wikichip.org
- https://intel.com
- https://www.amd.com/en
- https://www.techpowerup.com

Our database table was built using the following SQL command:

- *CREATE TABLE "cpus" (*
    - *"Name" TEXT,*
    - *"Core" INTEGER,*
    - *"Thread" INTEGER,*
    - *"ProcessorString" TEXT,*
    - *"L1CacheBytes" INTEGER,*
    - *"L2CacheBytes" INTEGER,*
    - *"L3CacheBytes" INTEGER*

  );

All CPUs we added to this table are done so with the following SQL command:

- *INSERT INTO cpus (*
    - *<Name>,*
    - *<Core>,*
    - *<Thread>,*
    - *<ProcessorString>,*
    - *<L1CacheBytes>,*
    - *<L2CacheBytes>.*
    - *<L3CacheBytes>*

  );

An example insert SQL command for an AMD Ryzen 9 5950X processor would be:

- *INSERT INTO cpus (*
    - *“AMD Ryzen 9 5950X”,*
    - *16,*
    - *32,*
    - *“AMD Ryzen 9 5950X 16-Core Processor”,*
    - *1024,*
    - *8192,*
    - *65536*

  );

A search for a particular CPU would be done using the processor string obtained using the CPUID command from the previous step. The SQL query would be:

- *SELECT * FROM cpus WHERE ProcessorString = <Processor String>;*

An example search query looking for an AMD Ryzen 9 5950X processor would be:

- *SELECT * FROM cpus WHERE ProcessorString = “AMD Ryzen 9 5950X 16-Core Processor”;*

The result of the example query would be:

- “AMD Ryzen 9 5950X”, 16, 32, “AMD Ryzen 9 5950X 16-Core Processor”, 1024, 8192, 65536

4.4 Determine if VM is Present

The final step is to determine if a VM is presently based on the information we have collected from the previous steps. We compare the collected CPU information with the CPU information pulled from the database and check for differences. If there are any differences, we assume a virtual machine is present. Likewise, if no CPU matches are made in the previous step, we also assume a virtual machine is present.

The pseudocode of the comparison code we wrote looks something like this:

Algorithm to Determine if Inside VM		
	Input: *Number of Physical Cores, Logical Cores, and Cache Capacities (L1, L2, L3) collected from the current machine.*	
	Output: *Returns True if CPU detail inconsistency detected. Otherwise, returns False.*	
1	***If there is no match from the CPU Database***	
2		*Return True*
3	***If Number of Physical Cores is NOT equal to the Number of Physical Cores from CPU Database***	
4		*Return True*
5	***If Number of Logical Cores is NOT equal to the Number of Logical Cores from CPU Database***	
6		*Return True*
7	***If L1 Cache Capacity is NOT equal to the L1 Cache Capacity from CPU Database***	
8		*Return True*
9	***If L2 Cache Capacity is NOT equal to the L2 Cache Capacity from CPU Database***	
10		*Return True*
11	***If L3 Cache Capacity is NOT equal to the L3 Cache Capacity from CPU Database***	
12		*Return True*
13	*Return False*	
14	***end***	

Table 5: Pseudocode to Determining if in a VM

4.5 Additional Considerations

Virtual machines are usually assigned a fraction of the total available number of cores in the CPU, making the method described in this chapter practical. However,

a user could assign all available CPU cores to a virtual machine in an attempt to defeat our proposed method. We addressed this possibility by conducting a series of experiments detailed in Chapter 5 to locate inconsistencies in CPU details even when all CPU cores were assigned to a VM. Although one may be lead to believe that assigning all CPU cores to a virtualized environment could circumvent our detection approach, our experimental results have demonstrated that many of our test cases showed inconsistencies that could be used to determine the VM's presence.

Chapter 5. Experiments

It should be emphasized that our proposed method of VM detection, detailed in Chapter 4, only works if administrators do not assign all CPU cores to the VM. In cases where all cores are assigned to a VM, the core counts and caching information should be correct, making our method of detection fail. We took this into consideration and decided to do experiments to locate inconsistencies that could be found even if all physical and logical cores of a CPU are assigned to a VM. This chapter intends to detail our testing methodology, the CPUs we used in the experiments and the results of the experiments.

5.1 Testing Methodology

We began by setting the scope of virtualization products we would test for CPU inconsistencies. We decided on the following popular hypervisor packages:

- VMware Workstation (Windows 10 Pro Version 16.1.2)
- Virtualbox (Windows 10 Pro Version 6.1.32)
- HyperV (Windows 10 Pro Version 9.0)
- XEN (XCP-ng Version 8.2.1)
- KVM/QEMU (Ubuntu 20.04 Version 4.2.1)

We would like to acknowledge that VMware ESX is another popular virtualization product in use. However, we chose not to include it in this paper due to extensive technical problems with compatible networking adapters required for the product to function.

We would also like to mention that while it is possible for skilled users to modify the aforementioned hypervisors to defeat our VM detection method, this thesis focuses on these hypervisors and how they perform against our method of detection out of the box. For these tests, we will allow changes to VM settings within reason and will make mention of such settings changes where necessary.

Each of the aforementioned virtualization products would be loaded onto computers with the following CPUs. Please note that all the below CPUs have hardware virtualization acceleration features built in. Modern virtualization

solutions require CPUs to possess this feature, so we were unable to test older CPUs that may not have supported such features.

CPU	**Cores/Threads**	**Cache Capacities (KBytes)**
AMD Ryzen 9 5950X	16 Cores 32 Threads	L1 = 1024 L2 = 8192 L3 = 65536
Intel Core i9-11900H	8 Cores 16 Threads	L1 = 640 L2 = 10240 L3 = 24576
Intel Core i5-6300HQ @ 2.30GHz	4 Core 4 Threads	L1 = 256 L2 = 1024 L3 = 6144
Intel Core i7-4770 @ 3.40GHz	4 Core 8 Threads	L1 = 256.0 L2 = 1024.0 L3 = 8192.0
Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz	6 Cores 6 Threads	L1 = 384 L2 = 1536 L3 = 9216
AMD Ryzen 3 3200U with Radeon Vega Mobile Gfx	2 Cores 4 Threads	L1 = 192 L2 = 1024 L3 = 4096
AMD Ryzen 7 3800X 8-Core Processor	8 Cores 16 Threads	L1 = 512 L2 = 4096 L3 = 32768
Intel Xeon E5-2651 v2 @ 1.80GHz	12 Cores 24 Threads	L1 = 768 L2 = 3072 L3 = 30720

AMD Ryzen Threadripper 3960X	24 Cores 48 Threads	L1 = 1536 L2 = 12288 L3 = 131072
Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz	4 Cores 8 Threads	L1 = 256 L2 = 1024 L3 = 8192
Intel(R) Core(TM) i7 CPU 950 @ 3.07GHz	4 Cores 8 Threads	L1 = 256 L2 = 1024 L3 = 8192

Table 6: CPU Details

Loading this number of virtualization products onto each computer would be greatly time-consuming and inefficient, so we opted instead to use bootable USB drives and SATA SSDs with the virtualization products already installed on them. For desktop/tower computers, we opted to use the SATA SSDs for performance purposes, while the bootable USBs were used in cases where the SSD could not easily be used, such as with laptop computers. We grouped up VMware Workstation and Virtualbox onto the same bootable Windows 10 Pro USB and SATA SSD while making a separate bootable USB drive and SATA SSDs for every other virtualization product. Finally, all bootable media used for the experiments had a Windows 10 Pro VM pre-installed with a software program called “detectvm.exe” as modeled in Figure 12 to collect CPU information and find anomalies.

Interfacing with a new CPU would involve the following steps:

- Booting up either a USB drive or SATA SSD with a preinstalled virtualization product.
- Setting the existing VM on the bootable media to use all physical CPU cores available. For example, assign 8 cores when using a CPU with 8 cores and 16 threads.
- Run “detectvm.exe” and collect results.

- Setting the existing VM on the bootable media to use all logical CPU cores available. For example, assign 16 cores when using a CPU with 8 cores and 16 threads.
- Run “detectvm.exe” and collect results.
- Repeat this process until all virtualization products have been tested on the CPU.

The approach of reverse engineering described in this section ultimately relies on performing experiments on each hypervisor and CPU, observing the results and drawing conclusions from the results. We selected the CPUs with a wide range of core counts, thread counts and cache capacities to draw more well-rounded conclusions about how a particular hypervisor may virtualize a CPU for the virtual machine. Considering that our method intends to test only for physical core count, logical core count and cache capacities, it is possible that CPUs exist that will cause a hypervisor to virtualize the CPU differently from our observed results. The alternative approach would be to debug each of the hypervisors and observe the CPU instructions that make up the hypervisors. This would ascertain exactly how the hypervisors virtualize the CPU details exposed to the virtual machine but would be far more laborious – especially for closed source hypervisors such as VMware Workstations. Furthermore, this method may be sensitive to hypervisor version updates.

The next sections will show the results and notes we received during our experiments.

5.2 VMware Workstation

The results of running our experiments on VMware Workstation were as follows. The incorrect values are noted.

CPU	**All Physical Cores**	**All Logical Cores**
AMD Ryzen 9 5950X	16 Cores 16 Threads (Incorrect) L1 = 1024 Kbytes	32 Cores (Incorrect) 32 Threads

	L2 = 8192 Kbytes L3 = 65536 Kbytes	L1 = 2048 Kbytes (Incorrect) L2 = 16384 Kbytes (Incorrect) L3 = 131072 Kbytes (Incorrect)
Intel Core i9-11900H	8 Cores 8 Threads (Incorrect) L1 = 640 Kbytes L2 = 10240 Kbytes L3 = 24576 Kbytes	16 Cores (Incorrect) 16 Threads L1 = 1280 Kbytes (Incorrect) L2 = 20480 Kbytes (Incorrect) L3 = 49152 Kbytes (Incorrect)
Intel Core i5-6300HQ @ 2.30GHz	4 Core 4 Threads L1 = 256 Kbytes L2 = 1024 Kbytes L3 = 6144 Kbytes	
Intel Core i7-4770 @ 3.40GHz	4 Core 4 Threads (Incorrect) L1 = 256 Kbytes L2 = 1024 Kbytes L3 = 8192 Kbytes	8 Core (Incorrect) 8 Threads L1 = 512 Kbytes (Incorrect) L2 = 2048 Kbytes (Incorrect) L3 = 8192 Kbytes
Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz	6 Cores 6 Threads L1 = 384 Kbytes L2 = 1536 Kbytes	

	L3 = 9216 Kbytes	
AMD Ryzen 3 3200U with Radeon Vega Mobile Gfx	2 Cores 2 Threads (Incorrect) L1 = 192 Kbytes L2 = 1024 Kbytes L3 = 4096 Kbytes	4 Cores (Incorrect) 4 Threads L1 = 384 Kbytes (Incorrect) L2 = 2048 Kbytes (Incorrect) L3 = 4096 Kbytes
AMD Ryzen 7 3800X 8-Core Processor	8 Cores 8 Threads (Incorrect) L1 = 512 Kbytes L2 = 4096 Kbytes L3 = 32768 Kbytes	16 Cores (Incorrect) 16 Threads L1 = 1024 Kbytes (Incorrect) L2 = 8092 Kbytes (Incorrect) L3 = 32768 Kbytes
Intel Xeon E5-2651 v2 @ 1.80GHz	12 Cores 12 Threads (Incorrect) L1 = 768 Kbytes L2 = 3072 Kbytes L3 = 30720 Kbytes	24 Cores (Incorrect) 24 Threads L1 = 1536 Kbytes (Incorrect) L2 = 6144 Kbytes (Incorrect) L3 = 61440 Kbytes (Incorrect)
AMD Ryzen Threadripper 3960X	24 Cores 24 Threads (Incorrect) L1 = 1536 Kbytes L2 = 12288 Kbytes L3 = 131072 Kbytes	48 Cores (Incorrect) 48 Threads L1 = 3072 Kbytes (Incorrect) L2 = 24576 Kbytes (Incorrect)

		L3 = 262144 Kbytes (Incorrect)
Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz	4 Cores 4 Threads (Incorrect) L1 = 256 Kbytes L2 = 1024 Kbytes L3 = 8192 Kbytes	8 Cores (Incorrect) 8 Threads L1 = 512 Kbytes (Incorrect) L2 = 2048 Kbytes (Incorrect) L3 = 16384 Kbytes (Incorrect)
Intel(R) Core(TM) i7 CPU 950 @ 3.07GHz	4 Cores 4 Threads (Incorrect) L1 = 256 Kbytes L2 = 1024 Kbytes L3 = 8192 Kbytes	8 Cores (Incorrect) 8 Threads L1 = 512 Kbytes (Incorrect) L2 = 2048 Kbytes (Incorrect) L3 = 16384 Kbytes (Incorrect)

Table 7: VMware Workstation Results

VMware Workstation appeared to have great difficulty with distinguishing between the physical and logical cores of the CPU. When attempting to remedy this problem by assigning all logical cores to the VM, we found that the caching information was double what it should be, and the number of physical cores became incorrect. We observed that the CPUs with an equal number of physical and logical cores were immune to our detection method, as the number of physical cores, logical cores, and caching information were all correct when all physical cores were assigned to the VM.

5.3 VirtualBox

The results of running our experiments on VirtualBox were as follows. The incorrect values are noted.

CPU	All Physical Cores	All Logical Cores
AMD Ryzen 9 5950X	16 Cores 16 Threads (Incorrect) L1 = 1024 Kbytes L2 = 8192 Kbytes L3 = 1048576 Kbytes (Incorrect)	32 Cores (Incorrect) 32 Threads L1 = 2048 Kbytes (Incorrect) L2 = 16384 Kbytes (Incorrect) L3 = 2097152 Kbytes (Incorrect)
Intel Core i9-11900H	8 Cores 8 Threads (Incorrect) L1 = 640 Kbytes L2 = 10240 Kbytes L3 = 196608 Kbytes (Incorrect)	16 Cores (Incorrect) 16 Threads L1 = 1280 Kbytes (Incorrect) L2 = 20480 Kbytes (Incorrect) L3 = 393216 Kbytes (Incorrect)
Intel Core i5-6300HQ @ 2.30GHz	4 Core 4 Threads L1 = 256 Kbytes L2 = 1024 Kbytes L3 = 24576 Kbytes (Incorrect)	
Intel Core i7-4770 @ 3.40GHz	4 Core 4 Threads (Incorrect) L1 = 256 Kbytes L2 = 1024 Kbytes L3 = 32768 Kbytes (Incorrect)	8 Core (Incorrect) 8 Threads L1 = 512 Kbytes (Incorrect) L2 = 2048 Kbytes (Incorrect)

		L3 = 65536 Kbytes (Incorrect)
Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz	6 Cores 6 Threads L1 = 384 Kbytes L2 = 1536 Kbytes L3 = 55296 Kbytes (Incorrect)	
AMD Ryzen 3 3200U with Radeon Vega Mobile Gfx	2 Cores 2 Threads (Incorrect) L1 = 192 Kbytes L2 = 1024 Kbytes L3 = 8192 Kbytes (Incorrect)	4 Cores (Incorrect) 4 Threads L1 = 384 Kbytes (Incorrect) L2 = 2048 Kbytes (Incorrect) L3 = 16384 Kbytes (Incorrect)
AMD Ryzen 7 3800X 8-Core Processor	8 Cores 8 Threads (Incorrect) L1 = 512 Kbytes L2 = 4096 Kbytes L3 = 262144 Kbytes (Incorrect)	16 Cores (Incorrect) 16 Threads L1 = 1024 Kbytes (Incorrect) L2 = 8092 Kbytes (Incorrect) L3 = 524288 Kbytes (Incorrect)
Intel Xeon E5-2651 v2 @ 1.80GHz	12 Cores 12 Threads (Incorrect) L1 = 768 Kbytes L2 = 3072 Kbytes L3 = 368640 Kbytes (Incorrect)	24 Cores (Incorrect) 24 Threads L1 = 1536 Kbytes (Incorrect) L2 = 6144 Kbytes (Incorrect)

		L3 = 737280 Kbytes (Incorrect)
AMD Ryzen Threadripper 3960X	24 Cores 24 Threads (Incorrect) L1 = 1536 Kbytes L2 = 12288 Kbytes L3 = 3145728 Kbytes (Incorrect)	48 Cores (Incorrect) 48 Threads L1 = 3072 Kbytes (Incorrect) L2 = 24576 Kbytes (Incorrect) L3 = 6291456 Kbytes (Incorrect)
Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz	4 Cores 4 Threads (Incorrect) L1 = 256 Kbytes L2 = 1024 Kbytes L3 = 32768 Kbytes (Incorrect)	8 Cores (Incorrect) 8 Threads L1 = 512 Kbytes (Incorrect) L2 = 2048 Kbytes (Incorrect) L3 = 65536 Kbytes (Incorrect)
Intel(R) Core(TM) i7 CPU 950 @ 3.07GHz	4 Cores 4 Threads (Incorrect) L1 = 256 Kbytes L2 = 1024 Kbytes L3 = 32768 Kbytes (Incorrect)	8 Cores (Incorrect) 8 Threads L1 = 512 Kbytes (Incorrect) L2 = 2048 Kbytes (Incorrect) L3 = 65536 Kbytes (Incorrect)

Table 8: VirtualBox Results

Virtualbox, much like Workstation, experienced great difficulty distinguishing between physical and logical CPU cores. As such, the number of physical cores was always equal to the number of logical cores detected by the VM in our tests. The caching capacities were also especially troublesome with Virtualbox – every

CPU we tested had a problem with core counts, caching information, or both. We observed that Virtualbox seemed to follow the following set of rules when calculating cache capacities for VMs:

- L1 Cache = (Single L1 Cache Capacity) * (Number of cores assigned to VM) * 2
- L2 Cache = (Single L2 Cache Capacity) * (Number of cores assigned to VM)
- L3 Cache = (Single L3 Cache Capacity) * (Number of cores assigned to VM)

The most egregious of the above rules is the calculation of the L3 capacity. Traditionally, the L3 cache of a CPU is a cache that is shared among most, if not all, of the cores. It appears that Virtualbox attempts to assign each core its own L3 cache, making the calculated L3 cache capacity far greater than what it is actually supposed to be. For instance, with the AMD Ryzen 9 5950X CPU, the L3 cache is to be 65536 Kbytes. We assigned 16 cores to a VM and the L3 cache was detected as 1048576 Kbytes, which is 65536 multiplied by 16.

5.4 HyperV

The results of running our experiments on HyperV were as follows. The incorrect values are noted.

CPU	**All Physical Cores**	**All Logical Cores**
AMD Ryzen 9 5950X	8 Cores (Incorrect) 16 Threads (Incorrect) L1 = 512 Kbytes (Incorrect) L2 = 4096 Kbytes (Incorrect) L3 = 32768 Kbytes (Incorrect)	16 Cores 32 Threads L1 = 1024 Kbytes L2 = 8192 Kbytes L3 = 32768 Kbytes (Incorrect)
Intel Core i9-11900H	4 Cores (Incorrect)	8 Cores

	8 Threads (Incorrect) L1 = 320 Kbytes (Incorrect) L2 = 5120 Kbytes (Incorrect) L3 = 24576 Kbytes	16 Threads L1 = 640 Kbytes L2 = 10240 Kbytes L3 = 24576 Kbytes
Intel Core i5-6300HQ @ 2.30GHz	4 Core 4 Threads L1 = 256 Kbytes L2 = 1024 Kbytes L3 = 6144 Kbytes	
Intel Core i7-4770 @ 3.40GHz	2 Core (Incorrect) 4 Threads (Incorrect) L1 = 128 Kbytes (Incorrect) L2 = 512 Kbytes (Incorrect) L3 = 8192 Kbytes	4 Core 8 Threads L1 = 256 Kbytes L2 = 1024 Kbytes L3 = 8192 Kbytes
Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz	6 Cores 6 Threads L1 = 384 Kbytes L2 = 1536 Kbytes L3 = 9216 Kbytes	
AMD Ryzen 3 3200U with Radeon Vega Mobile Gfx	1 Cores (Incorrect) 2 Threads (Incorrect) L1 = 96 Kbytes (Incorrect) L2 = 512 Kbytes (Incorrect) L3 = 4096 Kbytes	2 Cores 4 Threads L1 = 192 Kbytes L2 = 1024 Kbytes L3 = 4096 Kbytes

AMD Ryzen 7 3800X 8-Core Processor	4 Cores (Incorrect) 8 Threads (Incorrect) L1 = 256 Kbytes (Incorrect) L2 = 2048 Kbytes (Incorrect) L3 = 32768 Kbytes	8 Cores 16 Threads L1 = 512 Kbytes L2 = 4096 Kbytes L3 = 32768 Kbytes
Intel Xeon E5-2651 v2 @ 1.80GHz	6 Cores (Incorrect) 12 Threads (Incorrect) L1 = 384 Kbytes (Incorrect) L2 = 1536 Kbytes (Incorrect) L3 = 30720 Kbytes	12 Cores 24 Threads L1 = 768 Kbytes L2 = 3072 Kbytes L3 = 30720 Kbytes
AMD Ryzen Threadripper 3960X	12 Cores (Incorrect) 24 Threads (Incorrect) L1 = 768 Kbytes (Incorrect) L2 = 6144 Kbytes (Incorrect) L3 = 16384 Kbytes (Incorrect)	24 Cores 48 Threads L1 = 1536 Kbytes L2 = 12288 Kbytes L3 = 16384 Kbytes (Incorrect)
Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz	2 Cores (Incorrect) 4 Threads (Incorrect) L1 = 128 Kbytes (Incorrect) L2 = 512 Kbytes (Incorrect) L3 = 8192 Kbytes	4 Cores 8 Threads L1 = 256 Kbytes L2 = 1024 Kbytes L3 = 8192 Kbytes

Intel(R) Core(TM) i7 CPU 950 @ 3.07GHz	2 Cores (Incorrect) 4 Threads (Incorrect) L1 = 128 Kbytes (Incorrect) L2 = 512 Kbytes (Incorrect) L3 = 8192 Kbytes	4 Cores 8 Threads L1 = 256 Kbytes L2 = 1024 Kbytes L3 = 8192 Kbytes

Table 9: HyperV Results

HyperV was successfully able to distinguish how many threads are present per core, as evidenced by trying HyperV on both CPUs that had an equal number of physical cores and threads, and CPUs that did not have an equal number of physical cores and threads. However, there were some problems with detecting the correct amount of L3 cache for high-core count CPUs, such as the AMD Ryzen 9 5950X and the AMD Ryzen Threadripper 3960X. Most CPUs have only 1 L3 cache shared among the cores, but higher core count CPUs, such as the aforementioned CPUs, possess multiple L3 caches. HyperV makes the assumption that there will only be 1 L3 cache exposed to the VM, making the total L3 cache capacity far less than expected. For instance, the AMD Ryzen 9 5950X has 2 L3 caches, each with a 32768 Kbyte capacity, totaling 65536 Kbytes. HyperV only detects a single L3 cache, leading to the incorrect total of 32768 Kbytes. This behavior was observed when we assigned both all the physical cores to the VM and all logical cores to the VM.

We also observed that when assigning all logical cores to a VM, as long as the CPU had only a single L3 cache shared among its cores, the VM will possess the correct core count, thread count and caching capacity, thus defeating our method of VM detection.

5.5 XEN

For this experiment, we used the XEN hypervisor housed in the XCP-ng virtualization platform. We used XCP-ng Center Version 20.04.01.33 to install and control virtual machines remotely.

The results of running our experiments on XEN were as follows. The incorrect values are noted.

CPU	All Physical Cores	All Logical Cores
AMD Ryzen 9 5950X	16 Cores 16 Threads (Incorrect) L1 = 1024 Kbytes L2 = 8192 Kbytes L3 = 1048576 Kbytes (Incorrect)	32 Cores (Incorrect) 32 Threads L1 = 2048 Kbytes (Incorrect) L2 = 16384 Kbytes (Incorrect) L3 = 2097152 Kbytes (Incorrect)
Intel Core i9-11900H	8 Cores 8 Threads (Incorrect) L1 = 640 Kbytes L2 = 10240 Kbytes L3 = 196608 Kbytes (Incorrect)	16 Cores (Incorrect) 16 Threads L1 = 1280 Kbytes (Incorrect) L2 = 20480 Kbytes (Incorrect) L3 = 393216 Kbytes (Incorrect)
Intel Core i5-6300HQ @ 2.30GHz	4 Core 4 Threads L1 = 256 Kbytes L2 = 1024 Kbytes L3 = 24576 Kbytes (Incorrect)	
Intel Core i7-4770 @ 3.40GHz	4 Core 4 Threads (Incorrect) L1 = 256 Kbytes L2 = 1024 Kbytes	8 Core (Incorrect) 8 Threads L1 = 512 Kbytes (Incorrect)

	L3 = 32768 Kbytes (Incorrect)	L2 = 2048 Kbytes (Incorrect) L3 = 65536 Kbytes (Incorrect)
Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz	6 Cores 6 Threads L1 = 384 Kbytes L2 = 1536 Kbytes L3 = 55296 Kbytes (Incorrect)	
AMD Ryzen 3 3200U with Radeon Vega Mobile Gfx	2 Cores 2 Threads (Incorrect) L1 = 192 Kbytes L2 = 1024 Kbytes L3 = 8192 Kbytes (Incorrect)	4 Cores (Incorrect) 4 Threads L1 = 384 Kbytes (Incorrect) L2 = 2048 Kbytes (Incorrect) L3 = 16384 Kbytes (Incorrect)
AMD Ryzen 7 3800X 8-Core Processor	8 Cores 8 Threads (Incorrect) L1 = 512 Kbytes L2 = 4096 Kbytes L3 = 262144 Kbytes (Incorrect)	16 Cores (Incorrect) 16 Threads L1 = 1024 Kbytes (Incorrect) L2 = 8092 Kbytes (Incorrect) L3 = 524288 Kbytes (Incorrect)
Intel Xeon E5-2651 v2 @ 1.80GHz	12 Cores 12 Threads (Incorrect) L1 = 768 Kbytes L2 = 3072 Kbytes	24 Cores (Incorrect) 24 Threads L1 = 1536 Kbytes (Incorrect)

	L3 = 368640 Kbytes (Incorrect)	L2 = 6144 Kbytes (Incorrect) L3 = 737280 Kbytes (Incorrect)
AMD Ryzen Threadripper 3960X	24 Cores 24 Threads (Incorrect) L1 = 1536 Kbytes L2 = 12288 Kbytes L3 = 3145728 Kbytes (Incorrect)	48 Cores (Incorrect) 48 Threads L1 = 3072 Kbytes (Incorrect) L2 = 24576 Kbytes (Incorrect) L3 = 6291456 Kbytes (Incorrect)
Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz	4 Cores 4 Threads (Incorrect) L1 = 256 Kbytes L2 = 1024 Kbytes L3 = 32768 Kbytes (Incorrect)	8 Cores (Incorrect) 8 Threads L1 = 512 Kbytes (Incorrect) L2 = 2048 Kbytes (Incorrect) L3 = 65536 Kbytes (Incorrect)
Intel(R) Core(TM) i7 CPU 950 @ 3.07GHz	4 Cores 4 Threads (Incorrect) L1 = 256 Kbytes L2 = 1024 Kbytes L3 = 32768 Kbytes (Incorrect)	8 Cores (Incorrect) 8 Threads L1 = 512 Kbytes (Incorrect) L2 = 2048 Kbytes (Incorrect) L3 = 65536 Kbytes (Incorrect)

Table 10: Xen Results

The results of this experiment were identical to the results obtained from VirtualBox. Both the core counts and caching information was problematic, to the

point where all CPUs tested were found with inconsistencies. Similar to Virtualbox, the L3 cache is multiplied by the number of cores assigned to the VM. As such, the L3 cache gives away the presence of the VM.

5.6 KVM/QEMU

KVM/QEMU possesses a setting that allows users to set the following pieces of information exposed to the VM:

- Number of cores
- Number of threads per core
- CPUID processor string
- Caching information

Due to QEMU's ability to emulate hardware instead of virtualizing, we were able to set the number of cores to a greater number than what was physically available in the CPU. Furthermore, we found that upon initial installation, a bogus CPUID processor string would be automatically assigned to the VM based on the number of physical cores and CPU features that exist within the CPU. For instance, on our testbench running an AMD Ryzen 9 5950X CPU with 16 physical cores and 32 threads, KVM/QEMU assigned the string "AMD EPYC-Milan Processor". Not only is this information incorrect, but this string is also not a valid CPUID processor string for any AMD EPYC processor available. A valid processor string for an AMD EPYC processor would include the model number and the number of cores in the CPU, such as "AMD EPYC 7763 64-Core Processor". We found that KVM/QEMU uses an XML file to control the CPU information exposed to the VM. We included the snippet of XML code seen in Figure 13 for our initial experiment with the AMD Ryzen 9 5950X CPU to best match the CPU's specifications.

```
<cpu mode="host-passthrough">
  <topology sockets="1" cores="16" threads="2"/>
  <cache mode="passthrough"/>
</cpu>
```

Figure 13: XML Code to bypass VM detection method for KVM/QEMU

The above snippet works as follows:

- mode=”host-passthrough”
    - Copies the CPUID processor string from the host CPU to the VM. This allows for a legitimate CPUID processor string to be presented to our CPU database. In the case of the above snippet of XML code, when interfacing with an AMD Ryzen 9 5950X CPU, the string “AMD Ryzen 9 5950X 16-Core Processor” was detected by the VM.
- <topology sockets=”1” cores=”16” threads=”2”/>
    - Sets the VM’s CPU sockets, core count, and the number of threads per core. This particular example is tailored to the AMD Ryzen 9 5950X, which possesses 16 cores and 32 threads. When running this snippet of code, the VM correctly detected 16 physical cores and 32 threads. When we used this code on different CPUs, we tailored the number of cores and threads to the CPU we were interfacing with.
- cache mode=”passthrough”
    - Copies the values of each cache level from the host CPU to the VM. This allows for a correct cache capacity total detected inside the VM.

The XML code snippet described above was able to defeat our method of VM detection as it successfully exposed the correct physical core count, thread count and caching information to the VM. Testing this on the other CPUs, we found that they also were all able to defeat our method of VM detection.

5.7 Summary

This section intends to summarize the results of all the experiments in the previous subsections. The below tables will display the CPUs tested according to the hypervisor and the number of cores allocated to the VM. CPUs listed as “pass” were able to bypass our method of VM detection, while “fail” indicates an inconsistency was detected.

All Physical Cores Allocated

CPU	VMware Workstation	Virtualbox	HyperV	Xen	KVM/QEMU
AMD Ryzen 9 5950X	Fail	Fail	Fail	Fail	Pass
Intel Core i9-11900H	Fail	Fail	Fail	Fail	Pass
Intel Core i5-6300HQ @ 2.30GHz	Pass	Fail	Pass	Fail	Pass
Intel Core i7-4770 @ 3.40GHz	Fail	Fail	Fail	Fail	Pass
Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz	Pass	Fail	Pass	Fail	Pass
AMD Ryzen 3 3200U with Radeon Vega Mobile Gfx	Fail	Fail	Fail	Fail	Pass
AMD Ryzen 7 3800X 8-Core Processor	Fail	Fail	Fail	Fail	Pass
Intel Xeon E5-2651 v2 @ 1.80GHz	Fail	Fail	Fail	Fail	Pass

AMD Ryzen Threadripper 3960X	Fail	Fail	Fail	Fail	Pass
Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz	Fail	Fail	Fail	Fail	Pass
Intel(R) Core(TM) i7 CPU 950 @ 3.07GHz	Fail	Fail	Fail	Fail	Pass

Table 11: Summary of Results when all Physical Cores are assigned to VM

All Logical Cores Allocated					
CPU	**VMware Workstation**	**Virtualbox**	**HyperV**	**Xen**	**KVM/QEMU**
AMD Ryzen 9 5950X	Fail	Fail	Fail	Fail	Pass
Intel Core i9-11900H	Fail	Fail	Pass	Fail	Pass
Intel Core i5-6300HQ @ 2.30GHz	Pass	Fail	Pass	Fail	Pass
Intel Core i7-4770 @ 3.40GHz	Fail	Fail	Pass	Fail	Pass
Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz	Pass	Fail	Pass	Fail	Pass

AMD Ryzen 3 3200U with Radeon Vega Mobile Gfx	Fail	Fail	Pass	Fail	Pass
AMD Ryzen 7 3800X 8-Core Processor	Fail	Fail	Pass	Fail	Pass
Intel Xeon E5-2651 v2 @ 1.80GHz	Fail	Fail	Pass	Fail	Pass
AMD Ryzen Threadripper 3960X	Fail	Fail	Fail	Fail	Pass
Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz	Fail	Fail	Pass	Fail	Pass
Intel(R) Core(TM) i7 CPU 950 @ 3.07GHz	Fail	Fail	Pass	Fail	Pass

Table 12: Summary of Results when all Logical Cores are assigned to VM

Chapter 6. Analysis

This section aims to provide further information on our findings, detailed in Chapter 5. We intend to provide insight into the significance of the results along with how such findings could be used by those looking to further “harden” their virtual machines against detection. We also wish to talk more about the limitations of our method of virtual machine detection.

6.1 Significance

Before we talk about the significance of our research, we’d like to begin by reiterating the details and reasoning of our experiments in the first place. Our novel method of virtual machine detection involves checking an environment of the CPU details, including CPU core numbers and caching information exposed and comparing them with a database of known information about the CPU reported in use by the machine. In the context of virtual machines, users can assign a set number of CPU cores to virtual machines. If this number of assigned CPU cores does not match the maximum number of cores physically built into the CPU, we can deduce a virtual machine is present. For instance, if the environment we are interfacing with reports to be using a CPU we know possesses 8 physical cores and 16 logical cores yet reports less than the aforementioned figures, we can deduce virtualization is present here. We acknowledge the fact that users may purposefully assign all CPU cores to a virtual machine in hopes of evading our proposed method of detection. As a result, we conducted the experiments detailed in Chapter 5 to see if any CPU detail inconsistencies could be detected even when all physical and logical CPU cores are assigned to a virtual machine. Many of our results showed that despite assigning all physical and logical CPU cores to virtual machines, we were still able to find inconsistencies that could be used to detect the presence of the virtualized environment. Only a handful of our tests were able to bypass our method of detection, and we intend to summarize these findings below.

- VMware Workstation

    - Virtual machines making use of Workstation will detect an equal number of physical and logical CPU cores. That said, Workstation will assume there is only one logical core per every physical core, which is not true for all CPUs. As a result, only CPUs with one logical core per physical core were able to defeat our method of detection when assigned all physical cores. As an example, the Intel Core i5-6300HQ with 4 physical cores and 4 logical cores was able to bypass our method of detection when all cores were assigned to the virtual machine.
- VirtualBox
    - Virtual machines making use of VirtualBox will detect an equal number of physical and logical CPU cores. In addition to this, the Level 3 cache capacity is multiplied by the number of CPU cores assigned to the virtual machine, giving erroneous results for every CPU we tested. We were, unfortunately, unable to create a scenario where the VirtualBox virtual machine had the correct number of CPU cores and caching capacities. Unless modifications are made to the VirtualBox hypervisor, VirtualBox is vulnerable to this method of virtual machine detection regardless of the CPU used or the number of CPU cores assigned.
- HyperV
    - Virtual machines making use of HyperV were correctly able to deduce the number of physical and logical cores, unlike VMware Workstation and VirtualBox. All CPUs we experimented with could bypass our method of detection except for CPUs with multiple Level 3 caches built into them, which included the AMD Ryzen 9 5950X and the AMD Ryzen Threadripper 3960X. HyperV only detects a single Level 3 cache inside the virtual machine, making HyperV unreliable for usage in higher-end CPUs that possess more than one Level 3 cache.
- XEN

    - Virtual machines making use of XEN will detect an equal number of physical and logical CPU cores, similar to VMware Workstation and VirtualBox. The results for XEN were identical to that of VirtualBox – the Level 3 cache capacity is multiplied by the number of CPU cores assigned to the virtual machine. Also, like VirtualBox, no test cases possessed the correct CPU information to what we had recorded in our database of CPU details.
- KVM/QEMU
    - All virtual machines making use of KVM/QEMU were able to bypass our method of detection. Thanks to the emulation layer provided by QEMU, we can adjust in the settings files how the CPU will be presented to the virtual machine, allowing us to tailor the CPU details to the CPU we were testing at the time.

In summary, the significance of our research is that many virtualization products that may be used in the context of malware analysis can be detected using our method, even if all physical and logical cores are assigned to them. We have shown that while, in many cases, a virtual machine can be detected in such scenarios, certain CPUs can bypass our proposed method depending on the virtualization product in use. The one large exception to our tests was KVM/QEMU, which makes use of emulation to allow users to set various CPU details according to their preferences. Until a more permanent solution can be found, malware analysts may wish to embrace the usage of KVM/QEMU to avoid problems with malware that may use our method of virtual machine detection.

6.2 Limitations

This section aims to detail the limitations of this method of virtual machine detection. While we did talk about some limitations regarding some CPUs in use during our tests in section 6.1, we are looking beyond simply the CPUs, and are focusing on other aspects that could also defeat our method of detection.

6.2.1 BIOS Disable CPU Cores

Interfacing with a testbench machine running an AMD Ryzen 9 5950X CPU with a Gigabyte X570 Aorus Master motherboard, we found that it is possible to disable individual cores in the CPU inside the motherboard's BIOS. Disabling cores will always give an incorrect core count and will make our method of detection always think it is detecting a VM, even when no VM is present. By disabling cores in the CPU, overall CPU performance will be reduced, making it counterproductive for users to persistently turn off cores. While this is a limitation to our method of VM detection, we believe this limitation likely will have a minimal effect due to the reduction in CPU performance for the user.

6.2.2 Updated CPU Database

For an attacker to use this method of detection, he will consistently need to maintain the CPU database used. Processor manufacturers such as Intel and AMD frequently release new CPU models throughout the year, making this potentially a grueling task. If an attacker were to incorporate this method of VM detection in malware, the malware would need to have some way of updating its CPU database, such as having a centralized server it would connect to regularly for updates. This could lead to trouble for the attacker, as sending internet requests can be a means of detection by security systems such as AntiVirus programs.

6.2.3 Trusted Sources of CPU Information

Due to the vital role CPU information plays in our proposed method of VM detection, it is important to use trustworthy sources to keep the CPU database updated. In this thesis, we made use of the web sources: cpubenchmark.net, wikichip.org, amd.com, intel.com and techpowerup.com as our sources for CPU information. While we consider the official AMD and Intel websites to be trustworthy sources, it is possible for information sourced from the other web sources to be incorrect, which could pose a threat to this method of detection. If incorrect information were to be collected and added to the CPU database, our method of detection would erroneously think VMs are present even when they are not. Developers intending to implement our proposed method of VM detection will need

to find a source of information that is both trustworthy and can be easily used to update the CPU database.

6.2.4 CPUID Spoofing

Our method is centralized around the CPUID command to identify what CPU we are interfacing with. If a user were to spoof the value returned by the CPUID command, he would be able to fool attackers into thinking they are interfacing with an entirely different CPU.

VMware Workstation, as an example, provides an interface to spoof various CPUID return values based on the original EAX input. Each VM possesses a ".vmx" file, which contains various settings of the VM. This file can be modified to spoof CPUID return values. The two primary lines used for spoofing are as follows.

- **monitor_control.enable_fullcpuid = "TRUE"**
- **cpuid.X.Y = Z** where X is the original EAX input in hexadecimal, Y is EAX, EBX, ECX or EDX, and Z is the binary representation of the value that is to be exposed to the VM.

To spoof the processor string, we will be interacting with 0x80000002, 0x80000003, and 0x80000004. As seen in section 4.2, the values returned by these CPUID calls will be in little endian format, so we must keep this in mind when spoofing these return values. If we wanted to spoof the processor string "AMD Ryzen 7 3800X 8-Core Processor" for example, we want the CPUID commands to return the following.

Command	**Return Value**	**Little Endian ASCII**	**Big Endian ASCII**
CPUID(EAX = 0x80000002)	EAX = 0x20444D41	" DMA"	"AMD "
	EBX = 0x657A7952	"ezyR"	"Ryze"
	ECX = 0x2037206e	" 7 n"	"n 7 "
	EDX = 0x30303833	"0083"	"3800"

CPUID(EAX = 0x80000003)	EAX = 0x2d382058	“-8 X”	“X 8-”
	EBX = 0x65726F43	“eroC“	“Core”
	ECX = 0x6f725020	“orP ”	“ Pro”
	EDX = 0x73736563	“ssec”	“cess”
CPUID(EAX = 0x80000004)	EAX = 0x2020726F	“ ro”	“or ”
	EBX = 0x20202020	“ “	“ “
	ECX = 0x20202020	“ “	“ “
	EDX = 0x00202020	“ “	“ “

Table 13: CPUID Processor String Output for AMD Ryzen 7 3800X Processor

The above hexadecimal values are to be converted into binary values which will be inserted into the “.vmx” file as follows.

- **cpuid.80000002.eax = "00100000010001000100110101000001"**
- **cpuid.80000002.ebx = "01100101011110100111100101010010"**
- **cpuid.80000002.ecx = "00100000001101110010000001101110"**
- **cpuid.80000002.edx = "00110000001100000011100000110011"**
- **cpuid.80000003.eax = "00101101001110000010000001011000"**
- **cpuid.80000003.ebx = "01100101011100100110111101000011"**
- **cpuid.80000003.ecx = "01101111011100100101000000100000"**
- **cpuid.80000003.edx = "01110011011100110110010101100011"**
- **cpuid.80000004.eax = "00100000001000000111001001101111"**
- **cpuid.80000004.ebx = "00100000001000000010000000100000"**
- **cpuid.80000004.ecx = "00100000001000000010000000100000"**
- **cpuid.80000004.edx = "00000000001000000010000000100000”**

As seen in Figure 14, CPU-Z, a diagnostic tool used for CPUs, shows the CPUID processor string that we have now spoofed. Please note that this VM is natively

running on an AMD Ryzen 9 5950X processor and that the VM is running under VMware Workstation. It should also be noted that the CPU we are attempting to spoof in this case has 8 physical cores and 16 logical cores. Only 8 logical cores are being detected by the VM, meaning that more work will need to be done to completely defeat our method of VM detection. Additionally, CPUID processor string spoofing will not change caching information detected by the VM.

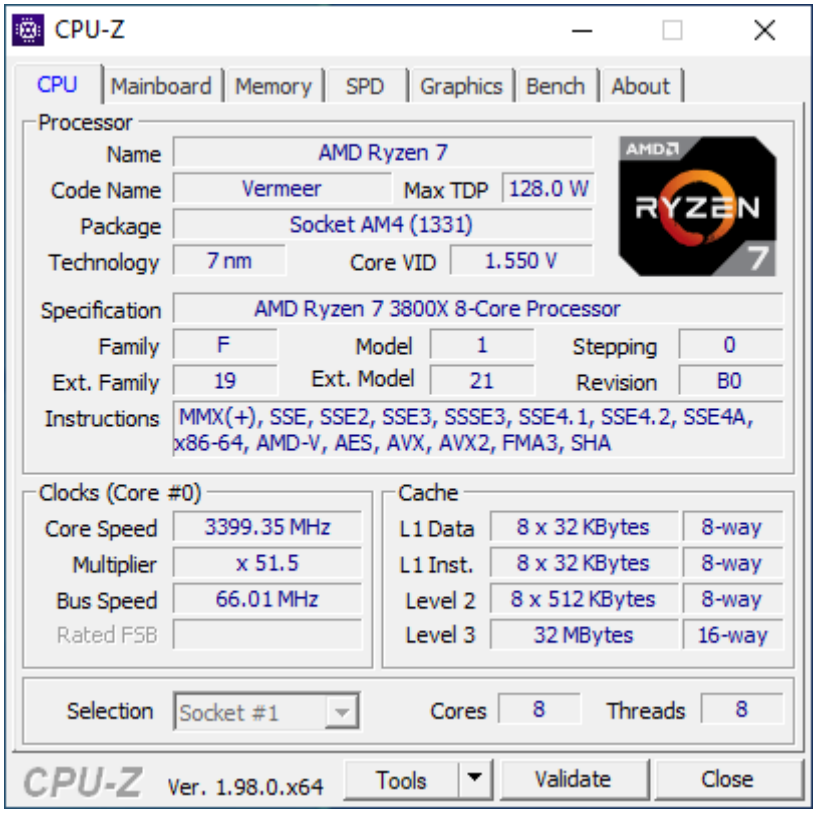


Figure 14: CPUID Processor String Spoofing

For an attacker to defeat our proposed method using this limitation, they would need to spoof the CPUID processor string to a different CPU's processor string and assign the VM the correct number of CPU cores and cache capacities according to the spoofed processor string. However, to the best of our knowledge, a user cannot explicitly set the cache capacities that will be exposed to the virtual machine except for the KVM/QEMU virtualization platform. The hypervisor of the virtualization products controls the cache capacities exposed to the VMs and only allows users to change the number of CPU cores provided to the VMs. With this in

mind, an attacker would need to know how a virtualization product will present the CPU details to the VM ahead of time and work around these details when spoofing the CPUID processor string. For instance, as outlined in section 5.2, we know that VMware Workstation will assume that there is only 1 logical CPU core per physical logical core provided to the VM. An attacker would need to do the following in this case:

1. Find a CPU model that has the same number of physical and logical CPU cores.
2. Use a machine with a physical CPU that has a greater or equal number of CPU cores as the spoofed CPU model.
3. Use a machine with a physical CPU that has the correct L1, L2 and L3cache capacities that will match the total cache capacities of the CPU they intend to spoof when all cores are assigned to the VM.

It is to the best of our knowledge that no commercial desktop or laptop CPU exists currently that satisfies all these 3 conditions. Furthermore, the other virtualization products investigated also possessed their own problems with CPU details, such as over-representing level 3 cache capacities with VirtualBox and Xen.

6.2.5 API Hooking

API hooking is a technique commonly used by malware [73] to intercept and modify the return values of API calls made by other applications. The concept of API hooking could be thought of in a similar way to a classic man-in-the-middle attack. A client sends a request to a destination only for another party to intercept the request on its way to the destination, and spoof or otherwise modify the request or return values from the destination. In the case of API hooking, however, when an application attempts to call a particular API function, the API “hook” will be called instead of the intended API function, allowing someone to spoof the return values from this “API.”

Within the context of this thesis, we depend on the “GetLogicalProcessorInformation” Windows API function to collect information about the number of physical CPU cores, logical CPU cores and CPU caching

capacities. Assuming a malware program were to also make use of this API function to determine CPU details, and the malware program made use of the exact same CPU details we've focused on in this paper, it is possible to "harden" a VM by spoofing the CPU details via API hooking. A visualization of how a possible application of API hooking in this scenario can be seen in Figure 15.

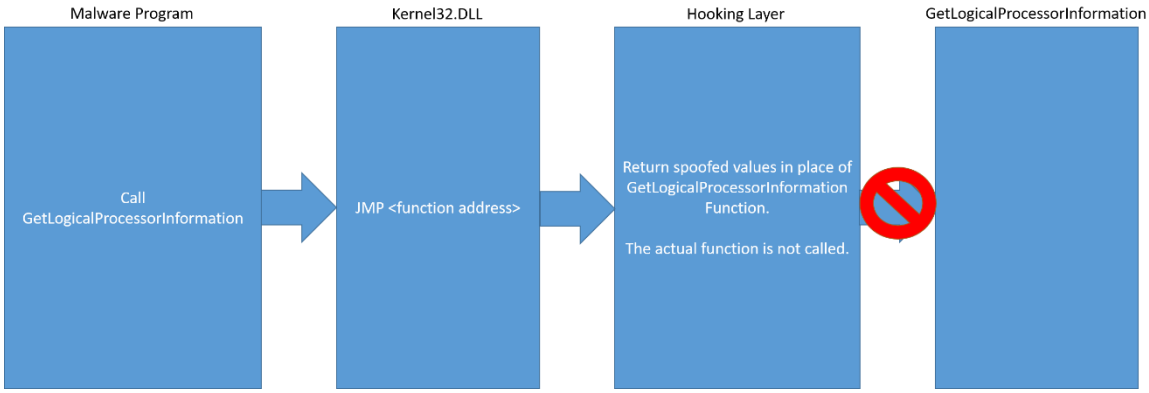


Figure 15: API Hooking of the GetLogicalProcessorInformation Function

With this in mind, it is possible to spoof the number of physical CPU cores, the number of logical CPU cores and the CPU caching capacities to match with a CPU model that is different from the one the virtual machine is using. Combining this with CPUID spoofing to return a CPUID processor string with matching details will defeat our proposed method of virtualization detection. Figure 16 shows a demonstration where we were able to spoof a VMware Workstation VM into thinking it was interfacing with an AMD Ryzen 3 3200G CPU. Due to the fact VMware Workstation only assigns 1 thread per core for a VM, we chose this CPU because it has an equal number of physical and logical cores. From there, we "hooked" the "GetLogicalProcessorInformation" API function to return caching values that were correct for our chosen CPU model. Running "detectvm.exe", the program we wrote to check our CPU details with the CPU database shows that we have successfully defeated our method of detection.

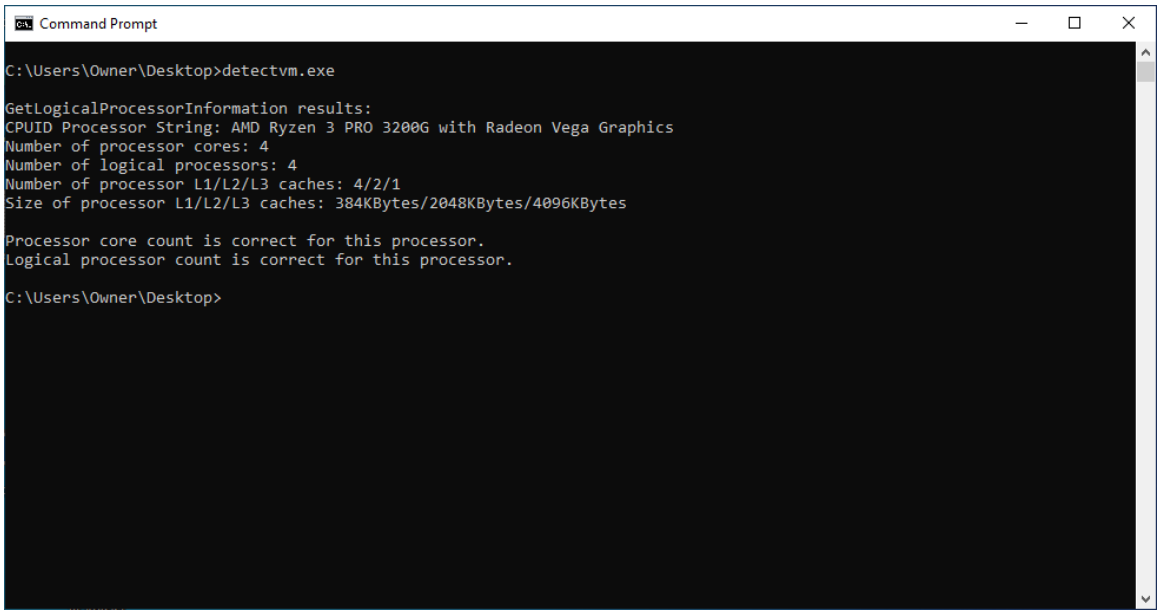


Figure 16: API Hooking and CPUID Spoofing a VMware Workstation VM

6.2.6 System Emulation

Emulation involves a piece of software “pretending” to represent a system present so a piece of software can run properly. In the context of this method of VM detection, it is possible to emulate a CPU such that all relevant CPU details are consistent with the real hardware. As evidenced by our results with KVM/QEMU in section 5.6, it is feasible to use emulation to defeat our method of detection.

Chapter 7. Conclusions and Future Works

7.1 Thesis Summary

This thesis aimed to explore the usage of CPU details as a means of VM detection. With malware authors seeking out new ways of detecting VMs to make their malware stay undetected for longer, it is important for researchers to also be seeking out ways of detecting VMs to stay one step ahead. We went about detecting VM presence by checking the physical CPU core count, the logical CPU core count, and caching information exposed to a system. We have presented the fact that if the number of CPU cores assigned to a virtual machine is less than the maximum number of cores built into the CPU in use, an attacker could detect the presence of the virtual machine. We also demonstrated that even if all physical and logical cores of the CPU are assigned to a virtual machine, inconsistencies still existed in many cases that could be used to detect the presence of the VM as detailed in Chapter 5. We believe our method of detection is of significant concern and should be addressed by malware analysts along with anyone who regularly works with VM sensitive software. We believe it is practical for an attacker to utilize this method of detection to keep malware hidden from detection systems for longer.

7.2 Future Works

The work described in this thesis focuses on fingerprinting the CPU based on the number of physical CPU cores, the number of logical CPU cores, and the CPU cache capacities detected by the operating system. There exist other details about modern CPUs that could be used to supplement our method of VM detection. The obvious downside to this would be the increased size and complexity of the CPU database an attacker would need to keep and maintain, but a determined enough attacker could still make such a method work. This section intends to briefly detail these features.

7.2.1 Base Clock Speed

Each CPU has a base clock speed which can be different from CPU to CPU. In the case of our AMD Ryzen 9 5950X test bench, the base clock speed is 3.4GHz.

If we are to impersonate another CPU, the clock speed should match the CPU model.

7.2.2 CPU Supported Features

CPUID, the CPU instruction that we used in this thesis to identify the processor string, can obtain a multitude of other information about the CPU outside of just the processor string. One of these key pieces of information is the CPU features that are built into the processor, such as support for hyperthreading and various instruction sets. From the attacker's perspective, this information could be used to deduce the presence of a VM should the CPUID results differ from what is expected for the CPU model reported present. We can check this CPU information by executing the following code:

- MOV EAX, 1
- CPUID

The above code returns values in the EAX, EBX, ECX and EDX registers.

Processor version information is returned in the EAX register in the following format:

EAX	
Bit Range	**Description**
[0-3]	Stepping ID
[4-7]	Model
[8-11]	Family ID
[12-13]	Processor Type
[14-15]	RESERVED
[16-19]	Extended Model ID
[20-27]	Extended Model ID
[28-31]	RESERVED

Table 14: CPUID(EAX=1) EAX Register

Additional information will be made available in the EBX register concerning logical processors in the following format:

EBX

Bit Range	Description
[0-7]	Brand Index
[8-15]	CLFLUSH line size
[16-23]	Max number of addressable IDs for logical processors
[24-31]	Local APIC ID of the currently executing logical processor

Table 15: CPUID(EAX=1) EBX Register

The ECX and EDX registers offer information regarding miscellaneous pieces of information about the CPU such as instruction sets supported. For each bit, 1 indicates the processor supports the feature, while 0 indicates otherwise. The bit features are as follows:

ECX	
Bit Number	**Description**
1	Streaming SIMD Extensions 3 (SSE3).
2	PCLMULQDQ Instruction.
3	64 bit DS Area.
4	Monitor/MWAIT.
5	Virtual Machine Extensions.
6	Safer mode extensions.
7	Enhanced Intel SpeedStep Technology Present.
8	Thermal Monitor 2.
9	SSSE3 Extension is Present.
10	L1 Context ID
11	Value of 1 indicates the processor supports IA32_DEBUG_INTERFACE MSR for silicon debugging.
12	Value of 1 indicates the processor supports FMA extensions using YMM state
13	CMPXCHG16B Available.
14	xTPR Update Control.
15	Perfmon and Debug Capability.
16	RESERVED
17	Process-context identifiers.

18	Value of 1 indicates the processor supports the ability to prefetch data from a memory mapped device.
19	Value of 1 indicates the processor supports SSE4.1.
20	Value of 1 indicates the processor supports SSE4.2.
21	Value of 1 indicates processor supports x2APIC feature.
22	Value of 1 indicates the processor supports MOVBE instruction.
23	Value of 1 indicates the processor supports POPCNT.
24	A value of 1 indicates that the processor's local APIC timer supports one-shot operation using a TSC deadline value.
25	Value of 1 indicates the processor supports AESNI instruction extensions.
26	A value of 1 indicates that the processor supports the XSAVE/XRSTOR processor extended states feature, the XSETBV/XGETBV instructions, and XCR0.
27	A value of 1 indicates that the OS has set CR4.OSXSAVE[bit 18] to enable XSETBV/XGETBV instructions to access XCR0 and to support processor extended state management using XSAVE/XRSTOR.
28	Value of 1 indicates the processor supports AVX instruction extensions.
29	Value of 1 indicates the processor supports 16 bit floating point conversion instructions.
30	Value of 1 indicates the processor supports RDRAND instruction.
31	Hypervisor is Present.

Table 16: CPUID(EAX=1) ECX Register

EDX	
Bit Number	**Description**
0	Floating Point Unit On-Chip.

1	Virtual 8086 Mode Enhancements.
2	Debugging Extensions.
3	Page Size Extension.
4	Time Stamp Counter.
5	Model Specific Registers RDMSR and WRMSR Instructions.
6	Physical Address Extension.
7	Machine Check Exception.
8	CMPXCHG8B Instruction.
9	APIC On-Chip is present.
10	RESERVED
11	SYSENTER and SYSEXIT Instructions.
12	Memory Type Range Registers.
13	Page Global Bit.
14	Machine Check Architecture.
15	Conditional Move Instructions.
16	Page Attribute Table.
17	36-Bit Page Size Extension.
18	Processor Serial Number Present.
19	CLFLUSH Instruction
20	RESERVED
21	Debug Store
22	Thermal Monitor and Software Controlled Clock Facilities
23	Intel MMX Technology
24	FXSAVE and FXRSTOR Instructions
25	SSE Extension Support
26	SSE2 Extension Support
27	Self Snoop
28	Max APIC IDs reserved field is Valid
29	Thermal Monitor
30	RESERVED
31	Pending Break Enable

Table 17: CPUID(EAX=1) EDX Register

If an attacker possessed a database that contained what values that are to be in the EAX, EBX, ECX and EDX registers upon executing CPUID with EAX = 1, it may provide the attacker another means of detecting the presence of a VM or an otherwise spoofed CPU. It should be noted that some features can be “turned off” in the motherboard’s BIOS/UEFI, which could alter the return values of CPUID.

7.3 Thesis Conclusions

This thesis presented a novel method of virtual machine detection. 5 different hypervisors were tested using 10 different CPUs. CPU detail inconsistencies were found in many test cases despite all physical and logical CPU cores assigned to the test VMs. At the time of typing, we are not aware of any malware that is making use of this method of detection. It is vital that malware analysts and anyone who works with VM sensitive software to take note of this method, as it may pose a threat to current “hardened” VM environments.

Bibliography

[1] "Virtual Machine for Malware Analysis," 8 November 2021. [Online]. Available: https://www.geeksforgeeks.org/virtual-machine-for-malware-analysis/.

[2] L. Abrams, "New Evasion Encyclopedia Shows How Malware Detects Virtual Machines," 1 March 2020. [Online]. Available: https://www.bleepingcomputer.com/news/security/new-evasion-encyclopedia-shows-how-malware-detects-virtual-machines/.

[3] S. Vilkomir-Preisman, "Malware Evasion Techniques Part 2: Anti-VM Blog," 29 October 2019. [Online]. Available: https://www.deepinstinct.com/blog/malware-evasion-techniques-part-2-anti-vm-blog.

[4] A. Dinaburg, P. Royal, M. Sharif and W. Lee, "Ether: Malware Analysis via Hardware Virtualization Extensions," *ACM CCS 2008,* 2008.

[5] J. Rutkowska, "Red Pill... or how to detect VMM using (almost) one CPU instruction," November 2004. [Online]. Available: https://www.invisiblethings.org/papers/redpill.html.

[6] D. Mandal and Y. Zhang, "Black Hat 2017 Europe," 2017. [Online]. Available: https://www.blackhat.com/eu-17/briefings.html#the-great-escapes-of-vmware-a-retrospective-case-study-of-vmware-g2h-escape-vulnerabilities.

[7] "History of Virtualization," Probrand, [Online]. Available: https://www.probrand.co.uk/it-services/vmware-solutions/history-of-virtualisation.

[8] G. J. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," *Communication of the ACM,* vol. 17, no. 7, pp. 412-421, 1974.

[9] "Understanding Full Virtualization, Paravirtualization, and Hardware Assist," 2007.

[10] K. Adams and O. Agesen, "A Comparison of Software and Hardware Techniques for x86," ACM, San Jose, 2006.

[11] L. Singh and S. I. Hassan, "Virtualization Evolution For Transparent Malware Analysis," *International Journal of Scientific Research,* vol. 2, no. 6, p. 101, 2013.

[12] "Malware," AV-ATLAS, [Online]. Available: https://www.av-test.org/en/statistics/malware/.

[13] S. Morgan, "Cybercrime To Cost The World $10.5 Trillion Annually By 2025," [Online]. Available: https://cybersecurityventures.com/hackerpocalypse-cybercrime-report-2016/.

[14] "Virus:DOS/CIH," F-Secure, [Online]. Available: https://www.f-secure.com/v-descs/cih.shtml.

[15] "Worm: W32/Mydoom," F-Secure, [Online]. Available: https://www.f-secure.com/v-descs/novarg.shtml.

[16] "The life and death of the Zeus Trojan," Malwarebytes Labs, [Online]. Available: https://www.malwarebytes.com/blog/news/2021/07/the-life-and-death-of-the-zeus-trojan.

[17] "What is Monero (XMR)?," [Online]. Available: https://www.getmonero.org/get-started/what-is-monero/.

[18] "73 Ransomware Statistics Vital for Security in 2022," Panda Security, [Online]. Available: https://www.pandasecurity.com/en/mediacenter/security/ransomware-statistics/.

[19] "Ransomware As A Service (RAAS) Explained," Crowdstrike, 2 February 2022. [Online]. Available: https://www.crowdstrike.com/cybersecurity-101/ransomware/ransomware-as-a-service-raas/.

[20] "EternalBlue - Everything There is to Know," Check Point Reserach, 29 September 2017. [Online]. Available: https://research.checkpoint.com/2017/eternalblue-everything-know/.

[21] "What is WannaCry Ransomware?," Kaspersky, [Online]. Available: https://www.kaspersky.com/resource-center/threats/ransomware-wannacry.

[22] "The Best Parental Control to Keep Your Kids and Family Safe Online," Net Nanny, [Online]. Available: https://www.netnanny.com/.

[23] "What is Adware? - Definition and Explanation," [Online]. Available: https://www.kaspersky.com/resource-center/threats/adware.

[24] "FIREBALL – The Chinese Malware of 250 Million Computers Infected," Check Point Research, 1 June 2017. [Online]. Available: https://blog.checkpoint.com/2017/06/01/fireball-chinese-malware-250-million-infection/.

[25] "Emotet," Malwarebytes, [Online]. Available: https://www.malwarebytes.com/emotet.

[26] "How ransomware spreads: 9 most common infection methods and how to stop them," Emsisoft, 19 December 2019. [Online]. Available: https://blog.emsisoft.com/en/35083/how-ransomware-spreads-9-most-common-infection-methods-and-how-to-stop-them/.

[27] N. Mott, "Scammers Distribute Crypto-Mining Malware via Cracked Games," 26 June 2021. [Online]. Available: https://www.pcmag.com/news/scammers-distribute-crypto-mining-malware-via-cracked-games.

[28] "Stuxnet," Malwarebytes, [Online]. Available: https://www.malwarebytes.com/stuxnet.

[29] "Evasive malware goes mainstream," Lastline Labs, 22 April 2015. [Online]. Available: www.net-security.org/malware_news.php?id=3022.

[30] "What is Cuckoo?," [Online]. Available: https://cuckoosandbox.org/.

[31] "VMProtect Software," [Online]. Available: https://vmpsoft.com/.

[32] "EXEINFO PE," [Online]. Available: http://www.exeinfo.byethost18.com/?i=1.

[33] O. Aslan and R. Samet, "A Comprehensive Review on Malware Detection," *IEEE Access,* vol. 8, pp. 6249-6271, 2020.

[34] K. Alzarooni, "Malware Variant Detection," 2012.

[35] M. D. Schroeder and J. H. Saltzer, "A Hardware Architecture for Implementing Protection Rings," *Communications of the ACM,* vol. 15, no. 3, pp. 157-170, 1972.

[36] D. Barrett and G. Kipper, Virtualization and forensics : a digital forensic investigator's guide to virtual environments, Amsterdam: Syngress/Elsevier, 2010.

[37] "Intel® Virtualization Technology (Intel® VT)," [Online]. Available: https://www.intel.ca/content/www/ca/en/virtualization/virtualization-technology/intel-virtualization-technology.html.

[38] "AMD Pro Technologies," [Online]. Available: https://www.amd.com/en/technologies/pro-technologies.

[39] "Memory Virtualization," VMware Inc., 31 May 2019. [Online]. Available: https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.resmgmt.doc/GUID-6E85F6DE-7365-4C28-B902-725D3C76C2E6.html.

[40] "Characteristics of Virtualization," 11 July 2022. [Online]. Available: https://www.geeksforgeeks.org/characteristics-of-virtualization/.

[41] "Overview of virtual machine snapshots in vSphere (1015180)," VMware Inc, 24 January 2022. [Online]. Available: https://kb.vmware.com/s/article/1015180.

[42] L. S. Vailshery, "Application virtualization market revenue worldwide in 2018 and 2026," Statista, 6 June 2022. [Online]. Available: https://www.statista.com/statistics/1252479/application-virtualization-market-size/.

[43] "Develop faster. Run anywhere.," Docker, [Online]. Available: https://www.docker.com.

[44] J. Johnson, "Four container security vulnerabilities and how to avoid them," [Online]. Available: https://www.techtarget.com/searchsecurity/tip/Container-security-best-practices-help-mitigate-risks-and-threats.

[45] "Kubernetes," [Online]. Available: https://kubernetes.io/.

[46] "Containerization, the next generation of virtualization," Hewlett Packard Enterprise, 15 November 2021. [Online]. Available: https://community.hpe.com/t5/hpe-blog-saudi-arabia/containerization-the-next-generation-of-virtualization/ba-p/7154442.

[47] C. Newton, "How Do Emulators Work? A Beginner’s Guide to Video Game Emulation," [Online]. Available: https://bagogames.com/how-do-emulators-work-a-beginners-guide-to-video-game-emulation/.

[48] "Difference between Emulation and Virtualization," [Online]. Available: https://www.javatpoint.com/emulation-vs-virtualization.

[49] "VMware Backdoor I/O Port," 6 February 2008. [Online]. Available: https://sites.google.com/site/chitchatvmback/backdoor.

[50] "Mechanisms to determine if software is running in a VMware virtual machine (1009458)," Vmware Inc., 1 May 2015. [Online]. Available: https://kb.vmware.com/s/article/1009458.

[51] T. Klein, "ScoopyNG," 2008. [Online]. Available: https://www.trapkit.de/tools/scoopyng/.

[52] D. Quist and V. Smith, "Detecting the Presence of Virtual Machines Using the Local Data Table," p. 9, 2009.

[53] "SetThreadAffinityMask function (winbase.h)," Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-setthreadaffinitymask.

[54] "Global Descriptor Table," [Online]. Available: https://wiki.osdev.org/Global_descriptor_table.

[55] "AMD 64 Architecture Programmer's Manual Volume 3," in *SLDT Store Local Descripton Table Register*, 2021, p. 457.

[56] "Interrupt Descriptor Table," [Online]. Available: https://wiki.osdev.org/Interrupt_Descriptor_Table.

[57] "More tricks from Conficker and VM detection," SANS, 10 February 2009. [Online]. Available: https://isc.sans.edu/forums/diary/More+tricks+from+Conficker+and+VM+detection/5842/.

[58] D. Quist, "VMDetect," 2006. [Online]. Available: http://www.offensivecomputing.net/dc14/vmdetect.cpp.

[59] "Intel® 64 and IA-32 Architectures Software Developer Manual Volume 2A," in *CPUID Instruction*, 2021, p. 234.

[60] "AMD 64 Architecture Programmer's Manual Volume 3," in *Obtaining Processor Information Via the CPUID Instruction*, 2021, p. 595.

[61] "CPUID - OSDev Wiki," [Online]. Available: https://wiki.osdev.org/CPUID.

[62] "VM Detection Methods in Malware | G DATA," G Data, 8 May 2020. [Online]. Available: https://www.gdatasoftware.com/blog/2020/05/36068-current-use-of-virtual-machine-detection-methods.

[63] E. Traut, "Systems and methods for using synthetic instructions in a virtual machine". United States Patent US7552426B2, 14 October 2003.

[64] "Intel® 64 and IA-32 Architectures Software Developer Manual Volume 3," Intel, December 2021. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[65] "Necurs.P2P – A New Hybrid Peer-to-Peer Botnet," MalwareTech, 22 February 2016. [Online]. Available: https://www.malwaretech.com/2016/02/necursp2p-hybrid-peer-to-peer-necurs.html.

[66] "Just Another VM Detection (was "VM Detection Combo")," 18 August 2009. [Online]. Available: http://my.opera.com/jaelanicu/blog/just-another-vm-detection-was-vm-detection-combo.

[67] "VPCEXT," [Online]. Available: https://search.unprotect.it/technique/vpcext/.

[68] "Detect if your program is running inside a virtual machine," 05 April 2005. [Online]. Available: https://www.codeproject.com/Articles/9823/Detect-if-your-program-is-running-inside-a-Virtual.

[69] "GravityRAT - The Two-Year Evolution Of An APT Targeting India," Talos Intelligence, 26 April 2018. [Online]. Available: https://blog.talosintelligence.com/2018/04/gravityrat-two-year-evolution-of-apt.html.

[70] teixeira0xfffff, "Anti-VM Techniques with MSAcpi_ThermalZoneTemperature," 2019. [Online]. Available: https://gist.github.com/teixeira0xfffff/36293713c254c69a7ba2353e8d64afce#file-msacpi_thermalzonetemperature-ps1.

[71] M. Noorafiza, H. Maeda, T. Kinoshita and R. uda, "Virtual Machines Detection Methods using IP Timestamps pattern Characteristic," *International Journal of Computer Science & information Technology (IJCSIT),* vol. 8, 2016.

[72] "GetLogicalProcessorInformation function (sysinfoapi.h)," Microsoft, 13 10 2021. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-getlogicalprocessorinformation.

[73] S. Grinberg, "API Hooking – Tales from a Hacker’s Hook Book," Cynet, 29 October 2020. [Online]. Available: https://www.cynet.com/attack-techniques-hands-on/api-hooking.

[74] "A powerful disassembler and a versatile debugger," hex-rays, [Online]. Available: https://hex-rays.com/ida-pro/.

[75] "Ghidra," NSA, [Online]. Available: https://ghidra-sre.org/.