University of Zurich^UZH

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

HyperDtct: Hypervisor-Based Ransomware Detection

Jan Marc Lüthi
Zurich, Switzerland
Student ID: 21-740-204

Supervisor: Jan von der Assen, Chao Feng, Dr. Alberto Huertas Celdran
Date of Submission: July 1st, 2024

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Kurzfassung

Ransomware stellt eine wachsende Bedrohung für Institutionen und kritische Infrastruktur dar. Das Aufkommen von Ransomware-as-a-Service ermöglicht es auch Benutzern ohne tiefergreifende technische Kenntnisse, hochwertige Ziele anzugreifen, was Ransomware zu einem lukrativen kriminellen Geschäftsmodell macht. Um die Ausführung von Ransomware zu verhindern, werden neue Erkennungs- und Klassifizierungssysteme benötigt. Zwar existieren zahlreiche Lösungen zur Erkennung von Ransomware, doch viele sind potenziell unsicher, da sie sich auf demselben Betriebssystem wie die Malware befinden. Einige Lösungen nutzen einen Hypervisor, um ihr Erkennungssystem von der Ransomware zu isolieren und zu verbergen. Um Einsicht in die virtuelle Maschine zu erhalten und die zur Erkennung von Malware erforderlichen Daten zu sammeln, greifen viele auf dieselbe Library zurück.

Diese Arbeit stellt *HyperDtct* vor, einen neuartigen Ansatz zum Sammeln von Systemaufrufen von der Hypervisor-Ebene. *HyperDtct* nutzt die gesammelten Systemaufrufe zur Erkennung von Ransomware. Das vorgeschlagene System ist eine Sandbox. Verschiedene Algorithmen zur Klassifizierung und Erkennung von Anomalien sowie Techniken zur Auswahl von Features werden anhand von dreizehn gutartigen und elf Ransomware Beispielen evaluiert. Darunter befinden sich verbreitete und schädliche Beispiele wie Babuk und LockBit Dark. Die Ergebnisse dieser Evaluierungen zeigen, dass *HyperDtct* die betrachteten Beispiele mit einem hohen F_1-Score von 0.97 in gutartig und bösartig klassifizieren kann. Zusätzlich haben die Experimente gezeigt, dass *HyperDtct* die für das System bisher unbekannte Ransomware LockBit und Babuk in weniger als zehn Sekunden erkennen kann.

Abstract

Ransomware is an ever-growing threat affecting institutions and critical infrastructure. The emergence of Ransomware-as-a-Service also allows users without advanced technical knowledge to target high-value targets, which makes ransomware a lucrative criminal enterprise. New detection systems are needed to detect ransomware's malicious behavior. Although numerous solutions exist to detect ransomware, many are vulnerable because they operate on the same operating system as the malware. Some approaches leverage hypervisors to isolate and conceal their detection system from the ransomware. Many rely on the same library to introspect the virtual machine and gather the data necessary to detect malware.

This work proposes *HyperDtct*, a novel way to collect system calls at the hypervisor level and detect ransomware based on these collected logs. The proposed system functions as a sandbox. The experiments conducted throughout this thesis assess various classifier and anomaly detection algorithms and feature selection techniques using thirteen benign samples and eleven ransomware samples. This includes prolific and harmful samples such as Babuk and LockBit Dark. These evaluations indicate that *HyperDtct* can classify the considered samples with a high F_1 score of 0.97 into benign and malicious. Experiments have also shown that *HyperDtct* can detect the previously unseen samples LockBit and Babuk within less than ten seconds.

Acknowledgments

First and foremost, I would like to thank my supervisor, Jan von der Assen, for his support and guidance, the exciting discussions, and his tremendously helpful suggestions throughout this thesis. His unwavering enthusiasm and knowledge of the topic greatly contributed to the success of this work and made the entire process much more engaging. Furthermore, I would like to extend my gratitude to Prof. Dr. Burkhard Stiller and the Communication Systems Group for providing me with the opportunity to explore such a fascinating topic.

Contents

Chapter 1

Introduction

Ransomware is malware that encrypts files on a device, making them useless to the victim. It demands a ransom to decrypt and restore access to the files. The earliest appearance of ransomware can be traced back to 1989 when Joseph Popp created the AIDS ransomware program. This program was spread as a Trojan on a floppy disk, which was state-of-the-art at the time. After the malware successfully encrypted the files on the main system drive, it demanded a ransom payment to be sent to a post office box [1].

Since then, ransomware has become an ever-growing global threat, targeting high-profile institutions and critical infrastructure, ranging from hospitals and schools to government agencies. Criminals now have various payment techniques to extract the ransom anonymously: various prepaid services and digital currencies, such as Bitcoin, offer anonymity and are not tied to traceable bank accounts [1].

With the emergence of Ransomware-as-a-Service (RaaS), the business model of ransomware has become even more effective, allowing attackers without advanced technical skills to acquire highly sophisticated ransomware and to attack high-value targets [2], [3]. Examples of sophisticated ransomware attacks targeting high-value targets include the attack on the D.C. Metropolitan Police Department by the Babuk ransomware group [4] and the attack on Colonial Pipeline Co., the largest oil pipeline in the United States of America, by the hacker group DarkSide [5]. In both instances, the attackers demanded over USD 4 million as ransom to restore the encrypted files and prevent the publication of sensitive data. While Colonial Pipeline Co. paid the requested ransom to DarkSide, Babuk rejected the police department's counteroffer of USD 100,000 and published the extorted sensitive data. In 2023, ransomware payments reached an all-time high of USD 1 billion [6], as companies suffer more frequent and sophisticated ransomware attacks.

1.1 Motivation

Due to the ongoing growth of the threat ransomware poses, there is a need for detection systems to detect ransomware. These detection systems leverage two methods to gather data for detecting ransomware [7]: *static* methods try to extract features from the executable files without executing them. While this method is secure since the file does not need to be executed to classify it as benign or malware, it is also easily circumvented by malware. *Dynamic* methods, on the other hand, execute the file and collect features from its runtime behavior. This is harder to evade for malware, as even malware that tries to obfuscate its behavior must execute some fundamental behavior to reach its malicious purpose. To leverage the collected data, a machine learning (ML) model can be utilized to detect malware. ML has been widely deployed in malware detection because it outperforms heuristic detection approaches, particularly for unknown and complex malware [8].

While many existing ransomware detection systems run in the same operating system as the ransomware, this approach can be bypassed if ransomware elevates its privileges to the kernel level. Indeed, privilege escalation is attempted and partially achieved by several ransomware families and samples [9], [10]. A popular approach to keep the detection system isolated from the operating system is Virtual Machine Introspection (VMI). VMI enables the collection of behavioral data from outside a virtual machine at the hypervisor level, which has the advantage of having higher privileges than the kernel mode of the guest operating systems (OS). Even if a rootkit gains access to the kernel of the guest-level operating system, the detection tool remains secure.

Various studies use VMI to extract information from virtual machines while keeping their detectors safe in another VM or machine (*cf.*, Section 3.1). These previous works then use the collected features to detect whether malware is running in the guest VM using detection methods ranging from ML to heuristic detection. Various related works rely on system calls as features [10]–[13], showing how efficient these are as features for detecting malware. However, studies detecting malware solely based on system calls rely on the LibVMI [14] library to collect their behavioral data. Anti-forensic malware could find and exploit a way to bypass detection in these environments, making research on system-call-based detection systems using LibVMI ineffective. Therefore, there is a need to find new ways to collect system calls of malware samples at the hypervisor level and evaluate whether this data can be used for detection systems. HyperDbg [15] is a hypervisor-based debugger that can be used for high-performance and stealthy debugging of user and kernel applications. The debugger allows for the automation of debugging and monitoring flows and could, therefore, be used to automatically collect data from malicious samples at the hypervisor level. The developers have already shown that their debugger is stealthy enough to monitor a large proportion of evasive malware. However, it remains yet to be seen whether the extracted data can be used to detect ransomware.

1.2 Description of Work

This work proposes *HyperDtct*, a prototypical detection system based on HyperDbg. *HyperDtct* is designed and implemented as a proof-of-concept system that demonstrates how system calls can be collected using HyperDbg to detect and classify benign and malicious software. The system is implemented as a sandbox designed to execute potentially malicious executables in a safe environment isolated from vulnerable systems.

To verify that the data collected with the proposed system can be used for a detection system, several ML and anomaly detection (AD) algorithms were implemented and fitted to the collected data. The effectiveness and performance of these models were evaluated on eleven ransomware samples, ranging from ransomware built for academic purposes to recent samples of Babuk and LockBit Dark ransomware. Additionally, thirteen benign samples were used for comparison. The results achieved on all samples indicate that *HyperDtct* can successfully be used as a detection system, with the best model achieving an F_1 score of 0.97 when trained and evaluated on all samples.

1.3 Thesis Outline

After this chapter introduces the reader to this work, Chapter 2 provides background on how hypervisors work, the process of VMI, relevant malware and how it can be detected, and finally, how ML can be leveraged for this detection. This knowledge assists the reader in comprehending related work described in Chapter 3 before the different resulting research opportunities are explored in Chapter 4, elaborating on the VMI capabilities of different Hypervisors. Chapter 5 introduces the reader to the scenario, gives a system overview, and elaborates on how data is extracted and preprocessed for the models before Chapter 6 shows how *HyperDtct* is initially implemented. Then, this implementation is evaluated using various ransomware samples and extended in Chapter 7, and finally, the last chapter summarizes the findings and future work.

Chapter 2

Background

The purpose of this chapter is to provide the information needed to understand this thesis and its related work. First, different hypervisors, their architectures, and basic functionality are elaborated. Subsequently, the discussion shifts to the concept of virtual machine introspection before malware and malware detection are explored. The chapter culminates with an introduction to machine learning in the context of computer security, thus providing a comprehensive overview of the background knowledge required for this thesis.

2.1 Hypervisor

Virtualization allows splitting up and sharing the resources of a single physical machine into several virtual machines (VMs). This is enabled by a hypervisor or virtual machine monitor (VMM). Hypervisors are responsible for presenting the guest OS with a virtualized version of the hardware and scheduling its executions on the physical hardware.

The Intel x86 architecture is built based on four protection rings, where ring 0 (kernel mode) is the most privileged and ring 3 (user mode) is the least privileged [16]. Hypervisors operate with more privileges than kernel mode. Thus, they operate at ring -1, although this is not an actual protection ring [7].

Two main types of hypervisors are distinguished: Type I and Type II [17]: Type II or hosted hypervisors run on a host OS and use various functions of the underlying OS, such as its file system, to run their VMs. Examples of this type of hypervisor include VMware Workstation or Oracle Virtual Box. Type I, or native/bare-metal hypervisors, run directly on the hardware and thus do not have the layer of a host OS underneath them. Examples include Xen Project and VMware ESXi. In the subsequent chapters, the term ”hypervisor” is used synonymously with a Type I hypervisor, as this thesis focuses on Type I hypervisors. The difference between the two types of hypervisors is further illustrated in Figure 2.1.

The tasks of a hypervisor can be illustrated in the following characteristics [18]:

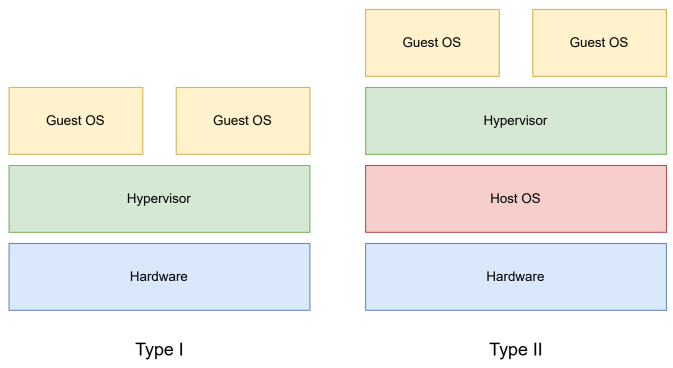


Figure 2.1: Type I vs. Type II Hypervisor

1. The VMM provides an environment for programs that is essentially identical to the original machine

2. Programs executed in this environment show, at worst, only minor decreases in speed

3. The VMM is in complete control of system resources

When an OS runs a process, it runs it in user mode, while the OS itself runs in kernel mode. To satisfy requirement number 3, a guest OS must run in user mode, but to adhere to requirement number 1, the guest OS should not be aware of this. [18] define two sets of instructions: *Sensitive instructions* are those that can only be executed in kernel mode, while *privileged instructions* trigger a trap if executed in user mode. A trap is a synchronous exception intentionally triggered to invoke the OS [19]. When the guest OS now executes a sensitive instruction, which is only allowed in kernel mode, the hypervisor thus must trap the resulting system call and execute it as if the guest OS was running directly on the hardware while keeping the VM isolated from other VMs possibly running concurrently on the same hardware [19].

There are various approaches to do this [19]. **Trap-and-emulate** emulates all kernel-mode code at the hypervisor level. Before Intel and AMD introduced virtualization in their CPUs in 2005, this approach was only possible if the sensitive instructions were a subset of the privileged instructions, as defined by [18]. Because of this, there was a need for other approaches to virtualization. **Binary translation** allowed VMs before hardware-assisted virtualization: The hypervisor replaces sensitive instructions in code executed on a guest VM at runtime with calls to the hypervisor to handle them. With various optimizations, this method can operate at nearly full speed. **Paravirtualization**, on the other hand, modifies the source code of the guest VMs: Instead of executing sensitive instructions, the guest OS invokes hypercalls, similar to processes issuing system calls to the OS. The hypervisor is responsible for providing an application programming interface (API) to handle said hypercalls. **Hardware-assisted virtualization**: Intel and AMD extended most of their CPUs to support hardware-assisted virtualization. This is done with additional instructions, specifically for virtualization, as well as adding hardware support for additional page tables required by VMs. With these extensions, the trap-and-emulate approach becomes possible. Today, most hypervisors discussed in this thesis rely on hardware support whenever possible.

2.1.1 Xen

The Xen hypervisor refers to its guest VMs as "domains." It supports two kinds of domains: Control Domain (Dom0) and unprivileged domains (DomU). Each Xen hypervisor has exactly one Dom0, which is the first VM it starts and contains all the physical drivers required for the hardware the system is running on. This VM can also be used to monitor and manage the other unprivileged VMs [20]. Figure 2.2 illustrates a simplified Xen architecture and its interaction with the hardware.

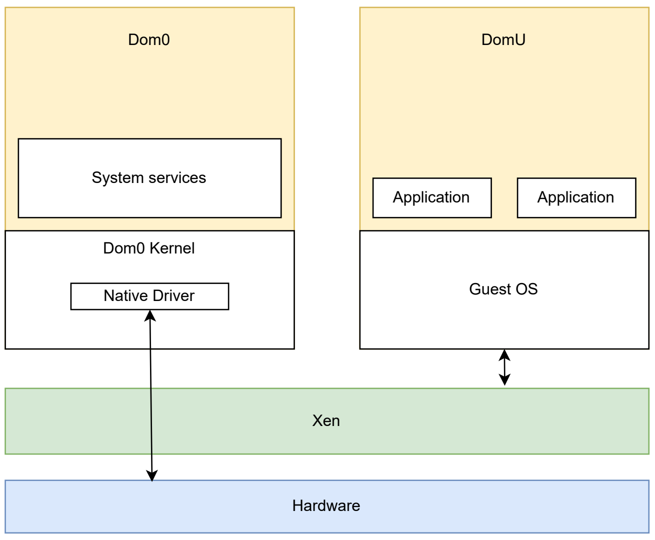


Figure 2.2: Simplified Xen Architecture

2.1.2 Kernel-Based Virtual Machine

Using Kernel-based Virtual Machine (KVM), we can turn a Linux Kernel into a hypervisor. VMs running on that hypervisor appear as running processes on the host. Contrary to Xen, the Control Domain (or Dom0) is not on the same level as the guest VMs: The OS whose kernel was turned into a hypervisor is now operative below the guest VM, as portrayed in figure 2.3 [21]. Because of this, the KVM environment can be interpreted as a Type 2 hypervisor (*e.g.*, as [19] did). However, this thesis interprets the entire KVM environment as a Type 1 hypervisor because the kernel, which is turned into a hypervisor, operates directly on the hardware and has no OS underneath it.

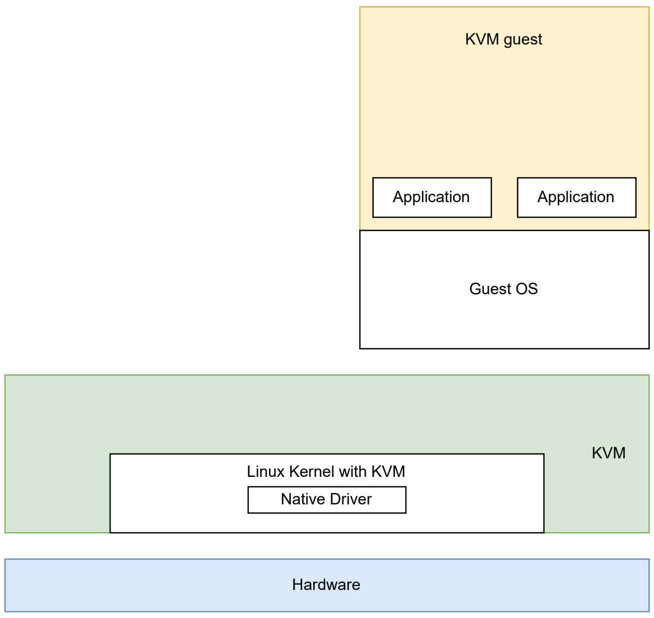


Figure 2.3: Simplified KVM Architecture

2.1.3 ESXi

Similarly to KVM, VMware ESXi does not run a privileged VM but instead implements a microkernel at the hypervisor level. This means that while it has the same structure as KVM, contrary to KVM, the footprint of an ESXi hypervisor is smaller, as it implements only the minimal mechanisms of an OS [22]. Also, contrary to the hypervisors mentioned above, ESXi is proprietary software.

2.1.4 Hyper-V

Like ESXi, Microsoft Hyper-V is proprietary software distributed for free with Windows and Windows Server. VMs running on a Hyper-V hypervisor are named ”Partition,” and similarly to Xen, there is a privileged *Root Partition* and possibly multiple unprivileged *Child Partitions*. The *Child Partitions* are managed and created from the *Root Partition* and can be further distinguished into enlightened and unenlightened: Enlightened Partitions are aware that they run in a virtualized environment. Thus, they accelerate their I/O operations by accessing hardware directly over VMBus and bypassing the slower emulated hardware. On the other hand, unenlightened Partitions access resources over the hypervisor [23]. The difference between enlightened and unenlightened *Child Partitions* is visualized in Figure 2.4.

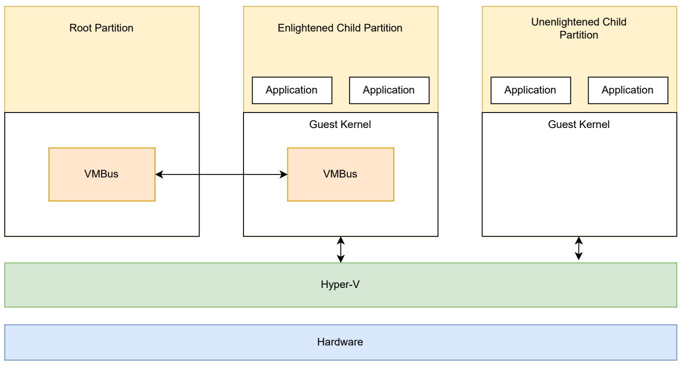


Figure 2.4: Simplified Hyper-V Architecture

2.2 Virtual Machine Introspection

VMI is a method that allows obtaining internal state information from a guest VM from outside of said VM: The security monitor is located at the hypervisor level and inspects a guest VM. This approach has two benefits [24]. First, the monitor is isolated from the guest VM, making tampering with the detection system more difficult for malicious software. Second, because the hypervisor has access to the hardware state of the VM, the monitor can obtain information about the VM's activities. The concept was first introduced by [25]: The authors created Livewire, which detects intrusions in a modified version of VMware Workstation.

Because the required data is collected below the OS at the hypervisor level, thus being below the high-level abstraction provided by the OS (*e.g.*, Processes, files, etc.), there is a semantic gap between the low-level data collected and the desired higher-level view of the OS. This semantic gap is one of the biggest challenges of VMI [26]. "Bridging the semantic gap" refers to deriving the higher-level view of the introspected OS from low-level data collected at the hypervisor level.

There are various approaches to bridge the semantic gap, which [27] classified into four categories, depending on how the semantic information is retrieved:

- **In-VM**: Approaches that fall into this category do not bridge the semantic gap. Instead, they avoid it by deploying an agent inside the guest VM, which exposes the guest OS' behavior to the hypervisor.
- **Out-of-VM delivered**: These approaches use delivered semantic information to bridge the semantic gap. Thus, they require information about the guest OS internals, which makes VMI deeply tied to the guest OS type and version.
- **Out-of-VM derived**: To avoid the shortcomings of Out-of-VM delivered approaches, out-of-VM derived VMI makes use of hardware-provided functionalities by observing and interpreting hardware states and events, such as writes to the CR3 control register to identify different processes. This makes these techniques OS-agnostic and resistant to kernel data attacks and specific malware evasion techniques.

- **Hybrid techniques**: These VMI techniques combine the abovementioned categories to achieve more robustness and/or reliability.

Various research studies were conducted on each category; however, the available technology depends mainly on the hypervisor. For a concise overview of what tool is available on which hypervisor, see Section 4.1.

VMI enables the investigation of a guest VM without interrupting it. Because of its independence from the monitored OS, VMI assists in malware collection, malware analysis, intrusion detection, intrusion prevention, stealthy debugging, cloud security, and mobile security [28].

2.3 Malware

Malware is an acronym for malicious software, while benign software refers to harmless software. Malware is code running on a computerized system whose presence or behavior the system administrators are unaware of; if they were aware of the code and its behavior, they would not permit it to run. Malware compromises the confidentiality, integrity, or availability of the system by exploiting existing vulnerabilities in a system or by creating new ones [7].

Zero-day vulnerabilities (or zero-days) are those vulnerabilities for which no patch or fix has been publicly released and where the vendor might not even be aware of the vulnerability yet. Several parties are interested in information about these vulnerabilities, such as software vendors and cybercriminals, and finding such information can fetch a great deal of money [29].

Various types of malware exist; however, this thesis focuses on the type of ransomware, which is explained in its own subsection. Ransomware employs multiple counter-measures against detection and recovery systems, such as **Rootkits**, which elevate the malware's privileges. A hypervisor-based analysis environment is safe from this, as the hypervisor operates below the kernel of the guest VM; however, certain rootkits are posing a threat to this environment: The firmware operating a device, such as a USB device, can be updated with malicious firmware exposing the hypervisor and enabling malicious operations on it [30]. After elaborating on ransomware, further evasion techniques employed by malware are explored.

2.3.1 Ransomware

Ransomware can be categorized into four main forms [31]. **Scareware** informs the user that they have been infected with malware. This information is usually delivered by a fake antivirus interface, which includes offers to purchase software to remove the malware. The scareware is removed as soon as the user purchases this software and runs it. Contrary to other forms of ransomware, most scareware does no damage to the system and instead hopes to extract money from users by simply scaring them [7]. **Locker ransomware** locks the user out of the computer and tries to block access to the device or an application. This form of ransomware is generally easy to overcome by rebooting in safe mode or running an on-demand virus scanner [32]. **Leakware or Doxware** threatens to publish users' data unless a ransom is paid [31]. **Crypto ransomware** is the most common type of ransomware and the one focused on in this thesis. It aims to encrypt data important to victims but does not interfere with essential computer functions. The data is encrypted using cryptography algorithms such as AES and RSA [32]. As these encryption methods are currently almost impossible to decrypt if implemented correctly, this fosters the need to detect and stop the ransomware before it can encrypt too many files. Ransomware can, in most cases, not be distinctly put into one of those categories, as many families of ransomware employ multiple aspects of the forms above. For example, WannaCry encrypts data and threatens victims with releasing their data publicly unless the ransom is paid [31]. Many ransomware families also pressure victims by accusing them of a crime or setting a deadline by which the victim must pay ransom to recover their data [1].

Ransomware attacks typically follow a sequence, which can be split up into six primary stages [33]:

1. *Distribution*: The malware is spread to the targeted device in this phase. Several attack vectors exist, such as phishing emails or malicious websites.
2. *Infection*: The executable malicious code is downloaded, and the ransomware installs itself on the targeted device.
3. *Staging*: The ransomware sets up and establishes persistency to exist beyond a reboot. The malicious code establishes connectivity with the attackers' command and control server.
4. *Scanning*: The ransomware scans the targeted device and its networks for files to encrypt.
5. *Encryption*: This is the stage where the encryption of the files begins. The malware encrypts all previously identified files while the user is unaware of what's happening.
6. *Payday*: The victim is informed about what happened, and a ransom is demanded.

2.3.2 Evasive Malware

Malware authors actively want to prevent the detection of their software and use various approaches to achieve this. [34] have developed a compiler that automatically exports critical system calls to separate processes, thus splitting up suspicious system call patterns across multiple processes. Their results showed that malware compiled with their custom compiler could evade detection from real-world behavioral detection tools. With more research leveraging learning models to detect malware, evading the detection of such models has also gotten into the focus of malware authors. Depending on the features used by the model to detect malware, evading it becomes easy: [35] have shown that performing a few changes to malware binaries without compromising their functionality is enough to avoid a malware detection system. They argue that deploying a robust detection methodology that analyzes executable bytes may be challenging. [36], on the other hand, propose an AI-powered ransomware framework that can be integrated into existing ransomware samples, modifying its encryption algorithm, rate, and duration to avoid detection through behavior-based detection systems. With their work, they managed to circumvent detection through Deep Q-Learning and Isolation Forest models of the ransomware affecting the system in a few minutes with >90% accuracy.

Anti-forensic methods are a subcategory of evasive malware equipped with methods and tools to obfuscate investigations. They might include avoiding detection or detecting the presence of a forensic tool to alter the behavior accordingly [37]. This kind of evasive malware tries to identify whether they are executed in a forensic environment, such as under a debugger, in a VM, or inside a sandbox. If, for example, a VM is detected, the malware might alter its behavior by not executing or trying to infect the host. There are many ways to identify whether the current OS is running as a VM, such as identifying special instructions, timing measurements, and OS markers [38].

2.4 Malware Detection

Fighting against these evasion techniques, malware detection systems employ various methods to detect malware nonetheless. Detecting malware can be separated into three stages [8]: Malware analysis, feature extraction, and classification.

2.4.1 Malware analysis

Malware analysis is the process of extracting data from a malicious sample. Malware can be analyzed using dynamic or static analysis: While static analysis uses the extraction of information without executing the software, dynamic analysis extracts information from the running software [39].

The static analysis approach relies on attributes found in a sample, which can be extracted without executing the sample. Examples of such attributes include dynamic-link library (DLL) imports, hex dump, and assembly code, like [40] used to train and evaluate their detection model. The advantage of this approach is that it is swift. However, detectors relying only on static analysis can be easily evaded: Various code obfuscation techniques bypass models trained on static attributes. General code obfuscation techniques aim to confuse the understanding of how a program functions. These can range from simple layout transformations to complicated changes in control and data flow [41]. The most common code obfuscation technique is packing [42]: A packer is a program that transforms an executable binary file into another configuration using compression and/or encryption to protect/hide the executable's original content. While packing can be used with non-malicious intentions, it is commonly used by malware authors to buy time until detection or avoid detection altogether.

Evading detection systems relying on a dynamic analysis approach is more challenging, as this approach uses malware's interaction with the computer system as the data source. Such runtime behavior can include file operations, registry changes, network artifacts, or system calls. A disadvantage compared to static analysis is that this approach is more time-consuming and potentially infects the analysis environment.

Dynamic analysis requires several properties to hold to allow the safe collection of relevant data [7]:

- Data gathered by the analysis framework must not be compromised by the malware
- The analysis framework must not be detectable by the malware
- The framework needs to meet the requirements of the malware to execute its malicious behavior
- The framework must limit/emulate network access by the malware

Many researchers have proposed hybrid approaches, using static and dynamic features to leverage the advantages of both techniques when analyzing malware. [43] extracted 261 combined features from one dynamic and static analysis dataset. To test the detection model that was built, they took 311 application samples consisting of 165 benign apps from the Play Store and 146 malicious apps from VirusShare. The test results showed that their hybrid analysis model could increase detection by about 5%.

2.4.2 Detection Techniques

After the malware has been analyzed and relevant features extracted, different techniques exist to detect it. Each of these techniques comes with its advantages and disadvantages, and there is no single solution that fits all scenarios:

Signature-Based Detection Techniques

This approach creates signatures of malware executables and stores them in the detection system's database. To classify an unknown file using traditional signature-based techniques, its signature is created and compared with the signature of known malware in the database. If a match is found, the file is classified as malicious [39]. Various approaches exist to extract a signature. A signature can be a static feature of the executable file or also a behavioral signature. String scanning, for example, extracts byte sequences of malware executables. This technique is extensively used by antivirus scanners such as ClamAV [8]. [44], on the other hand, leveraged behavior sequences to create behavioral signatures for polymorphic malware detection. In general, signature-based techniques have the advantage that they are faster than behavior-based detection. However, these detection models are also very prone to obfuscation methods and cannot detect zero-day attacks [8].

Behavior-Based Detection Techniques

These techniques observe the behavior performed by malware during execution to detect it. This approach is not as easy to evade by obfuscation as the signature-based detection approach is: Although the program code might change, the program's behavior will still be similar [8]. Thus, the majority of new malware can be detected with this method [45]. Various procedures can be leveraged to detect malware based on behavior: Monitoring system calls, file changes, processes, and network, to name a few mentioned in [8]. Disadvantages of the behavior-based technique include the high computational overhead and the fact that the malware is already executing when it can be detected, potentially already doing damage [39].

2.5 Leveraging Machine Learning For Malware Detection

This thesis leverages ML to use the data extracted with VMI. In this context, ML refers to algorithms and processes that can generalize past data to predict future outcomes. Supervised ML methods use probabilities of previously observed events to infer the probabilities of new events, while unsupervised methods draw abstractions from unlabeled datasets and apply these to new data. On the other hand, AD is applied where a significant class imbalance is present, with most samples being "normal" while only a tiny percentage are outliers [46]. In the context of this work, the terms anomaly and outlier are used as synonyms.

Related work uses various supervised ML algorithms to classify malware. A broad overview of algorithms used in related work or this thesis is given here, based on [46]:

- **Naive Bayes (NB)**: The NB classifier is one of the oldest statistical classifiers and is called "naive" because it makes the strong statistical assumption that features are chosen independently from some unknown distribution. Even though this assumption often does not hold, it is quite effective for problems such as spam classification.

- **Support Vector Machine (SVM)**: An SVM is a linear classifier that tries to separate two classes in the dataset by producing a hyperplane in the vector space. It uses a hinge loss function, which penalizes the points on the wrong side of the hyperplane or very near it on the correct side.

- **K-Nearest Neighbors (KNN)**: KNN is a lazy learning algorithm that puts off most computations to prediction time instead of training time. Instead of generalizing training data, KNN stores them in the model and predicts a result based on the most common label out of the test samples k nearest neighbors.

- **Random Forest (RF)**: RF is an ensemble of decision trees. After training, predictions are made by letting each tree vote (classification) or taking the statistical mean of each tree's predictions (regression).

Unsupervised learning algorithms, on the other hand, offer the promise to skip the laborious step of feature engineering: Instead of carefully selecting features to feed the learning algorithm as with supervised learning, unsupervised learning algorithms extract the features themselves [46]. Related work used the following unsupervised ML algorithms [47]:

- **Deep Neural Network (DNN)**: DNNs are feed-forward networks, which means data flows through the graph composed of nodes and edges without forming a cycle. A DNN consists of an input layer, n hidden layers, and an output layer. Figure 2.5 provides an overview of the architecture.
- **Convolutional Neural Network (CNN)**: CNNs are primarily used in image processing. CNNs are structured similarly to DNNs; however, at least one of the hidden layers performs convolution. To use CNNs in Malware analysis, collected data must be encoded to images [48].

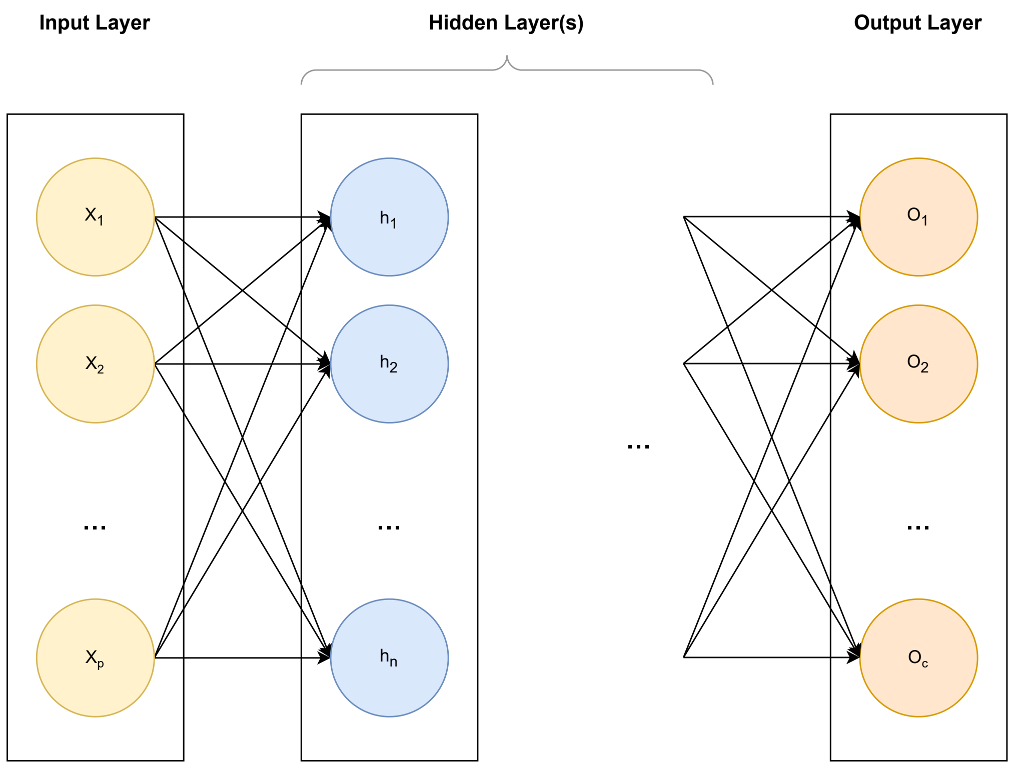


Figure 2.5: DNN Architecture, Adapted From [47]

The possibility of avoiding feature selection is very tempting; however, [49] found that RF accuracy outperforms the accuracy of DNN models in malware classification using various feature sets.

In contrast to the ML methods mentioned previously, AD is best applied on imbalanced datasets. Because malware is usually a rare occurrence, more data on benign behavior is collected, making applying supervised learning difficult because of the class imbalance problem. AD makes use of both supervised and unsupervised ML, as well as other techniques to detect outliers. AD can further be separated into *novelty detection*, where the training data is unpolluted by anomalies, and *outlier detection*, where it is assumed that the data is contaminated by some anomalies [46]. This work considers the following algorithms for AD, based on [46]. Although they are based on supervised ML algorithms, the modifications to these algorithms in the context of AD give them characteristics of unsupervised learning:

- **Isolation Forest (IForest)**: The algorithm of IForest iterates through data points in the training set, randomly selects a feature, and then randomly selects a split value in the range between the maximum and minimum of the selected feature in the dataset. Then, the number of such splits to isolate a single sample is considered, with the intuition being that anomalous samples require fewer splits than inliers to be isolated.
- **Local Outlier Factor (LOF)**: LOF classifies anomalies using the local density around a sample. This means it measures the concentration of other points in the immediate surrounding region, where the size of this region can be defined as either a fixed distance or the closest n neighbors. Samples with a significantly lower local density than their neighbors are considered anomalies.

2.5.1 Performance Metrics

But how can the performance of different algorithms be compared? All following definitions are taken from [46]. A straightforward approach to comparing two models is the **cost function**, computing the cost of the two models on the same dataset and choosing the lower-cost one. Assuming a binary (positive, negative) classification scenario, the predictions of a model can fall into four possible pairs:

- True positive (TP): predicted positive + actual positive
- True negative (TN): predicted negative + actual negative
- False positive (FP): predicted positive + actual negative
- False negative (FN): predicted negative + actual positive

To now use these metrics for comparing binary classifier models, plot the **receiver operating characteristic (ROC)** curve. This curve plots the false positive rate (FPR) (*cf.*, Equation 2.1) on the x-axis and the true positive rate (TPR) (*cf.*, Equation 2.2, also known as recall) on the y-axis. A truly random classifier would achieve an ROC curve of the form $y = x$, while a perfect classifier's ROC curve would enclose an Area of 1.0. The area under the curve (AUC) can be calculated to utilize the ROC curve for performance comparison. The AUC can be interpreted as the probability that the classifier correctly classifies a randomly chosen truly positive example and a randomly chosen truly negative example into the correct categories. This is best illustrated with a random classifier, where the ROC curve would be $y = x$. It follows that the AUC is 0.5, which is equivalent to a random ordering of samples.

$$FPR = \frac{FP}{FP + TN} \tag{2.1}$$

$$TPR = \frac{TP}{TP + FN} \tag{2.2}$$

The F-score (*cf.*, Equation 2.4) combines precision (*cf.*, Equation 2.3) and recall (*cf.*, Equation 2.2) into one metric and harshly penalizes extremes. α is the relative weighting of precision and recall, where recall is considered α times as important as precision. F_1 score means that recall and precision have the same weight.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2.3}$$

$$F_\alpha = \frac{1 + \alpha}{\frac{1}{\text{precision}} + \frac{\alpha}{\text{recall}}} \tag{2.4}$$

Another metric often mentioned by related work is classification accuracy, which is the total proportion of correct labels. For a binary classification model, Equation 2.5 is the formal definition of the metric.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FN + FP} \tag{2.5}$$

Chapter 3

Related Work

This chapter gives an overview of the work related to this thesis. Table 3.1 first provides an overview of the related work before elaborating on malware detection using VMI. The section about VMI is organized according to the hypervisor used by the study due to its impact on the selection of available introspection tools.

3.1 Overview

Because of the mentioned impact on the selection of available introspection tools, Table 3.1 organizes related research by the hypervisor used, mirroring the structure of this chapter. Grouping by hypervisor, we see five groups of papers: The first is built upon the work of [50] and uses BitVisor for malware detection. The groups of Xen and KVM share similarities in their choice of tools and represent the bulk of the research. Meanwhile, proprietary hypervisors like ESXi and Hyper-V attract little to no attention from the research community. Lastly, some researchers implement a custom hypervisor to conduct their research, similar to [50], allowing for highly tailored research.

3.2 Malware Detection Using Virtual Machine Introspection

Ever since the paper [58] introduced the idea of VMI, various research has been conducted on malware detection using VMI. Because of their different architecture and licensing, some hypervisors are better suited as an intrusion detection system than others, which is noticeable in the amount of research per hypervisor:

Table 3.1: Related Work

Reference	Hypervisor	Features	Detection Method
[51] – 2019	BitVisor	Storage Access	KNN
[52] – 2022	BitVisor	Storage + Memory Access	RF
[53] – 2022	BitVisor	Storage Access	LSTM
[11] – 2023	Xen	System Calls	DNN
[12] – 2021	Xen	System Calls	RF
[54] – 2021	Xen	Multiple	Ensemble Learning
[55] – 2018	Xen	Multiple	RF
[48] – 2021	KVM	Intel Processor Trace	CNN
[13] – 2020	KVM	System Calls	Ensemble Learning
[10] – 2020	KVM	System Calls	Policy
[56] – 2021	ESXi	Memory Forensics	Signature matching
[57] – 2024	Custom	Instructions + System Calls	Policy
[15] – 2022	Custom	-	-
This Thesis – 2024	HyperDbg	System Calls	RF, NB, IForest, LOF

3.2.1 BitVisor

BitVisor is a thin hypervisor that introduces the idea of a parapass-through architecture. This architecture lets most guest OS I/O access bypass the hypervisor and only mediates the minimal access needed for security. It also supports only one VM [50].

Building upon the work of BitVisor, [59] propose WaybackVisor: An extension of BitVisor, which automatically transfers all I/O logs of SATA drives to a Hadoop cluster. WaybackVisor intercepts all write operations by extending the ATA driver at the hypervisor level to send all intercepted information over the network to a Hadoop cluster. Leveraging WaybackVisor, [51] intercepted all write operations from WannaCry, TeslaCrypt and the benign Zip-software. As features, the authors extracted the entropy of sectors, the total amount of read sectors, the total amount of written sectors, the variance of Logical Block Address (LBA) in read requests, and the variance of LBA in write requests from the logs created by WaybackVisor and generate and evaluate ML models by using RF, SVM and KNN algorithms. Their models achieved an F_1 score of 0.98, with the KNN algorithm performing best.

To enhance WaybackVisor, [52] also add a logging functionality for low-level memory access patterns. They then collected memory access data for ransomware (WannaCry, REvil, Darkside), wiper malware (CaddyWiper), and benign software (AESCrypt, Zip, Excel, PowerPoint, Firefox). When creating and evaluating their ML models (RF, SVM, KNN), they discovered that they were able to classify running software with a F_1 score of 0.93 and detect malware with a F_1 score of 0.95 *without* needing to bridge the semantic gap.

To enhance research into the storage access patterns of ransomware and its use in ransomware detection, [60] published RanSap, an open dataset consisting of seven samples of ransomware, as well as five popular benign software samples. Using said dataset, [53] build Foreseer, a DNN that models the entire feature set using Long Short-Term Memory aided by attention mechanisms. Foreseer can project future events to cut the time to predict malware presence by 40%. The combined work of [53], [60] shows how publishing datasets can lead to more focused research.

The need for standard ransomware datasets is also outlined by [61]: Their survey found that most researchers download ransomware samples from public websites and build their required datasets themselves. Thus, the authors proposed the implementation of standard ransomware datasets, such as RanSap, as a potential research direction.

3.2.2 Xen

Contrary to BitVisor, Xen hypervisor can host multiple VMs. Various tools used for malware analysis build upon the Xen hypervisor, such as Drakvuf [62] and LibVMI. Because of the availability of these tools, Xen offers itself as a viable platform for malware detection research:

[11] developed DeepHyperv, a DNN detecting malware threats in a guest VM by doing memory introspection from Dom0. From the captured virtual memory contents, they extract program execution traces. A program execution trace, in this case, the high-level view, consists of features such as process identifiers (PIDs), system calls, and registry-related system calls. These features are extracted using various tools such as Drakvuf, which includes the tools LibVMI and Volatility [63]. The authors use Drakvuf to link system calls with PIDs by monitoring changes to the CR3 register, which tracks the location of the page directory table. This, in turn, detects process switches and bridges the semantic gap. Using these extracted features, they train and evaluate their model using various malware and benign executables and conclude that DeepHyperv achieves an accuracy of more than 91 %.

Using similar tools and methods, [12] extract logs of VM memory from the VMM. These memory logs are then parsed to system call logs using Drakvuf. From the system call logs, features are extracted as a Bag of n-grams. The researchers then utilize an RF model to detect malware, using the University of New Mexico (UNM) and BareCloud datasets to validate their model. While the UNM dataset is based on Linux system calls, the BareCloud dataset consists of Windows binaries. VmShield achieved an accuracy of 81.23% to 99.52% on UNM's Dataset and 97.66% to 99.91% accuracy on the BareCloud dataset. Although multiple studies use the UNM dataset to evaluate and train their corresponding models, the dataset can not be found anymore under the link cited by these studies, as it appears to have been taken down.

With the same set of tools, [54] extract multiple features from VM memory, the hypervisor layer, and the hardware layer (using perf as an additional tool). In total, they extract 235 events. To allow a classifier model to use these many features, they use ensemble learning combined with a Voting method. The model was evaluated using malicious samples obtained from VirusShare [64], which include various malware types, such as trojans, worms, and ransomware. Using these samples, the authors reach a F_1 score of 0.94.

In contrast to previous research using the Xen hypervisor, [55] detect malware without such a deep view inside the guest System. To achieve this, they extracted performance counters maintained at the hypervisor level, performance at the hardware level, and traces collected at the hypervisor. They collect 329 indicators using the following tools: perf, xenperf, and xentrace. These indicators are separated into their originating VMs, and the program behaviors of the VMs are inferred. Thus, the authors bridge the semantic gap without requiring detailed insight from tools like Drakvuf. The authors use over 2000 benign and malicious software executables from various sources to train and evaluate their model and achieve a detection accuracy of 0.875 on the trained RF classifiers.

3.2.3 Kernel-Based Virtual Machine

Like Xen, KVM also supports LibVMI, thus allowing a similar extraction method: [13] propose KVMInspector, which uses both LibVMI for virtual machine introspection and strace, to get system call logs from inside the guest VM. The features extracted at the guest OS and hypervisor levels are then used to detect various malware, including rootkits. The proposed approach uses an ensemble heterogeneous classifier where the outputs of different models are aggregated using a VotingClassifier. The model reaches an accuracy of up to 99.92% when evaluated on the UNM dataset.

Without needing to collect any information at the guest OS level, [48] utilize Intel Processor Trace (IPT) to collect information about the executions of a process. To start recording IPT packets, VMI is used to extract the identification information of the target process and then send it to the hypervisor for IPT configuration. After this, the control flow information is converted to color images, and a CNN detects malware with an accuracy of 95% when evaluated using malicious executables as well as benign programs.

Similar to [13], [10] monitor system calls in a KVM environment. However, the proposed application, called RansomSpector, does not need a component inside the guest VM and instead traps all system calls to the hypervisor, using LibVMI to resolve addresses, where file and network system calls are sent to the detector. This detector matches the observed system call patterns with known ransomware patterns. To evaluate their tool, the authors use a dataset of various types of ransomware obtained from VirusShare and VirusTotal [65]. The authors conclude that their tool can effectively detect ransomware attacks with a small performance overhead.

3.2.4 ESXi

While the hypervisors mentioned above are open-source, ESXi is a proprietary hypervisor developed by VMware. Although VMware is a significant player in the virtualization technologies market and even a market leader in the hyper-converged infrastructure market [66], very little public research on malware detection was conducted using its ESXi hypervisor. On VMware's security blog, [56] published an article outlining how memory forensics can be employed on an ESXi VM: When creating a snapshot of a VM on ESXi, it is possible to include the complete state of the guest VM, including the contents of its memory, which are saved into a file with the extension ".vmem." This memory snapshot was then analyzed using Volatility. To then detect malware presence in a snapshot, the authors suggest several approaches:

1. Signature matching using the YARA Language [67]
2. Using memory analysis by directly accessing the OS objects and identifying "unaccounted" components or suspicious modifications to the function addresses of a process
3. Comparing memory snapshots of a homogeneous system and identifying outliers

The authors have developed a plugin for Volatility to detect API hooks of malware trying to hide its presence. Using this plugin, they execute and analyze the evasive Thanos ransomware [68], which tries to hide its presence by letting other processes execute it using API Hooks.

3.2.5 Custom

Using Intel VT-x, it is possible to write a custom hypervisor. Various examples exist of custom hypervisors [69], [70]. The advantage of implementing a custom hypervisor is that intercepting instructions and system calls can be done from within the custom hypervisor. The authors of [57] use this to trap instructions and hook system calls to expose virtual environment-detecting behavior applied by evasive malware to change their behavior in VMs. Instructions are trapped as explained in section 2.1, while system calls are hooked by replacing the physical memory page of the system call issued by the guest in the Intel Extended Page Table with a fake page table, adding a logging functionality before returning to the original function. An advantage of the proposed approach is that, contrary to previous research, it is independent of different virtualization environments; however, to implement the proposed approach, for example, with Hyper-V, an in-depth understanding of the architecture of the hypervisor is necessary. To evaluate their approach, the authors used 23 samples of evasive malware and compared them to other detection methods, such as Drakvuf [62] and VMShield [12]. Most other methods failed to identify any of the evasive samples, with Drakvuf performing the best by detecting 56.52% of all samples. However, their model successfully identified all samples, achieving a 100% detection rate. Additionally, it detected threats more than six times faster than the other methods.

Similarly to [57], HyperDbg [15] uses Intel VT-x and Extended Page Table to leverage the powers of a hypervisor for debugging purposes. The implemented debugger can be used for high-performance and stealthy debugging of user and Kernel applications. It operates on ring -1 by virtualizing an already running system. To verify the stealthiness of the debugger, the authors tested the debugger with 13 packers and protectors, none of which detected the debugger. HyperDbg is open-source and features a VMX-compatible script engine and extensive documentation [71], making it a useful tool for various tasks. Using their debugger, the authors analyzed over 10'000 samples collected from a malware database [72] and found that their debugger enabled debugging of 22% more samples than WinDbg. Additionally, the debugger is 2018x faster than WinDbg for system call recording.

3.3 Possible Research Directions

Much related work has been conducted on the open-source hypervisors Xen and KVM, leaving little room for a new research gap. A possible issue in this part of the related work is that all related work using only system calls as features relies on the library LibVMI. Therefore, novel ways to extract system calls could be explored. In contrast to open-source hypervisors, proprietary hypervisors like ESXi and Hyper-V are rarely used in current research, thus offering a possible research direction. Additionally, a custom hypervisor could be leveraged as a ransomware detection system, as only [57] did this to catch virtual environment-detecting malware. While implementing a custom hypervisor would require a deep understanding of the Processor and the corresponding virtualization technology, such as Intel VT-x or AMD-V, plenty of custom hypervisors that could be used exist [15], [69], [70]. Related work also uses various detection methods, from simple policy-based detection to complex neural networks. However, none have used AD models to detect ransomware. Therefore, the following research directions have been identified and are explored in the next chapter.

1. A novel way to extract system calls for a ransomware detection system.
2. Use ESXi or Hyper-V as the hypervisor for a ransomware detection system.
3. Implement or modify a custom hypervisor to use as a ransomware detection system.
4. Use AD models to detect ransomware.

Chapter 4

Research Opportunities

This chapter examines the theoretical and practical feasibility of the research directions identified in the related work. First, the VMI tools and opportunities are listed per hypervisor in Table 4.1. This section should give the reader an overview of what is possible on each hypervisor. After this, the research directions outlined in Section 3.3 are combined to research opportunities with the capabilities identified in Section 4.1. These research opportunities are then explored and tested for feasibility. The research opportunity discussed in this thesis is selected to conclude this chapter.

4.1 Virtual Machine Introspection Capabilities

Table 4.1: Comparison of Hypervisors.

Extraction Method	**BitVisor**	**ESXi**	**Xen**	**KVM**	**Hyper-V**
LibVMI	×	×	✓	✓	×
Memory dumps	×	✓(Snapshot)	-	-	✓(LiveKd)
System calls	(✓)	×	✓	✓	✓(LiveKd)
Performance logs	×	✓	✓	✓	✓
IPT	×	×	✓	✓	?

4.1.1 BitVisor

This thin hypervisor is open-source and was developed as a research project [50]. It features a para-passthrough architecture designed to shrink hypervisor code size by letting most guest OS I/O operations bypass the hypervisor, only mediating the minimal access needed for security. To allow direct access from the guest device drivers, the para-passthrough architecture limits the number of VMs to one and can thus be seen as a thin layer between OS and Hardware. Because only one VM is possible, traditional VMI approaches are impossible with this hypervisor. Instead, virtual machine introspection is achieved by modifying the hypervisor, as [51], [52], [60] did. Therefore, extracting system calls should be possible with an approach similar to trap and emulate; however, it requires a deep understanding of the hypervisor and the guest OS and is therefore not considered a research opportunity.

4.1.2 VMware ESXi

The ESXi hypervisor is proprietary software developed by VMware. Contrary to BitVisor, it supports VMI; however, only over the NSX manager [73]. Because this is proprietary software designed to offload antivirus and anti-malware agent processing to a dedicated secure virtual appliance delivered by VMware partners, this work can't leverage ESXi's guest introspection capabilities. Alternatively, ESXi offers a wide variety of logs and an API over vSphere, over which performance metrics of guest VMs can be read [74]. As shown by [56], VM snapshots are configurable to contain a memory snapshot, which can be analyzed using a memory forensic tool, such as Volatility. Although creating a snapshot is slow, increasing the latency of memory introspection, creating snapshots would enable the system to be restored to the state before the malware was executed. Because ESXi is closed-source, extending the hypervisor with the required logging functionality is also not an option.

4.1.3 Xen

The Xen hypervisor is an open-source Linux Foundation collaborative project with a native VMI API [75]. It serves as the foundation for various malware analysis tools like Drakvuf. To monitor the low-level details of a running virtual machine, researchers can use LibVMI, a library that evolved from a research project called XenAccess [26]. Various tools and/or toolsets are available to bridge the semantic gap, such as Volatility or Drakvuf. Since version 4.15, Xen also allows the export of IPT data from a guest VM to tools in dom0, thus allowing for process introspection [76].

4.1.4 Linux KVM

Linux KVM allows turning a Linux Kernel into a Type 1 hypervisor [21]. With some additional configurations, KVM offers support for LibVMI [14], thus allowing researchers to monitor the low-level details of a VM easily. To bridge the semantic gap, tools similar to those available for Xen are accessible, except for Drakvuf. As this thesis is written, Drakvuf is uniquely available for the Xen hypervisor. Linux KVM also allows the collection of IPT from a guest VM as [48] demonstrated.

4.1.5 Microsoft Hyper-V

Hyper-V does not support any form of VMI. However, it can be debugged using LiveKd [77], which allows the extraction of a memory dump. Using LiveKd, extracting hypercalls and other hypervisor-level information should also be possible; however, this requires a more intricate setup [78]. Enabling IPT monitoring should be possible [79]. However, no official library for simplifying the collection of those traces on Windows was found in this work.

4.2 Exploring Research Opportunities

Several opportunities can be explored with the capabilities found in Section 4.1 and the possible research directions outlined in Section 3.3. These opportunities are summarized in Table 4.2, displaying the hypervisor, the introspection method and tools used, and the research directions the opportunity covers. Because performance logs were always combined with other features, and IPT requires a deep understanding of the Intel architecture, these two extraction methods were not considered in the research opportunities. For convenience, the research directions outlined in Section 3.3 are as follows:

1. A novel way to extract system calls for a ransomware detection system.
2. Use ESXi or Hyper-V as the hypervisor for a ransomware detection system.
3. Implement or modify a custom hypervisor for a ransomware detection system.
4. Use AD models to detect ransomware.

Table 4.2: Research Opportunities

Nr.	Hypervisor	Introspection Method	Tools	Research Directions
1.	Hyper-V	Memory dump	LiveKd, Volatility	2,4
2.	ESXi	Memory dump	Snapshot, Volatility	2,4
3.	Custom	System Calls	HyperDbg	1, 3, 4

Opportunity Nr. 1

To explore the first opportunity, the following setup was used:

- Windows Server 2022 Standard Hyper-V partition (L1) with 4056 MB of RAM and nested virtualization enabled.
- Windows 10 Pro partition (L2) with 1024 MB of RAM.

After installing LiveKd and Volatility3, a memory snapshot of L2 was taken on L1 using the command shown in Listing 4.1.

Listing 4.1: LiveKd Memory Snapshot

```
livekd64.exe -hv Win10L2
            -o C:\Users\Administrator\Desktop\test.dmp
```

To test the usability of this memory dump, the processes running in L2 were extracted in Listing 4.2.

Listing 4.2: Extracting Processes With Volatility

```
python .\volatility3\vol.py
    -f C:\Users\Administrator\Desktop\test.dmp
    windows.pslist
```

This approach produced an accurate list of processes running in L2, thus showing the usability of this dump file for VMI. However, extracting a memory dump took a long time (over an hour for 1024 MB), making this approach unpractical.

Opportunity Nr. 2

Because ESXi is proprietary software, an evaluation license was acquired to explore the second opportunity. This work tried to install ESXi on the test desktop (Intel i7-3770), but after the installation failed multiple times with the error: "No network adapters were detected," this opportunity was not further explored.

Opportunity Nr. 3

The third opportunity to develop a custom setup to analyze system calls was explored using the following environment.

- Host: Physical host environment (Windows 10, Intel i7-6700k).
- Guest: VMware Player VM, with nested virtualization enabled and added virtual serial port (Windows 10).

Listing 4.3: Using HyperDbg To Extract System Calls

```
output create SysCallLog file C:\Users\Lj\Desktop\SyscallLog.txt
output open SysCallLog
!syscall script { printf(
    "[%llx:%llx] Syscall num: %llx, arg1: %llx, arg2: %llx\n",
    $pid, $tid, @rax, @rcx, @rdx
);} output {SysCallLog}
# ... (Runs until Ctrl + c )
output close SysCallLog
```

The connection from host to guest was established according to HyperDbg's documentation [80], where the guest was the debuggee and the host the debugger. To test the capabilities of *HyperDbg*, system calls were extracted. Listing 4.3 shows how HyperDbg can be used to extract system calls from a VM, while Listing 4.4 displays the exemplary output of this command:

Listing 4.4: Extracted System Calls

```
[54c:151c] Syscall num: 48, arg1: c2cabf5a90, arg2: 1f0003
[54c:151c] Syscall num: 48, arg1: c2cabf5a90, arg2: 1f0003
[288:126c] Syscall num: f, arg1: 1dc, arg2: 1dc
[288:126c] Syscall num: f, arg1: 1dc, arg2: 1dc
[288:126c] Syscall num: f, arg1: 15c, arg2: 15c
[288:126c] Syscall num: f, arg1: 15c, arg2: 15c
[288:126c] Syscall num: 1b0, arg1: 28, arg2: 4f4de7f510
[288:126c] Syscall num: 1b0, arg1: 28, arg2: 4f4de7f510
[288:126c] Syscall num: 1a1, arg1: 1c, arg2: 9
[288:126c] Syscall num: 1a1, arg1: 1c, arg2: 9
[288:126c] Syscall num: 164, arg1: 4f4de7f210, arg2: 0
...
```

To resolve the system call number to the corresponding system call, we need context information about the operating system running on the client, such as the OS and version. Given this information, system call tables collected by the community like [81] can be leveraged to resolve the corresponding system call.

4.3 Discussion

Although multiple opportunities would be feasible, the decision was made to prioritize the speed of the extraction process. Consequently, the first two opportunities are impractical due to their longer extraction times. Using HyperDbg, the detection system gains a deep insight into the virtualized system while collecting information protected by the hypervisor with little intrusion on the inspected client. The opportunity combines three research directions, allowing the exploration of a novel way to extract system calls at the hypervisor level, using a custom hypervisor and AD models to detect anomalies based on the extracted features.

Chapter 5

Architecture

The purpose of this chapter is to give the reader an overview of the proposed system, as well as the scenario in which this proposed system must operate, before elaborating on the concrete implementation of the system in the subsequent chapter.

5.1 Scenario

This work uses HyperDbg as part of a sandbox to monitor a running executable. The goal of the sandbox is to determine whether a running executable is ransomware while avoiding harm to other systems or the network. To assess the maliciousness of software, this work uses HyperDbg to analyze the system calls issued by the executables. For this purpose, HyperDbg monitors and logs the entirety of the system calls issued by the system in a timespan while the executable is running. ML and AD models then use these system calls to determine whether ransomware was running on the system. To train and evaluate these models, this work considers various ransomware and benign samples. Benign samples include various office apps and procedures such as zipping and unzipping files.

5.2 System Overview

This work considered various setups to extract system calls at the hypervisor level using HyperDbg.

1. Debugging a physical machine over a serial port: This requires two Windows machines. If these two machines are physical, at least one must have a serial port to be debuggable.
2. Debugging a VM running on VMware Workstation: This setup would allow for only a single physical machine, but our test environment would operate on a Type 2 hypervisor using nested virtualization.

3. Debug the local machine and send data over a TCP socket: HyperDbg allows local machine monitoring. To extract the collected information, HyperDbg can create different output sources, including writing to files, named pipes, and TCP sockets. This work utilizes this capability to send the collected logs over a TCP socket to another machine in the network.

These setups have advantages and disadvantages: Setups one and two allow for more control over the debugged system because HyperDbg can operate in debugger mode. Debugger mode allows HyperDbg to break to the debugger and step instructions in kernel mode. While setup three is less intrusive, it operates in VMI Mode, which means that features such as breaking to the debugger, step instructions in kernel mode, and starting processes from HyperDbg are not possible [82]. This thesis proposes to use setup number three because HyperDbg's main purpose is to monitor. Debugging features are not required, so the advantage of being less intrusive is greater. An overview of the setup is provided in Figure 5.1:

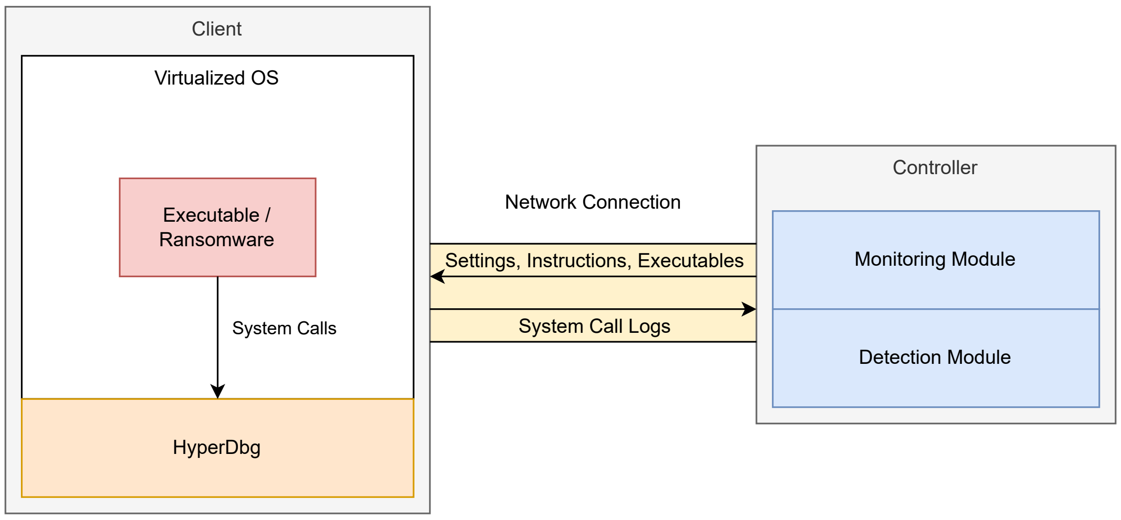


Figure 5.1: Proposed Architecture and Control Flow

This setup includes two systems with the following tasks:

- **Client**: This is the machine where HyperDbg runs and malware is executed. Because this machine is running malware, it should be as isolated as possible from other machines.

- **Controller**: This machine is connected to the client over the network. It is responsible for collecting the logs from the client and detecting whether ransomware is running on the client.

5.3 Logging System Calls

With HyperDbg, system calls can be logged comfortably: With the `!Syscall` command, HyperDbg can register an event, which triggers when Windows tries to execute a system call. This command unsets the Syscall Enable bit in the Extended Feature Enable Register, which lets system calls result in an undefined opcode exception. This exception can then be intercepted at the hypervisor level, where the system call is emulated and can be intercepted by the debugger [83]. At this point, context information of the system call can be extracted. To extract context information from the system call, [10] described which registers contain this information on the x64 platform: To get the number of the system call, the RAX register can be accessed. When a system call is invoked, the first four parameters are put into RCX, RDX, R8, and R9 registers, and the remaining parameters are pushed on the stack. Using HyperDbg, it is possible to access system state information easily: the variables $pname, $pid, $tid contain the values of the process name, the process id, and the thread id, respectively. These variables make it easy to deduct who issued a system call. Resolving the system call number to the corresponding system call will not be necessary for our detection system but is possible using system call tables collected by the community like [81]. The system call parameters could then be resolved to obtain more information about the purpose of the system call, for example, by using the types and structures defined in the `Winternl.h` header file. Resolving all system calls with their parameters requires more computation and memory to be accessed on the client machine, which this work tries to avoid. Because related work, except for [10], largely avoided computing the context information for system calls [12], [13], [84], this work tries to avoid it as well.

5.4 System Calls Preprocessing

To obtain services from the OS, a user program must make a system call, which invokes the operating system through a trap. The trap instruction switches from user to kernel mode and starts the operating system [19]. Because of this, various related work use system calls to detect malware using multiple approaches to process the raw system call logs to features. An overview of these approaches can be found in Table 5.1

Table 5.1: System Call Features Used By Related Work

Nr.	Preprocessing approach	Used In	Used Detection Algorithms
1	Bag-of-n-Grams, Frequency	[12], [13], [84]	RF, Ensemble Learning, DNN
2	Bag-of-n-Grams, TF-IDF	[84]	DNN

5.4.1 Bag-of-Words

This approach is inspired and commonly used by natural language processing. It converts a text document or a sentence into a vector of counts, where the vector contains an entry for every possible word in the vocabulary. If a word, such as ”Syscall,” appears three times in a document, the corresponding vector position for that word will have the value three [85]. In the context of system calls, this approach would count the occurrences of each unique system call in a trace. Because the Bag-of-Words loses the original sequence and structure of the analyzed text, the works that are listed next to this approach in Table 5.1 use the extension of this approach, which is called Bag-of-n-Grams: Instead of only counting the occurrences per word, the occurrences of a continuous sequence of n items is counted by a sliding window, which shifts by one to the right for each sequence. For example, the sentence ”System Calls are great” generates the 3-grams (three words per sequence): ”System Calls are”, ”Calls are great”. In related work, Bag-of-n-Grams was used in two ways (as depicted in Table 5.1):

1) Frequency: With this approach, a vector is created, where each n-gram is annotated with the number of occurrences, the same as in the Bag-of-Words approach. Thus, the vector is in the form of $V_{frequency} =< c1, c2, c3, ..., cZ >$, where Z is the number of unique n-grams and c is the number of occurrences of that particular n-gram in the trace.

2) TF-IDF: Term Frequency-Inverse Document Frequency (TF-IDF) not only considers the raw count of each n-gram, but the normalized count, where each word count is divided by the number of documents this word appears in [85]. In the context of system calls, this means that the frequency of a sequence of system calls is considered and how rare a sequence is in a collection of traces.

Choosing the optimal value for the n parameter in the Bag-of-n-Gram approach can be difficult. [86] provided a theoretical and experimental investigation, which sequence length is optimal for intrusion detection, and found that 6-gram and 7-gram have the best performance. [12], [13], [84] all use 6-gram to detect malware. To find the best configuration, this work will explore the performance of the detection models with various combinations of the approaches mentioned above and values for n. The results of this exploration will be documented in Chapter 6.

Chapter 6

Implementation

As mentioned, this chapter aims to give the reader a complete overview of the concrete implementation of the architecture proposed in Chapter 5. *HyperDtct* includes two physical machines: A client and a controller. The client is the machine on which HyperDbg monitors the execution of a sample, while the controller collects the logs generated by HyperDbg and instructs the client on how and what logs should be connected. This implementation makes use of the following platforms:

- Raspberry Pi 3 Model B+ with Ubuntu 22.04.4 LTS acting as controller machine.
- Intel i7-6700k PC with 16 Gigabytes (GB) RAM and 120 GB SSD running Windows 10 version 22H2 as the client machine.

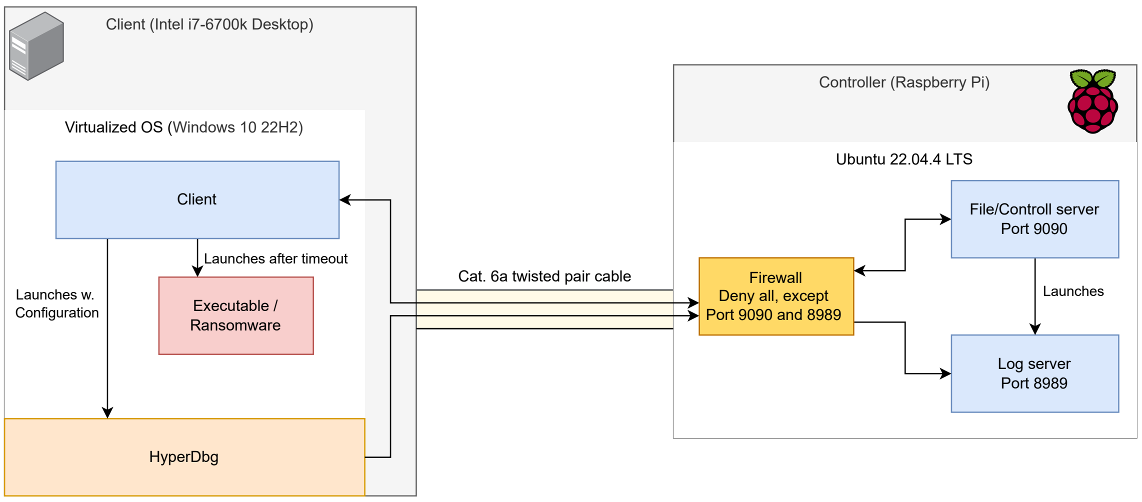

Client (Intel i7-6700k Desktop)
Virtualized OS (Windows 10 22H2)
Client
Launches after timeout
Executable / Ransomware
Launches w. Configuration
HyperDbg
Cat. 6a twisted pair cable
Controller (Raspberry Pi)
Ubuntu 22.04.4 LTS
File/Controll server Port 9090
Launches
Firewall Deny all, except Port 9090 and 8989
Log server Port 8989

Figure 6.1: Setup of Implementation

These platforms are interconnected as visualized in Figure 6.1. The client can be monitored from any machine, not just a Raspberry Pi. However, a Pi was used in this prototype to demonstrate that even a resource-constrained device can function as the controller.

The implementation for *HyperDtct* is available under [87]. The following sections describe these platforms and their corresponding components in depth. First, the client and how system call logs are collected using HyperDbg are elaborated. Then, the collection of logs is described from the controller's perspective before the assumptions underlying the described setup are stated, and the samples to train the models are elaborated. The last part of this chapter focuses on detecting malicious samples and evaluating these initial detection models.

6.1 Client

Setting up the client machine requires modification of the bare Windows setup: First, to allow ransomware to have some data to encrypt, the documents folder is populated with documents in various formats, such as PDF, HTML, and PNG, collected from [88]. Using `client/Utils/download_govdocs.py`, five archives were downloaded from [88], which equals roughly three GBs of files to encrypt. Additionally, the zip folder containing these documents is also placed in the documents folder to allow simulating unzip operations and have a larger target on the system for ransomware to encrypt. This implementation used version *0.8.2* of HyperDbg to monitor the client. At the time of writing, HyperDbg is still under active development and requires some configurations before it can be run to analyze the local system. This implementation followed this guide to set up the environment for HyperDbg [89].

In addition to the modifications suggested in the guide, Windows Defender was disabled because it was found that Windows' native antivirus software interfered with HyperDbg, as it led to a system crash each time it scanned the system concurrently with HyperDbg running. Therefore, this work disabled all components related to Windows Defender. As the client is part of a sandbox isolated from the rest of the network and *HyperDtct* requires malware to run uninterrupted to collect data for training, this modification is necessary in any case. The following steps were followed to disable Windows Defender.

1. Press Win + R, type `gpedit.msc`, and press Enter.
2. Computer Configuration > Administrative Templates > Windows Components > Microsoft Defender Antivirus
3. Select policy "Turn off Microsoft Defender Antivirus" and enable it.

Disabling Windows Defender reduced the occurrences of a system crash when logging system calls. However, it did not completely avoid them. To launch HyperDbg, Driver Signature Enforcement (DSE) must be disabled on Windows. While various approaches exist to achieve this, this work used the following approach:

1. Press and hold the shift key while clicking the restart button.
2. This opens the recovery settings. Select Troubleshoot > Advanced Options > Startup Settings and click Restart

3. The system reboots. Upon reboot, several options are presented. Press the key F7 to disable DSE.

This approach temporarily disables DSE on the system and must be executed again after each reboot. HyperDbg must be launched using the highest privileges and thus requires `client/start_logging.py` to be run as Administrator. To allow this while at the same time executing the potentially malicious samples with an unprivileged user, the client machine is set up with two user accounts:

- *Client*: This is the unprivileged user in whose session all samples are executed.
- *ClientAdmin*: The administrative user responsible for launching HyperDbg.

6.1.1 Monitoring Procedure

HyperDtct provides a wrapper for HyperDbg to collect system call logs: To start monitoring system calls on the client, run the script `client/start_logging.py` in the unprivileged user's session, entering the password of the administrative user. The script aims to provide an entry point to start logging system calls. It first checks whether HyperDbg is runnable by executing HyperDbg and trying to catch a single system call. Upon success, the script tests the connection to the controller machine. If these two prerequisites are successful, it starts the logging procedure. To log the system calls for a single executable, the following steps are executed:

1. Request the next file from the controller. The file is expected to be a zip file containing a batch script named `execute.bat`.
2. Extract the zip file.
3. Request the next log socket from the controller. The answer to this request contains information such as how long the log duration is expected to be and whether the file is expected to be malicious.
4. Generate a HyperDbg script based on the information received above.
5. Start HyperDbg with the generated script
6. Start `execute.bat` in the extracted zip file as the unprivileged user

The process is further outlined in Figure 6.2 as a sequence diagram, focusing on the sequence from the client's perspective.

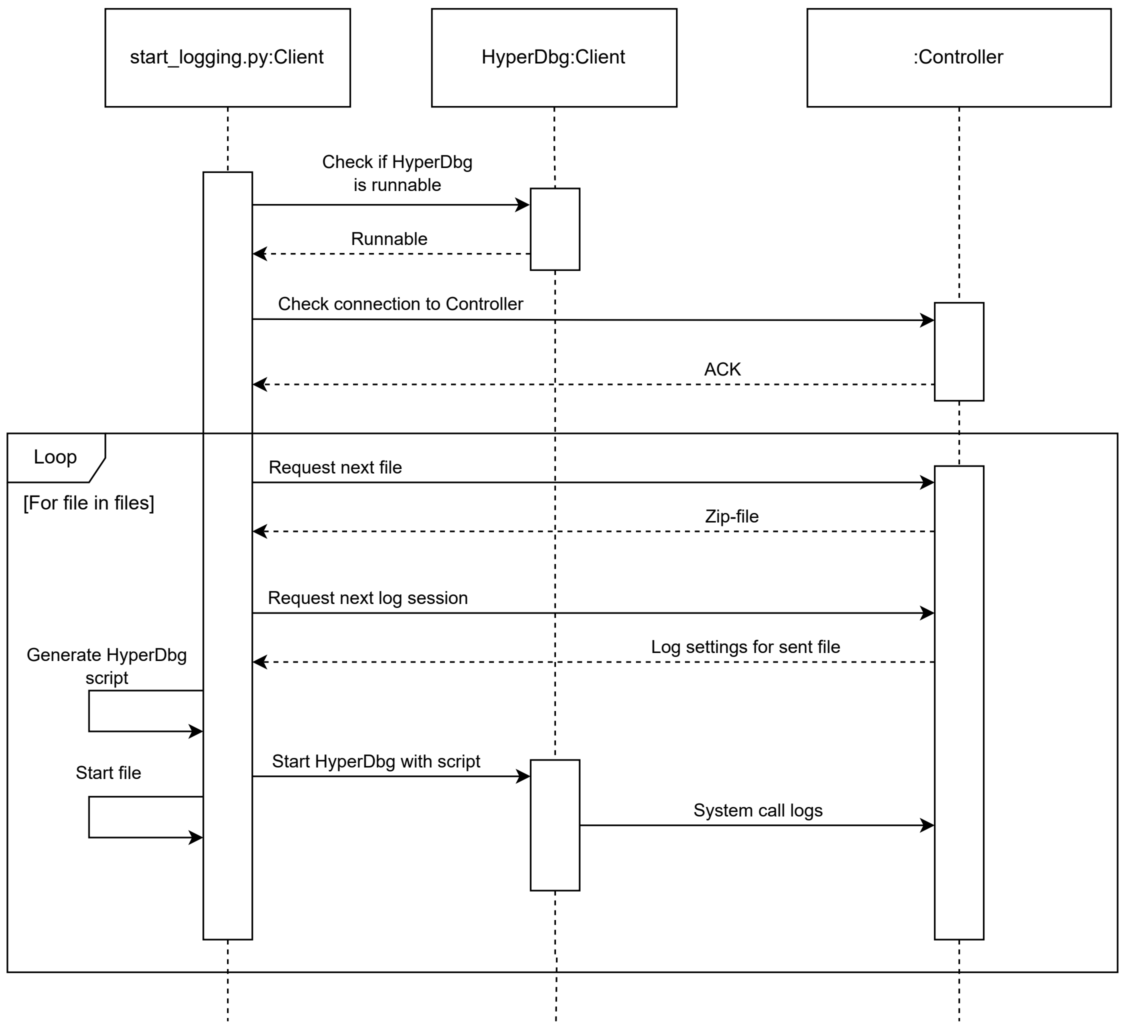


Figure 6.2: Log Collection Sequence from the Client's Perspective

6.1.2 Monitoring System Calls with HyperDbg

The generated script is based on a template stored in the directory `client/HyperDbg/ds_templates`. Listing 6.1 provides an example of such a generated script. First, HyperDbg connects to the local machine and loads its drivers and kernel modules. Then, HyperDbg builds its symbol table and opens the TCP connection to the controller. This logs PID, thread ID (TID), system call number, and the first four system call arguments as comma-separated values to the controller for the time specified by the controller. This part of the script was adapted from [90]. Finally, after the defined duration has elapsed, HyperDbg closes the connection to the controller and exits.

Listing 6.1: Logging System Calls in HyperDbg

```
# Connect to local machine and load vmm module
.connect local
load vmm

# Configure symbols, load them once, and then use only the local path
.sympath SRV*c:\Symbols
.sym reload

# Log output to controller socket
output create SysCallLog TCP 192.168.1.2:8989
output open SysCallLog

# Log all system calls in the following format
# pname,pid,tid,syscall,rcx,rdx,r8,r9\n
!syscall script {
    printf("%s,%x,%x,%llx,%llx,%llx,%llx,%llx\n",
    $pname, $pid, $tid, @rax, @r10, @rdx, @r8, @r9);
} output {SysCallLog}

# => Collect for n milliseconds (hex)
# => 927C0 == 600'000 ms == 10 minutes
sleep 927C0

# Close output and exit logger
output close SysCallLog
unload vmm
exit
```

6.2 Controller

The controller is a machine connected to the client over the network. Because the client is a sandbox, this work tries to isolate it as much as possible from the controller. Therefore, using Ubuntu's uncomplicated firewall (UFW), all connections to the controller are denied except connections on the configured controller socket (Port 9090) and log socket (Port 8989). A separate log socket is created because HyperDbg closes the socket connection after successfully logging all system calls to it. To start the monitoring server on the controller, the script `controller/start_server.py` can be run. The script accepts two arguments, `-input` and `-log_dir`. Input is either a zip file with the name format `[malicious|benign]_[file_name]_[duration]min.zip` or a directory, which contains a `settings.json` file, outlining in what order and with which settings the files in the directory are sent. The log directory is the directory where the collected logs will be output. For each file to be sent, the following settings can be adjusted in the file `settings.json`:

- *file*: The name of the zip file to be sent. This file is expected to be in the same directory as `settings.json`.
- *malicious*: Whether the executable in the zip directory is expected to be malicious.
- *minutes*: For how many minutes the file should be monitored.

- *name*: The name of the sample to be collected.
- *requires_admin*: Whether the executable must be launched with administrative privileges.

After the script is started, the controller socket accepts the following requests:

- *next_file*: Send the next file of the defined files. If all files were sent, the controller sends a special command, signaling to the client that all files were sent.
- *next_log*: Creates a new log socket for HyperDbg to connect to and sends the settings related to the expected executed file.
- *test_connection*: Sends a simple ACK, confirming the connection is working.

Because HyperDbg is not entirely stable and can crash the client unexpectedly, the controller must gracefully handle this sudden loss of connection. Therefore, to avoid the scenario where the controller waits endlessly on the client's input, a timeout of 60 seconds is set before the controller closes the connection and waits for new connections.

6.3 Assumptions

This work makes several assumptions for collecting system call logs using HyperDbg: First, it is assumed that ransomware cannot reboot or crash the system, as this would lead to HyperDbg not running anymore because DSE would be enabled again. Second, *HyperDtct* assumes that ransomware cannot infect the controller over the configured controller socket or log socket. In Chapter 7, these assumptions are evaluated and tested for different ransomware.

6.4 Collected Samples

System call logs for several scenarios and executables were collected using the explained setup.

6.4.1 Benign Software

To collect data and evaluate the system, this work used benign applications from [91], which distributes free applications for Windows, specifically packaged for portability. The platform is fully open-source, free, and maintained by over a hundred developers, translators, application packagers, designers, and release testers. The advantage of portable applications in the context of this work is that these executables can be sent to the client the same way ransomware would be, without requiring an installation process. For the process of initial data collection, the following samples were considered with the corresponding monitoring duration:

- executing LibreOffice Calc (only executed, no user interaction): 5 minutes
- LibreOffice Draw (only executed, no user interaction): 5 minutes
- LibreOffice Writer (only executed, no user interaction): 5 minutes
- installing 7zip: 2 minutes
- installing and uninstalling LibreOffice: 4 minutes
- copying all PDFs from the documents directory to another directory: 3 minutes
- unzipping files script: 5 minutes
- zipping files script: 5 minutes

These samples were packaged into a directory containing an `execute.bat` script, defining how the executable should be run, and compressed to a zip file. Each such script is responsible for ensuring that the software only runs during the defined monitoring period. To monitor the executable, it is executed on the client using the mentioned `execute.bat` script and run for the defined time. This process happens automatically, and no input from a user is required.

6.4.2 Ransomware-PoC

Ransomware-PoC is a simple proof-of-concept, open-source ransomware and available under [92]. It generates an AES key to encrypt local files in a starting directory, which can be provided by the user when executing the malware. Then, files with specific extensions, which are hard-coded in the source code, are encrypted and their names are appended with .wasted. The AES key is encrypted and displayed to the victim using a hard-coded RSA public key. Luckily for the victim, this software allows the user to decrypt the files again using the hard-coded private RSA key.

Ransomware-PoC was run twice for 10 minutes and encrypted files in the documents folder, populated with files collected from [88].

6.5 Detection

The detection module leverages the Python packages scikit-learn [93], pandas [94], and pyOD [95]. After preprocessing the collected logs using pandas and scikit-learn, the extracted features are used to train and evaluate the following algorithms:

- *RF*: RF is the algorithm most commonly used in related work. Thus, it is appropriate to test its performance in the context of this work.
- *NB*: NB is an alternative, simpler classifier to the previously selected RF.

- *IForest*: Building on the success of the RF algorithm, the IForest is selected as the AD alternative to the classifier RF.
- *LOF*: The LOF algorithm is selected to evaluate density-based ML methods on the collected data and to balance the number of classifiers and AD models considered.

Thus, the prototypical implementation of *HyperDtct* considers two anomaly detectors and two classifiers to detect malicious samples. The performance of each model is evaluated at each stage of development, and the models, as well as their corresponding hyperparameters, are subject to change, should their performance disappoint during evaluation.

6.5.1 Preprocessing and Training

The collected log files are read into a pandas DataFrame. Based on the file's name and the issuing process's name, all system calls are labeled as benign or malicious, based on the issuing process name. Only the system call number is considered, while the attributes of the system call are ignored. The system call numbers are grouped by timestamp and PID. *HyperDtct* only considers one client, thus logs are not collected concurrently from multiple machines. Because of this, grouping by timestamp and PID creates a unique index. Additionally, the timestamp is rounded to a certain time interval, such as five seconds, which results in a list of system calls issued in five seconds by a process. This process is outlined in Listing 6.2. During the implementation phase, this work explored several time intervals; however, no difference was found in model performance. To satisfy both the need for sufficiently long timestamps that allow normal software to issue enough system calls within this timeframe and the need for enough samples to train and evaluate the models, a timeframe of five seconds was chosen.

Listing 6.2: System Calls are Grouped by PID and Timestamp

```python
def group_by_pid_and_timestamp(self, df, floor='5s'):
    df['timestamp'] = pd.to_datetime(df['timestamp'], utc=True)
    df['timestamp'] = df['timestamp'].dt.floor(floor)
    df = df.drop(columns=['pname', 'tid', 'rcx', 'rdx', 'r8', 'r9'])
    grouped_df = df.groupby(['pid', 'timestamp']).agg({'syscall':list, 'malicious':
        'first'})
    return grouped_df
```

This list of system calls issued by a process in five seconds is then converted to a space-separated string of system call numbers, such that it can be used with the string-based bag-of-n-grams approach. In total, over 103'000 system calls were collected across the mentioned samples, over 27'000 of which stemmed from malicious samples. By grouping by PID and five seconds, 5230 benign and 241 malicious dataset entries are available for training and evaluation of the detection models.

The preprocessed data is used to train all models, allocating 80% for training and reserving 20% for evaluation. Using this split, only 56 malicious samples are available in the evaluation dataset, which must be considered during model and vectorizer evaluation.

Using the script `controller/detection/train_models.py`, the following combinations of algorithms, vectorizers, and n-grams are trained and evaluated:

$$\{\text{NB}, \text{RF}, \text{IForest}, \text{LOF}\} \times \{\text{TF-IDF}, \text{Frequency}\} \times \{1, 2, 3, 4, 5\}$$

Although [12], [13], [84], [86] have found 6-gram and 7-gram to have the best performance in detecting malware, they are not used during this stage of development. Because the dataset collected at this stage is neither large nor diverse, only smaller n-grams are considered. The AD models are trained with the hyperparameter *contamination* set to 0.044, as this is the contamination factor of anomalous entries in the collected data. After training, the models are serialized as pickle objects and saved. Evaluation scores and training durations for all models are saved to a comma-separated-values file. To extend the implementation with more models and preprocessors, these objects are handled using abstract classes. Thus, it is easy to extend the implementation should the need arise in the evaluation.

6.5.2 Model and Vectorizer Evaluation

As outlined in Section 7.1, this work uses the F_1 score to compare models. The models were initially fitted and evaluated without adjusting Hyperparameters, except setting *contamination* to 0.044 in the AD models. This led to the F_1 scores depicted in Figure 6.3. While the classifiers achieved high F_1 scores of over 95 % across all vectorizers and n-grams, the AD models did not achieve high scores. The notable exception was the LOF model, which showed considerably better performance when trained on frequency-based n-grams. Specifically, the LOF model trained on a frequency-based 3-gram dataset achieved the highest F_1 score of 76 %.

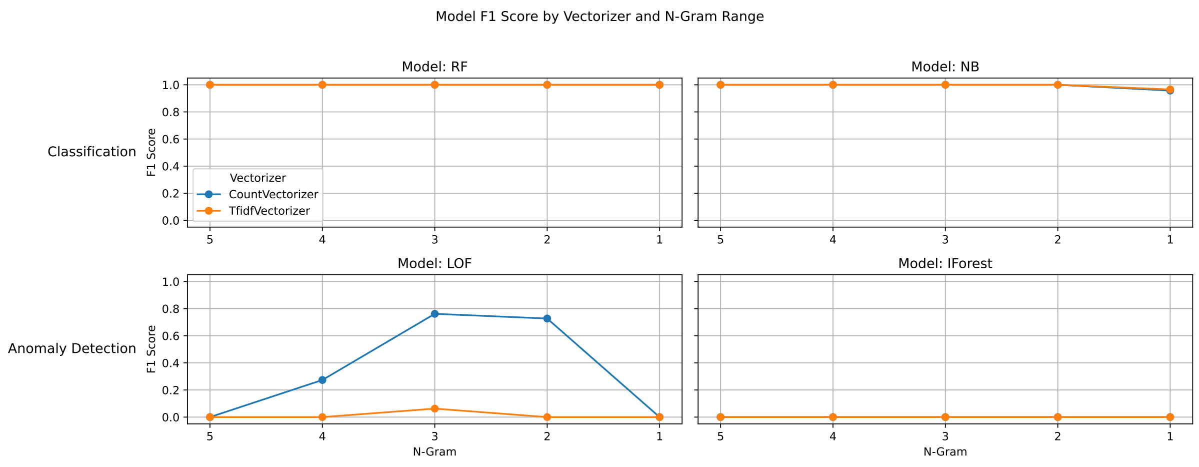


Figure 6.3: F_1 Score of Models Without Hyperparameter Tuning

Therefore, hyperparameter tuning was attempted to improve the performance of the AD models. During performance tuning, the vectorizer and the n parameter of the Bag-of-n-Gram approach were considered as parameters. With the contamination fixed to 0.044, the parameter grid in Listing 6.3 was evaluated to improve models for the IForest algorithm.

Listing 6.3: Parameter Grid Considered for the IForest Algorithm

```
# Define the parameters to be tuned for the IForest algorithm
param_grid = {
    'ngram': range(1, 6), # [1,2,3,4,5]
    'vectorizer': [CountVectorizer, TfidfVectorizer],
    'n_estimators': [50, 100, 200],
    'max_samples': ['auto', 0.5, 0.75, 1.0],
    'max_features': [0.5, 0.75, 1.0],
}
```

According to [96], *n_estimators* sets the number of base estimators in the ensemble (with the default being 100), *max_samples* defines the number of samples while *max_features* defines the number of features to draw from X to train each base estimator. With those hyperparameters considered and the combination of n and vectorizers, a total of 270 combinations of parameters are tested.

To improve the LOF algorithm models, the parameters outlined in Listing 6.4 were considered.

Listing 6.4: Parameter Grid Considered for the LOF Algorithm

```
# Define the parameters to be tuned for the LOF algorithm
param_grid = {
    'ngram': range(1, 6), # [1,2,3,4,5]
    'vectorizer': [CountVectorizer, TfidfVectorizer],
    'n_neighbors': [10, 20, 50],
    'algorithm': ['auto', 'ball_tree', 'kd_tree'],
    'metric': ['euclidean', 'manhattan', 'minkowski'],
}
```

n_neighbors is the number of neighbors to use by default for 'kneighbors' queries, *algorithm* defines the algorithm used to compute the nearest neighbor, and *metric* defines how the distance to the neighbors is measured [97]. Evaluating this parameter grid results in a total of 270 combinations to be tested.

To perform the tuning process and identify the optimal parameter combinations, the script `controller/detection/gridSearch.py` was executed with the corresponding model passed as an argument. The best F_1 scores from these runs and their respective configurations are presented in Table 6.1.

Table 6.1: Best Hyperparameters and Vectorizers per Algorithm

Algorithm	***Vectorizer***	***N-Gram***	***Hyperparameters***	F_1 ***score***
LOF	Frequency	3	'n_estimators': 200, 'max_samples': 1.0, 'max_features': 1.0	0.81
IForest	TF-IDF	2	'n_neighbors': 10, 'algorithm': 'auto', 'metric': 'euclidean'	0.81

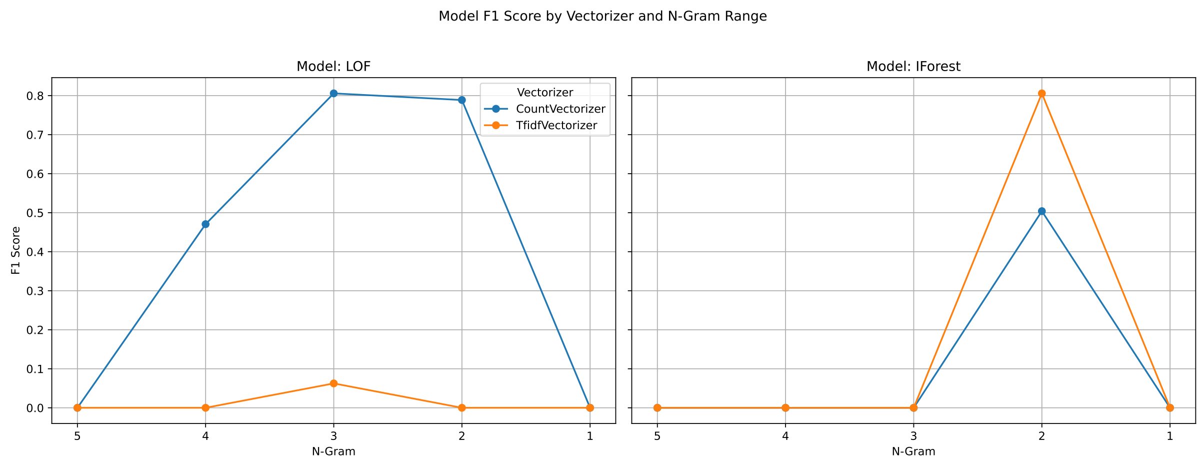


Figure 6.4: F_1 Score of AD Models with Improved Hyperparameters

Compared to the previous results in Figure 6.3, Figure 6.4 depicts the F_1 score across different n-grams using the AD models with the aforementioned, updated hyperparameters.

While the results for the LOF models have not improved remarkably, tuning the hyperparameters of the IForest algorithm improved both results of the frequency-based and TF-IDF-based 2-gram data and led to a model with an F_1 score of over 80 %. Despite these improvements, the classifier models still perform better than the AD models. Table 6.2 gives an overview of the best results achieved per algorithm and its evaluation metrics:

Table 6.2: Best Results per Algorithm

Algorithm	***Vectorizer***	***N-Gram***	F_1 ***Score***	***Precision***	***Recall***
NB	Frequency	5	1.0	1.0	1.0
RF	TF-IDF	5	1.0	1.0	1.0
IForest	TF-IDF	2	0.81	1.0	0.67
LOF	Frequency	3	0.81	1.0	0.67

Chapter 7

Evaluation

This chapter evaluates the sandbox prototype. Therefore, each section describes additional samples collected and potential modifications made to the system. Each section builds on the previous sections and considers the changes made in earlier evaluations.

7.1 Comparing Models

Because the data collected in this work is highly unbalanced, with only a minority of entries being malicious, choosing the right performance metric to evaluate models is important. Several performance metrics have been described in Section 2.5.1, making up the metrics this work can select from. Classification accuracy is a misleading metric in this work, as a model detecting not a single malicious entry would still achieve a high score because of the data's inherent high percentage of benign entries. A better metric would be the F_1 score, a harmonic mean of precision and recall. This metric provides a more accurate assessment of a model's performance in detecting malicious entries, highlighting models that not only detect a high proportion of malicious entries but also do so with a high degree of accuracy. Therefore, this work relies mainly on the F_1 score to compare models.

7.2 V2: Ransomware Collection

Only two samples of the same ransomware were considered for training the models during implementation. Thus, more malicious samples are needed. However, collecting data from malicious samples using the existing monitoring system is inefficient, as the client requires a manual reset after each sample. This section elaborates on how this work efficiently collected more malicious samples before elaborating on what additional samples were collected and how the model performance was affected.

7.2.1 Efficient Data Collection

To improve data collection efficiency, the system needs to be updated in two ways: to enable automatic client recovery after a malware infection and to handle potential client crashes. These updates will involve making changes to both the client and controller machines.

Changes to the Client

Because the considered prototype does not use an established VMM but a custom hypervisor, snapshots are not supported, and another approach must be found to reset the system. Windows provides functionalities to create backups for files and entire volumes, such as the command-line tool `wbadmin`, but not all functionalities of such tools are available for every version of Windows. On the client machine running Windows 10 version 22H2, creating a backup using `wbadmin` was supported, but restoring it was not. Another approach would be implementing a tool to automatically use the control panel GUI to restore the system, similar to how [15] restored their systems during evaluation. Because the malware samples considered in this version do not attempt to escalate privileges and only encrypt the user's folder without affecting the state of the operating system beyond a reboot, a more straightforward restore functionality was implemented. `client/System/recovery.py` recovers the encrypted files using compressed backup files stored on a drive only accessible with administrative privileges. Archive file formats considered include *.tar.gz*, *.zip* and *.7z*, among which the archive created using 7-Zip [98] was found to be the fastest, restoring a directory containing 63 GB of files in less than thirteen minutes. The command shown in Listing 7.1 was used to create a backup archive of a folder.

Listing 7.1: Create a File Backup of the Archive Directory using 7-Zip

```
7z.exe a -mx1 D:\FileBackup\Documents.7z .\Archive\
```

To restore the compressed backups, the algorithm in Listing 7.2 restores all archives contained in *FILE_BACKUP_PATH* to their original location, specified by the name of the archive. The algorithm assumes that all encrypted data can be found within one directory *RECOVERY_PARENT_DIR*, such as the user's home directory. Because this recovery method was enough to restore the system to automate data collection for the samples considered during this version, a recovery method to restore the operating system is deferred to later versions or for future work.

Listing 7.2: File Restoration Algorithm

```
zips = [
    f for f in os.listdir(FILE_BACKUP_PATH) if f.endswith('.7z')
    ]
for zip_file in zips:
    zip_path = os.path.join(FILE_BACKUP_PATH, zip_file)
    recovery_dir = os.path.join(
        RECOVERY_PARENT_DIR, os.path.splitext(zip_file)[0])
    print(f"Recovering {recovery_dir}...")
    if not os.path.exists(recovery_dir):
        print(f"{recovery_dir} not found, creating...")
        os.makedirs(recovery_dir)
    else:
        # Remove existing files
        remove_files_in_dir(recovery_dir)
    os.system(f'"{SEVEN_ZIP_PATH}" x {zip_path} -o{recovery_dir}')
```

Two obstacles must be addressed to continue logging after a potential crash of the client machine automatically: The script `client/start_logging.py` must be launched automatically with administrative privileges on client startup, and DSE must be disabled without requiring manual intervention. The problem of automatically launching the script `client/start_logging.py` with administrative privileges can be solved using the Taskschedule functionality in combination with the Autologon functionality provided with Windows. To automatically login to the privileged user, the following values were modified in the Windows Registry Editor in the key `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon`:

- `AutoAdminLogon`: 1
- `DefaultUserName`: ClientAdmin
- `DefaultPassword`: Password of ClientAdmin

The necessary configuration to automatically run the monitoring script with administrative privileges is documented in the script `client/System/setup_logging_task.ps1`, which can be run to apply these configurations. After these two configurations, the system automatically continues monitoring after a restart or potential crash.

To disable DSE after a potential crash, the method to disable DSE was updated: Instead of manually disabling DSE temporarily using the startup menu, the client machine boots on a thumb drive set up with EfiGuard [99]. EfiGuard is a portable x64 UEFI bootkit that patches the Windows boot manager, boot loader, and kernel at boot time to disable PatchGuard and DSE. Using EfiGuard, DSE can be disabled from the command line using `EfiDSEFix.exe` and thus can be disabled without an additional restart after a potential crash. To avoid malware modifying or encrypting the contents of EfiGuard, the thumb drive's access rights are set so that no user can access it.

These modifications update the assumptions of the prototype outlined in Section 6.3. The first assumption that ransomware cannot reboot or crash the system is relaxed, as in version V2, a reboot or crash can be handled as long as the monitoring task starts up again successfully. Additionally, compared to the previous assumptions, it is now assumed that ransomware does not elevate its privileges and thus has no access to the backups or the thumb drive containing the EfiGuard bootkit.

Changes to the Controller

To make logging more manageable, the scripts `controller/V2.sh` and `controller/V2-1.sh` summarize the entire process, including mounting an external thumb drive to store the logs, generating the input zip files and starting the server with the standard user's privileges. After the logs for all samples have been collected, the thumb drive is unmounted again, allowing the extraction and analysis of the logs on a secure machine. The recovery method for a sample is configured in the file settings under the key *recovery*. The system does not recover by default, and no actions are taken after successfully monitoring a sample. To restore the client files after a sample was run, the configuration *recovery* can be set to *client_files*. For V2, this is the only possible recovery method and is communicated to the client machine over the log settings for the sent file.

Unlike the version described in Chapter 6, this version of the controller must function without any user intervention and, therefore, must detect a potential crash of the client machine and be able to handle the client's requests again after it has recovered. To enable configuration of the behavior of the controller depending on the sample causing the crash, the option *on_crash* was added to the file settings with the following possible configurations:

- *retry* (default): If a crash occurred while monitoring this file, try to monitor it again. This option is best used for samples where a crash is unexpected.
- *skip*: Skip monitoring this file again in case of a crash. Using this option may result in an incomplete log.
- *retry_append*: To avoid having to monitor the entirety of the configured time again while still having logs of the configured duration, this option starts appending to the existing log and, after the client recovered, monitors the sample for the remaining time.

7.2.2 Collected Samples

The data for V2 was collected in two iterations, with each sample being run twice. In the first iteration (V2), the samples were run on 20 GB of files to encrypt, while in the second iteration (V2-1), 60 GB of files were available. V2 was run for a shorter time and already featured the client file-recovery implementation. Because several samples crashed the client during the first iteration, the system was modified to handle crashes as described above. However, no crashes were observed during V2-1. Table 7.1 gives an overview of the considered samples, their settings, and which samples crashed the client in the first run. The collected ransomware samples are briefly introduced in the following subsections.

Table 7.1: Overview of Samples and Settings Considered in V2

Sample	Malicious	Minutes V2	Minutes V2-1	V2 crash
Cry [100]	✓	10	20	×
RAASNet-PyCrypto [101]	✓	10	20	×
RanSim [102]	✓	10	20	✓
JavaRansomware [103]	✓	15	30	×
RAASNet-PyAES [101]	✓	30	60	×
RanSim-Slow [102]	✓	30	60	✓
Roar-AES-CTR [104]	✓	60	60	✓
Roar-ChaCha20 [104]	✓	60	60	×
Windows Defender Scan	×	5	5	×

1. Cry

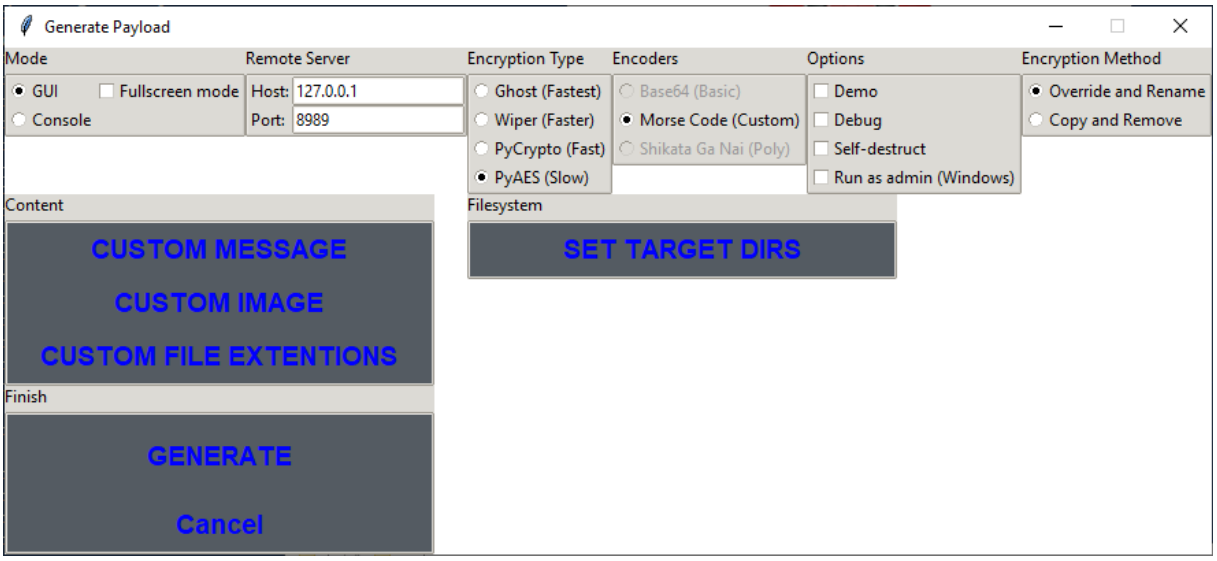
Cry [100] is an open-source, educational ransomware written in Go. It consists of two executables, `cry.exe` and `web.exe`. The first is the payload executed on the victim's machine, while the second is the control server, which would be executed on a different machine in a real scenario. To allow monitoring of the payload, the control server was executed with the payload on the client machine. The payload was modified to include PDF, GIF, PS, and ZIP file types to encrypt a higher proportion of the files. Additionally, Cry was made more malicious by not just creating an encrypted copy of a file but instead encrypting the file itself (as crypto-ransomware usually does).

RAASNet

To avoid the required login and update some deprecated imports of RAASNet, this work forked the repository from [101], a fork of the original, banned repository. The repository with the mentioned changes used in this work can be found under [105]. RAASNet demonstrates how RaaS allows non-technical users to create custom ransomware payloads, customizing the command and control server settings, the encryption method, and the displayed messages. All this can be done from a graphical user interface without requiring the attacker to write or edit a single line of code. It also allows the attacker to launch a command and control server to collect the private encryption keys. The payload can be created in the menu depicted in Figure 7.1.

While several settings can be customized, this work edited the following settings to generate the payloads:

- *Set Target Dirs*: This setting was modified to encrypt only the user's documents directory.
- *Encryption Type*: Besides crypto-ransomware, RAASNet could also distribute wiper malware. However, this work only generated payloads using the PyCrypto and PyAES encryption types.


Generate Payload

Mode: [✓ GUI] ☐ Fullscreen mode / Console

Remote Server: Host: 127.0.0.1 / Port: 8989

Encryption Type: Ghost (Fastest) / Wiper (Faster) / PyCrypto (Fast) / [✓ PyAES (Slow)]

Encoders: Base64 (Basic) / [✓ Morse Code (Custom)] / Shikata Ga Nai (Poly)

Options: ☐ Demo / ☐ Debug / ☐ Self-destruct / ☐ Run as admin (Windows)

Encryption Method: [✓ Override and Rename] / Copy and Remove

Content: CUSTOM MESSAGE / CUSTOM IMAGE / CUSTOM FILE EXTENTIONS

Filesystem: SET TARGET DIRS

Finish: GENERATE / Cancel

Figure 7.1: RAASNet Menu to Create Custom Ransomware Payloads

Using PyInstaller [106], RAASNet compiles executables from the generated payload, which can be easily distributed and executed.

RanSim

RanSim [102] is a ransomware simulation script written in PowerShell. It recursively encrypts files in the target directory using 256-bit AES encryption and can be downloaded from [102]. Similar to Ransomware-PoC, RanSim can be configured to encrypt files in a target path and also provides the functionality to decrypt them again. When executing this academic ransomware on a Windows system, it is essential to turn off Windows Defender Antivirus, as it recognizes the malware and tries to quarantine it. This work modified RanSim to encrypt files in the user's documents directory and included the file extensions '.html,' '.xml,' '.ps,' '.zip,' and '.gz' so that more files are encrypted. This work also considered an alternative version of RanSim named RanSim-Slow, which slows down file encryption by adding a two-second timeout between each file.

JavaRansomware

JavaRansomware [103] is another educational, open-source ransomware developed in Java. It encrypts files in the background using AES-256, a robust encryption algorithm, with an RSA-4096 Public Key to secure the AES symmetric key and store it in a local database. The file's contents are sent to a command and control server, which runs locally for demonstration purposes. The ransomware supports encryption and the decryption of files.

Roar

Ransomware Optimized with AI for Ressource-constrained devices (ROAR) [104] tries to hide from behavior-based detection techniques by employing reinforcement learning (RL) to enhance its impact on target devices while evading detection methods through behavioral fingerprinting-based anomaly detection. To achieve this, Ransomware-PoC [92] was adapted to be configurable at runtime regarding the encryption algorithm used, the encryption rate, and settings regarding encryption bursts. An encryption burst is a repeating phase in the encryption process of an ongoing ransomware attack, with two key components to its configuration. First, the burst duration is specified, either as a fixed number of seconds or a certain number of files to be encrypted. Second, a pause can be configured between bursts, in which the ransomware is idle and not encrypting files. The anomaly detection system used by [104] to train and evaluate ROAR considers performance metrics, CPU temperature, and network metrics to detect ransomware. [107] extended ROAR by considering system calls as the anomaly detector's basis for detecting anomalous behavior. This work considered both the results of [104] and [107]. While the configuration was not changed at runtime, the considered samples and configurations of this ransomware are based on the optimal findings of [104], [107], depicted in Table 7.2.

Table 7.2: Optimal Configuration for ROAR

Work	**Encr. Algorithm**	**Encr. Rate**	**Burst Duration**	**Burst Pause**
[104]	AES-CTR	500	10 seconds	5 seconds
[107]	ChaCha20	16	1 file	60 seconds

Both works considered an IoT crowdsensing scenario involving a resource-constrained device. However, *HyperDtct*'s scenario assumes a more powerful machine with more data to encrypt. Therefore, these configurations were modified to enable the encryption of more files by setting the encryption rate to 0, thus not limiting the number of bytes encrypted per burst. Additionally, in the best configuration found by [107], the burst pause is set to five seconds to encrypt more files.

7.2.3 Model Evaluation

With the data collected in V2 and V2-1, this work has amassed over 68'000 entries of five-second system call traces issued by several processes. In this subsection, the data collected in V2 and V2-1 is referred to as data collected in V2, as no distinction is made on the detection level. An overview of the data and how it is split into training and evaluation sets is provided by Figure 7.2. To estimate how well the different models perform when faced with samples not seen before, the models trained on the data collected in V1 were used to predict the newly collected data. For each algorithm considered in V1 (NB, RF, IForest, and LOF), the model yielding the best result, outlined in Table 6.2, was used to predict the maliciousness of samples collected in V2. Their performance was evaluated on the evaluation dataset of V2, such that it can be compared to models trained on data collected in V2. The results of these predictions are outlined in Table 7.3.

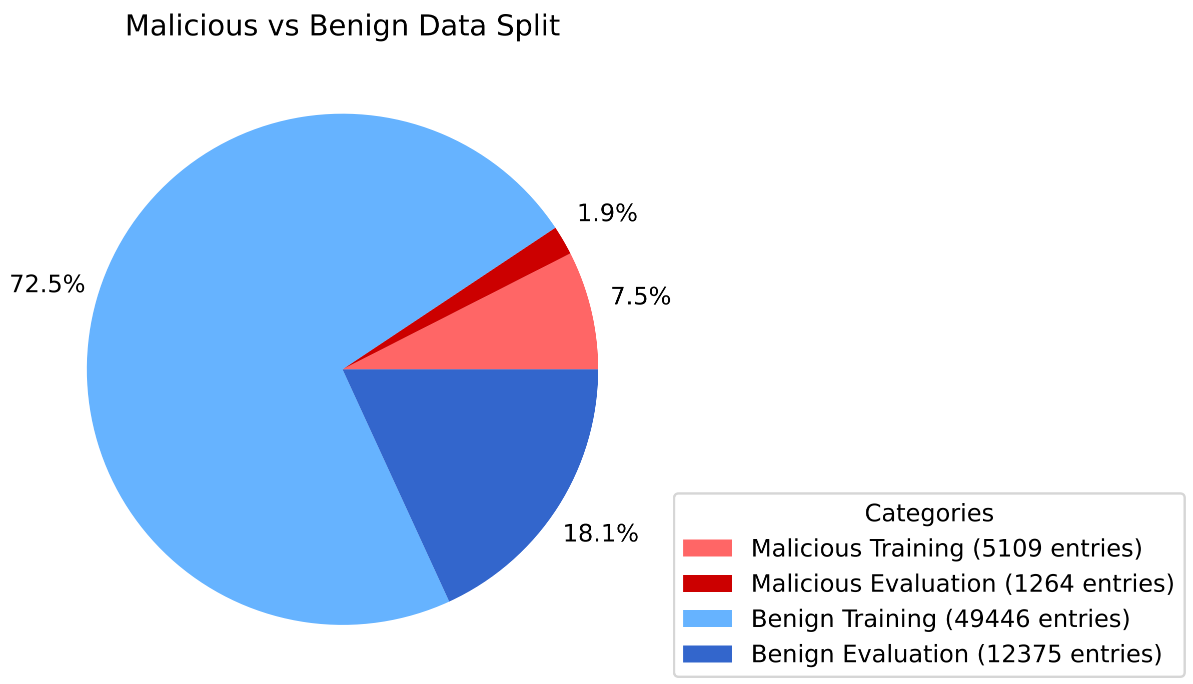


Figure 7.2: Overview over Data Collected in V2 and V2-1

Table 7.3: F_1 Scores of Best V1 Models per Algorithm Across V2 Evaluation Dataset

Algorithm	**Scaler**	**N-Gram**	F_1 **Score**
NB	Frequency	5	0.15
RF	TF-IDF	5	0.0
IForest	TF-IDF	2	0.08
LOF	Frequency	3	0.61

LOF reached a high F_1 score, considering the small number of dataset entries it was trained on. The other models, however, did not reach an F_1 score exceeding 0.15. To train models with the new training dataset, *HyperDtct* was modified to handle larger datasets. In V1, the vectorized datasets were handled as dense matrices. This worked fine, considering the entire dataset only consisted of about 5'500 entries. However, handling more entries becomes memory inefficient, as most values in the matrix are zero. Therefore, the features are converted to a scikit-learn-compatible sparse matrix format when they are formatted to vectors and, different from V1, all algorithms are now provided by scikit-learn, as the considered pyOD algorithms did not support sparse matrices. This change was facilitated in the models in the directory `controller/detection/models/V2`, allowing reproduction of the V1 models stored in the directory `controller/detection/models/V1` as they were documented in Chapter 6. Additionally, the *contamination* parameter of the AD models was updated to represent the contamination factor of the newly collected data, which is 9.3 %. After these changes were implemented and the models trained on all collected data, the F_1 scores outlined in Figure 7.3 were observed across vectorizers and n-grams.

Similar to the previous version, the performance of the classifier models remains better than that of the AD models. Overall, performance scores have decreased. The best

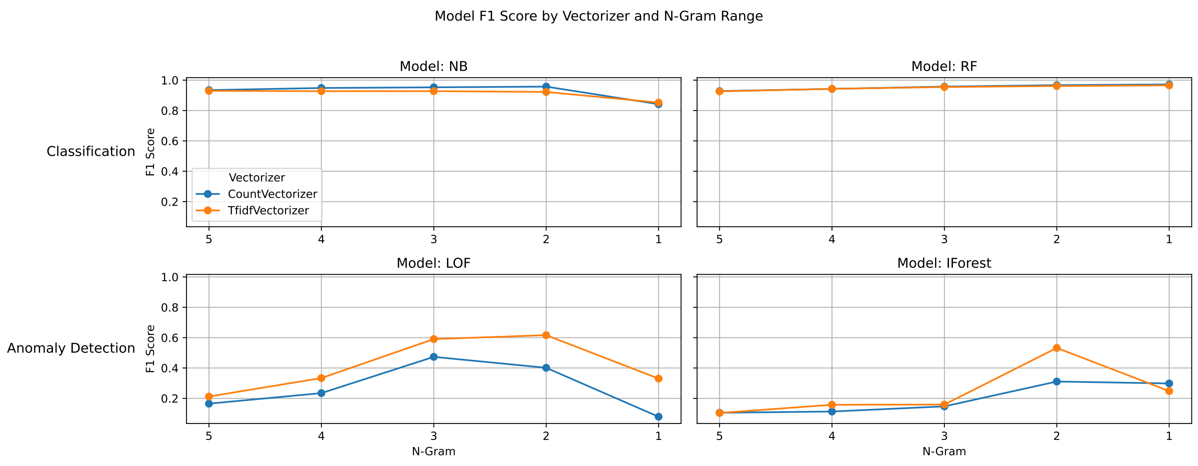


Figure 7.3: F_1 Score of Models Trained on V2 Data

models, outlined in Table 7.4, do not reach F_1 scores as high as the ones in V1, summarized in Table 6.2.

Table 7.4: Best Results per Algorithm

Algorithm	**Vectorizer**	**N-Gram**	F_1 **Score**	**Precision**	**Recall**	**Traing Time**
NB	Frequency	2	0.96	0.96	0.95	1.9 s
RF	Frequency	1	0.97	0.96	0.99	13.7 s
IForest	TF-IDF	2	0.53	0.67	0.44	3.5 s
LOF	TF-IDF	2	0.62	0.82	0.50	2.5 s

Therefore, it was attempted to improve the performance by considering larger n-grams. [12], [13], [84], [86] have found 6-gram and 7-gram to have the best performance; therefore these values for n are considered in the experiments with larger n-grams. Because larger values for n lead to larger dimensions of the feature space, two feature selection techniques, provided by scikit-learn, were considered: Variance Threshold (VT) removes low-variance features, while Recursive Feature Elimination (RFE) uses an external estimator to assign weights to the features. To assess how these feature selection methods would affect the models already implemented at this stage, 4- and 5-grams were also considered. Therefore, these combinations of feature selection techniques, vectorizers, and n-grams are evaluated across all models:

$$\{\text{VT}, \text{RFE}\} \times \{\text{TF-IDF}, \text{Frequency}\} \times \{4, 5, 6, 7\}$$

The *threshold* parameter in the VT selection method was set to 0.0001. Because the considered vectorized data is sparse, this threshold leads to a hefty reduction of features, reducing the number of features considered across the selected feature spaces from a range of 50'000 - 99'000 features to a range of 400-5'500 features. The selected value worked well; however, no further experiments with different values were conducted.

For the RFE method, the number of features to select was set to half of the complete feature space. Logistic regression was used as the estimator, and features were removed in step sizes of 1000 features. The feature selection methods did not heavily influence the F_1 score across both selection methods, except for the IForest models. Feature spaces vectorized using a frequency-based approach had a low deviation from the F_1 score of models trained and evaluated on the whole feature space for both the VT and RFE feature selection methods. On the other hand, feature spaces vectorized using TF-IDF experienced a higher fluctuation, with a decreased F_1 score of up to 0,04 lower. Feature selection methods applied on datasets for the IForest algorithm led to high instability in F_1 performance, sometimes increasing performance up to 0.08 and other times decreasing up to 0.13. These feature selection techniques can thus be applied to some, but not all, algorithms and n-gram ranges to reduce model scoring time, should these models be used in a real-time detection scenario. Because neither higher n-grams nor models with reduced feature spaces outperform the models in Table 7.4, applying feature selection techniques is left to future work, and neither higher n-grams nor feature selection is implemented in *HyperDtct*. The experiments and their corresponding results regarding feature selection and higher n-grams can be found in `controller/detection/fs_experiments.ipynb`.

To improve the classifier models, it was attempted to balance the training dataset by oversampling malicious vectors. To achieve this, two oversampling methods were considered [108]: Random Oversampling (ROS) and Synthetic Minority Oversampling Technique (SMOTE). While ROS naively duplicates and adds more samples of the underrepresented class, SMOTE creates synthetic minority class examples. These oversampling techniques were applied to the V2 training dataset and evaluated using the RF and NB classifier algorithms. With both oversampling techniques, the contamination factor of the dataset was increased to 0.5. After fitting the classifiers on the oversampled datasets, it was observed that the F_1 score improved for models trained on 4- and 5-gram datasets, while those trained on 1- and 2-gram datasets experienced a decrease, regardless of the vectorizer or oversampling technique used. The observed difference in F_1 score ranged from *+0.018* to *-0.070*. Because no model was found to beat the previous best F_1 score outlined in Table 7.4, it was decided not to implement oversampling into *HyperDtct*'s training process. The experiments and results regarding oversampling can be found in `controller/detection/os_experiments.ipynb`.

Similar to V1, hyperparameter tuning was conducted in an attempt to improve the performance of the models. Because the classifier models did not reach a perfect score in this version, their corresponding hyperparameters were also tuned. For the AD models, the considered parameter grid remained the same as outlined in Listings 6.3 and 6.4, the exception being the removal of the algorithm parameter from LOF, as it overrides this setting to 'brute' when fitted to sparse data [109]. For the NB algorithm, only the Hyperparameter *alpha* was considered, which is an additive smoothing parameter. Experiments considered the values *0.2*, *1.0* (default) and *2.0* [110]. For the RF algorithm, more hyperparameters were considered, as outlined in Listing 7.3.

n_estimators denotes the number of trees in the forest, *criterion* is the function to measure the quality of a split and thus determines how a tree grows, and *max_features* determines how the number of features is calculated when looking for the best split [111].

Listing 7.3: Parameter Grid Considered for the RF Algorithm

```
# Define the parameters to be tuned for the RF algorithm
param_grid = {
    'ngram': range(1,6), # [1,2,3,4,5]
    'vectorizer': [CountVectorizer, TfidfVectorizer],
    'n_estimators': [50, 100, 200],
    'criterion': ['gini', 'entropy', 'log_loss'],
    'max_features': ['log2', 'sqrt', None],
}
```

The best configurations and their corresponding F_1 score achieved in these experiments are outlined in Table 7.5. For all algorithms, except for the NB, better scores were achieved than without hyperparameter configuration (*cf.*, Table 7.4). Therefore, the improved hyperparameters were configured in the algorithms contained in the directory `controller/detection/models/V2`.

Table 7.5: Best Hyperparameters and Vectorizers per Algorithm on V2 Data

Algorithm	***Vectorizer***	***N-Gram***	***Hyperparameters***	F_1 ***Score***
LOF	TfidfVectorizer	2	'n_neighbors': 10, 'metric': 'manhattan'	0.627
IForest	CountVectorizer	1	'n_estimators': 100, 'max_samples': 1.0, 'max_features': 0.75	0.538
NB	CountVectorizer	2	'alpha': 1.0	0.958
RF	CountVectorizer	1	'n_estimators': 100, 'criterion': 'log_loss', 'max_features': 'log2'	0.974

7.3 V3: Real-World Ransomware Evaluation

Until this point, the evaluated ransomware consists only of samples created for academic purposes. To assess whether *HyperDtct* is also able to handle samples from the real world, the ransomware Babuk, responsible for the attack on the D.C. Metropolitan Police Department, and Lockbit black, a modern descendent of the ransomware responsible for the attack on Colonial Pipeline Co., are considered in this version of the work. Additionally, to verify that the models have not been overfitted to classify all system-call-intensive tasks as malicious, the file restoration process implemented in V2 (*i.e.*, a benign workload) is monitored as a sample. After a short introduction to the ransomware, the steps to monitor the samples are described, and the performance of the models is evaluated.

7.3.1 Babuk

A sample of the ransomware was fetched from [112]. When Babuk is executed, it first terminates a hard-coded list of processes, including various backup solutions. Then, it attempts to delete Windows Shadow Copies before encrypting files. Babuk spawns multiple threads and potentially does not encrypt all files to speed up this process. For large files, it avoids encrypting the entire file and instead only encrypts parts, rendering it unusable [113].

7.3.2 LockBit

With this sample [114], a very recent ransomware sample of LockBit 3.0, also called LockBit Black, is considered. It is called LockBit Black because it is a successor of BlackMatter, which came from the Darkside ransomware family [115] responsible for the Colonial Pipeline Co. attack. LockBit is widely recognized as the world's most prolific and harmful ransomware, responsible for causing billions of euros in damage. LockBit is offered as a RaaS solution, and version 3.0 encrypts a victim's files and threatens to publish them. More recent samples remain available despite a series of arrests and infrastructure takeovers disrupting the group's operations in early 2024 [116]. To avoid detection and analysis, LockBit makes use of code obfuscation and anti-debugging methods [115] and attempts to escalate its privileges [114].

7.3.3 Collecting V3 Samples

When attempting to execute Babuk, it was found that it did not encrypt files without administrative privileges. Therefore, Babuk was executed as the administrative user. Additionally, it was found that the client machine crashed when more than two drives were attached while Babuk was running. Therefore, the drive containing the file backups was removed. After these changes, Babuk ran successfully. The LockBit sample on the other hand did not require to be run as administrator, however, the monitoring process had to be restarted several times, as a memory corruption crashed the client, before the sample managed to run successfully. Although *HyperDtct* can successfully run and monitor the prolific samples Babuk and LockBit, the assumptions made in V2 did not hold. The monitoring process requires manual intervention because the system recovery method implemented in V2 is insufficient to restore the system. During V3, the system recovery was conducted using Clonezilla [117], and the samples were run manually without relying on automatic restoration. For completion, `/controller/input/V3/settings.json` was added, documenting the settings with which the samples were run.

7.3.4 Model Evaluation

To evaluate whether models fitted on data collected in V2 are able to detect LockBit and Babuk, the best-performing V2 models are used to predict the newly collected samples.

The results of the models evaluated on the Babuk dataset are outlined in Table 7.6, while the performance on LockBit is displayed in Table 7.7. Both tables display F_1 score, precision, and recall, as well as detection time, corresponding to the first five-second timeframe of the ransomware, which has been flagged as malicious.

Table 7.6: Results per Algorithm on the Babuk Dataset

Algorithm	**Vectorizer**	**N-Gram**	F_1 **Score**	**Precision**	**Recall**	**Detection Time**
NB	Frequency	2	0.84	0.74	0.97	5 - 10 s
RF	Frequency	1	0.93	0.91	0.95	5 - 10 s
IForest	Frequency	1	0.30	0.18	0.96	5 - 10 s
LOF	TF-IDF	2	0.23	0.13	0.97	5 - 10 s

Table 7.7: Results per Algorithm on the LockBit Dataset

Algorithm	**Vectorizer**	**N-Gram**	F_1 **Score**	**Precision**	**Recall**	**Detection Time**
NB	Frequency	2	0.96	0.92	1.0	0 - 5 s
RF	Frequency	1	0.96	0.99	0.93	5 - 10 s
IForest	Frequency	1	0.71	0.55	1.0	0 - 5 s
LOF	TF-IDF	2	0.47	0.31	1.0	0 - 5 s

Although the models have been fitted on neither Babuk nor LockBit, they achieve high performance in detecting these samples. Notably, the RF algorithm achieves high results in both instances. All models detect these particular ransomware samples within five to ten seconds and achieve a high recall metric. However, when evaluated on the benign file restoration sample, all models falsely flagged most timeframes, in which 7-zip issued system calls, as malicious. These results are outlined in Table 7.8, which uses accuracy as the performance metric, as the dataset contains no malicious entries. This indicates that the models considered at this stage are overfitted, flagging all system-call-intensive processes as malicious. When evaluating these three datasets combined, the models outlined in Table 7.9 achieve the best performance per algorithm. Notably, all these models achieve a high recall metric while suffering performance loss in precision by flagging benign samples as malicious.

Table 7.8: Results per Algorithm on the File Recovery Dataset

Algorithm	**Vectorizer**	**N-Gram**	**Accuracy**	**Flagged 7-zip Timeframes**
NB	Frequency	2	0.92	164 / 168
RF	Frequency	1	0.93	158 / 168
IForest	Frequency	1	0.84	141 / 168
LOF	TF-IDF	2	0.75	165 / 168

7.3.5 V3 Models

Because of the low precision achieved by the V2 models in Table 7.9, newer models should consider more system-call-intensive benign behavior during training. Therefore,

Table 7.9: Best Results of V2 Models per Algorithm on the Combined Dataset

Algorithm	**Vectorizer**	**N-Gram**	F_1 **Score**	**Precision**	**Recall**
NB	Frequency	2	0.72	0.57	0.99
RF	Frequency	2	0.76	0.63	0.97
IForest	Frequency	1	0.40	0.25	0.99
LOF	TF-IDF	3	0.30	0.19	0.84

the following samples are collected:

- Creating a `.7z` file backup using 7-zip, as outlined in Listing 7.1.
- Creating a `.zip` file backup using 7-zip.
- Restoring the created `.zip` backup using Powershell.

While these samples are collected as logs to the directory `controller/logs/V3`, the logs of Babuk, LockBit, and file recovery are moved to `controller/logs/V3-Eval`, which is excluded from training. Training the models on the additionally collected system-call-intensive benign samples and re-evaluating the performance on the combination of the Babuk, Lockbit, and file recovery datasets, as in Table 7.9, an increase in performance can be observed in Table 7.10. With no loss in recall, the precision was increased for most models, indicating that the additional training data led to the models flagging less benign timeframes as malicious. The best-performing vectorizer and n-gram remained largely the same as in Table 7.9, except for the RF model, where TF-IDF was found to perform better and LOF, with a decrease in n-gram from three to two.

Table 7.10: Best Results of V3 Models per Algorithm on the Combined Dataset

Algorithm	**Vectorizer**	**N-Gram**	F_1 **Score**	**Precision**	**Recall**
NB	Frequency	2	0.74	0.59	0.99
RF	TF-IDF	2	0.80	0.67	0.97
IForest	Frequency	1	0.42	0.27	0.99
LOF	TF-IDF	2	0.31	0.19	0.99

V3-1

Because collecting and fitting more system-call-intensive benign behavior has led to increased performance, in iteration V3-1, oversampling the three collected samples has been considered. Therefore, the directory `controller/logs/V3` was duplicated to `controller/logs/V3-1`, naively oversampling the system-call-intensive benign behavior. Of course, the duplicated timestamp leads to timeframes of V3 containing twice as many system calls instead of achieving the goal of oversampling the three collected samples. Although a mistake was made, models trained on these logs featured increased precision, as outlined in Table 7.11. However, no improvement was noticed when implementing this naive oversampling method correctly by modifying the timestamps to be in the future; therefore, the oversampled logs were removed. To repeat the experiment, the code used for correctly copying and modifying the logs can be found in `controller/detection/v3_model_performance.ipynb`. Still, the improved results outlined in Table 7.11 achieved by mistake indicate that more benign system-call intensive samples would benefit the models, and further experiments are conducted.

Table 7.11: Best Results of V3-1 Models per Algorithm on the Combined Dataset

Algorithm	**Vectorizer**	**N-Gram**	F_1 **Score**	**Precision**	**Recall**
NB	Frequency	2	0.75	0.60	0.99
RF	TF-IDF	2	0.82	0.71	0.97
IForest	Frequency	1	0.46	0.30	0.99
LOF	TF-IDF	2	0.32	0.19	0.99

V3-2

In another attempt to improve the models with oversampling, this version attempts to enhance by training them on selectively oversampled data. Processes that issue more than 40 system calls in most timeframes in the additional samples collected in V3 are oversampled. This approach is further described by Listing 7.4. After training the models on the oversampled logs, the models' performance did not increase more than one percentage point, achieving similar results as the failed oversampling approach of V3-1. Therefore, the data created in this version is also not considered for training in future versions. To allow reproduction of this experiment, the full version of the algorithm outlined in Listing 7.4 can be found in `controller/detection/v3_model_performance.ipynb`.

Listing 7.4: Oversampling System-Call-Intensive Processes

```python
# Load V3 samples, group by PID and timestamp
log_dir = os.path.join(os.getcwd(), '../logs/V3')
df = read_logs_from_dir(log_dir)
prep_df = Preprocessor.get(version=2).group_by_pid_and_timestamp(df)

# Get the five PIDs issuing the most timeframes with more than 20 system calls
filtered_df = prep_df[prep_df['syscall'].apply(lambda x: len(x) > 40)]
syscall_intense_pids = list(filtered_df.value_counts('pid').head(5).index)

# Oversample the data of the processes in syscall_intense_pids
df = df[df['pid'].isin(syscall_intense_pids)]
df['timestamp'] = pd.to_datetime(df['timestamp'])

# For each time the df is stacked, add another day to the timestamp to avoid wrong
    grouping
stacked_dfs = []
for i in range(1, 4):
    new_df = df.copy()
    new_date = datetime.now().date() + timedelta(days=i)
    new_df['timestamp'] = new_df['timestamp'].apply(
    lambda x: x.replace(year=new_date.year, month=new_date.month, day=new_date.day))
    stacked_dfs.append(new_df)

stacked_df = pd.concat(stacked_dfs, ignore_index=True).reset_index(drop=True)
```

V3-3

Finally, after the previous partly successful improvements in the classification of unseen data, this unseen data is included in the training set. Therefore, the models are also trained on Babuk, LockBit, and file restoration samples by copying the logs in `controller/logs/V3-Eval` to `controller/logs/V3-3`, which is included in the training. This leads to the results in Table 7.12, depicting the performance of the best models per algorithm when trained and evaluated on all samples collected in this work.

Table 7.12: Best Results per Algorithm

Algorithm	**Vectorizer**	**N-Gram**	F_1 **Score**	**Precision**	**Recall**
NB	Frequency	4	0.94	0.93	0.95
RF	Frequency	1	0.97	0.95	0.99
IForest	Frequency	1	0.49	0.65	0.40
LOF	Frequency	4	0.48	0.57	0.41

Chapter 8

Summary and Future Work

In summary, this thesis explored three research directions, identified based on related work. First, it proposed a novel way to extract system calls for a ransomware detection system. Secondly, it investigated the implementation or modification of a custom hypervisor for a ransomware detection system. Thirdly, it explored those aspects in the context of AD models to detect ransomware.

These research directions were explored by building a prototype, *HyperDtct*, of a ransomware detection system based on the hypervisor-based debugger HyperDbg, which uses a custom hypervisor to debug kernel- and user applications. Using HyperDbg, a novel way for extracting system call logs for ransomware detection was evaluated. These logs were then vectorized using the TF-IDF and Frequency-based approach across n-grams from one to five before fitting and evaluating models using ML classifier algorithms RF and NB and the AD algorithms IForest and LOF.

The results for these algorithms, vectorizers, and n-grams were evaluated across eleven ransomware and thirteen benign samples in three iterative development cycles, referred to as V1, V2, and V3. The key findings of these evaluations were the following. *i)* *HyperDtct*'s best model (RF) achieved an F_1 score of 0.97, showing that HyperDbg is a viable alternative to related work to extract system calls for ransomware detection. *ii)* Classifier models outperform the AD models in this work. While the best classifier model (RF) reached an F_1 score of 0.97, the best AD model (IForest) only reached an F_1 score of 0.49 in the final evaluation. The RF algorithm achieved the best performance, similar to related work. *iii)* The RF model also performed well when used on completely unseen samples, with an F_1 score of 0.80 achieved when evaluated on the unseen behaviors obtained from Babuk, LockBit, and file restoration. *iv)* RF detected the unseen Babuk and LockBit samples in five to ten seconds. Other models were even faster but achieved a lower precision than RF. *v)* While related literature found ranges of n-gram from six to seven most effective (*cf.*, Chapter 5), *HyperDtct* performed best on ranges from one to four, with the best model (RF) performing best on a frequency-based one-gram dataset. *vi)* Using hyperparameter tuning improved model performances in V1 and V2. The optimal parameters found for the RF algorithm in V2 are *'n_estimators': 100*, *'criterion': 'log_loss'* and *'max_features': 'log2'*. *vii)* Trying larger values for n-grams in combination with feature elimination did not achieve better results in V2. *viii)* While oversampling malicious entries in V2 and benign entries in V3 did not improve model performance, accidentally duplicating the number of system calls issued in benign timeframes improved performance in V3.

Based on these results, this thesis contributes to the defined research directions. The first and second directions were addressed by using HyperDbg. The evaluations demonstrated that HyperDbg is an effective tool for capturing system calls using a custom hypervisor, as evidenced by the high performance of the RF model using the extracted logs. The third direction, using AD models to detect ransomware, was explored but found to be less effective than classifier models. However, this work provides various experiments and evaluations that can serve as a foundation for future work on improving AD models in the context of system-call-based ransomware detection using HyperDbg.

8.1 Future Work

HyperDtct offers several opportunities to be improved by future work. In V3, it became clear that restoring the encrypted files was insufficient to bring the system back to a state where monitoring could be resumed. The samples Babuk and LockBit had to be run manually, outlining the need to implement a restoration method that allows the automatic collection of logs for such samples. Another opportunity is to implement a detection system that detects running ransomware in real-time, as this work shows that detection is possible on previously collected logs and provides useful experiments, such as the experiments conducted on feature selection. Furthermore, the maliciousness of processes is currently determined by their name, assuming that only a single process is responsible for malicious activity. This assumption held for the samples considered in this work, even for harmful samples like LockBit and Babuk, but could be exploited by evasive ransomware, such as the example implemented by [34].

Bibliography

[1] P. O'Kane, S. Sezer, and D. Carlin, "Evolution of ransomware", *IET Networks*, vol. 7, no. 5, pp. 321–327, Sep. 2018. DOI: 10.1049/iet-net.2017.0207.

[2] M. Wade, "Digital hostages: Leveraging ransomware attacks in cyberspace", *Business Horizons*, vol. 64, no. 6, pp. 787–797, 2021. DOI: 10.1016/j.bushor.2021.07.014.

[3] P. H. Meland, Y. F. F. Bayoumy, and G. Sindre, "The Ransomware-as-a-Service economy within the darknet", en, *Computers & Security*, vol. 92, p. 101 762, May 2020, ISSN: 01674048. DOI: 10.1016/j.cose.2020.101762.

[4] E. Caroscio, J. Paul, J. Murray, and S. Bhunia, "Analyzing the Ransomware Attack on D.C. Metropolitan Police Department by Babuk", in *2022 IEEE International Systems Conference (SysCon)*, 2022, pp. 1–8. DOI: 10.1109/SysCon53536.2022.9773935.

[5] J. Beerman, D. Berent, Z. Falter, and S. Bhunia, "A Review of Colonial Pipeline Ransomware Attack", in *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW)*, 2023, pp. 8–15. DOI: 10.1109/CCGridW59191.2023.00017.

[6] C. TEAM, *Ransomware Hit $1 Billion in 2023*, Jul. 2024. [Online]. Available: https://www.chainalysis.com/blog/ransomware-2024/ (visited on 03/27/2024).

[7] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach, "Dynamic Malware Analysis in the Modern Era—A State of the Art Survey", *ACM Computing Surveys*, vol. 52, no. 5, pp. 1–48, Sep. 2020, ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3329786.

[8] O. Aslan and R. Samet, "A Comprehensive Review on Malware Detection Approaches", *IEEE Access*, vol. 8, pp. 6249–6271, 2020. DOI: 10.1109/ACCESS.2019.2963724.

[9] J. Huang, J. Xu, X. Xing, P. Liu, and M. K. Qureshi, "FlashGuard: Leveraging Intrinsic Flash Properties to Defend Against Encryption Ransomware", in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Dallas Texas USA: ACM, Oct. 2017, pp. 2231–2244. DOI: 10.1145/3133956.3134035.

[10] F. Tang, B. Ma, J. Li, F. Zhang, J. Su, and J. Ma, "RansomSpector: An introspection-based approach to detect crypto ransomware", en, *Computers & Security*, vol. 97, p. 101 997, Oct. 2020. DOI: 10.1016/j.cose.2020.101997.

[11] A. Gaur, A. Singh, A. Nautiyal, G. Kothari, P. Mishra, and A. Jha, "DeepHyperv: A deep neural network based virtual memory analysis for malware detection at hypervisor-layer", in *2023 International Conference on Advances in Intelligent Computing and Applications (AICAPS)*, 2023, pp. 1–6. DOI: `10.1109/AICAPS57044.2023.10074347`.

[12] P. Mishra, P. Aggarwal, A. Vidyarthi, *et al.*, "VMShield: Memory Introspection-Based Malware Detection to Secure Cloud-Based Services Against Stealthy Attacks", *IEEE Transactions on Industrial Informatics*, vol. 17, no. 10, pp. 6754–6764, 2021. DOI: `10.1109/TII.2020.3048791`.

[13] P. Mishra, I. Verma, and S. Gupta, "KVMInspector: KVM Based introspection approach to detect malware in cloud environment", *Journal of Information Security and Applications*, vol. 51, p. 102 460, 2020, ISSN: 2214-2126. DOI: `10.1016/j.jisa.2020.102460`.

[14] *LibVMI.* [Online]. Available: `https://libvmi.com/` (visited on 02/19/2024).

[15] M. S. Karvandi, M. Gholamrezaei, S. Khalaj Monfared, *et al.*, "HyperDbg: Reinventing Hardware-Assisted Debugging", in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22, event-place: Los Angeles, CA, USA, New York, NY, USA: Association for Computing Machinery, 2022, pp. 1709–1723, ISBN: 978-1-4503-9450-5. DOI: `10.1145/3548606.3560649`.

[16] 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.

[17] R. P. Goldberg, "Architectural principles for virtual computer systems", PhD Thesis, 1973.

[18] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures", vol. 17, no. 7, 1974.

[19] A. S. Tanenbaum, *Modern operating systems*, 4th ed. Upper Saddle River: Pearson Prentice Hall, 2015, ISBN: 0-13-359162-X.

[20] *Xen Project Software Overview.* [Online]. Available: `https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview` (visited on 02/27/2024).

[21] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "Kvm: The Linux virtual machine monitor", in *Proceedings of the Linux symposium*, Issue: 8, vol. 1, Dttawa, Dntorio, Canada, 2007, pp. 225–230.

[22] C. Chaubal, "The Architecture of VMware ESXi", Tech. Rep.

[23] *Hyper-v Architecture*, Apr. 2022. [Online]. Available: `https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/hyper-v-architecture` (visited on 02/28/2024).

[24] G. Wang, Z. J. Estrada, C. Pham, Z. Kalbarczyk, and R. K. Iyer, "Hypervisor Introspection: A Technique for Evading Passive Virtual Machine Monitoring", in *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, Washington, D.C.: USENIX Association, Aug. 2015. [Online]. Available: `https://www.usenix.org/conference/woot15/workshop-program/presentation/wang`.

[25] T. Garfinkel, M. Rosenblum, *et al.*, "A virtual machine introspection based architecture for intrusion detection.", in *Ndss*, Issue: 2003, vol. 3, San Diega, CA, 2003, pp. 191–206.

[26] B. D. Payne, M. D. P. D. A. Carbone, and W. Lee, "Secure and Flexible Monitoring of Virtual Machines", in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, 2007, pp. 385–397. DOI: 10.1109/ACSAC.2007.10.

[27] Y. Hebbal, S. Laniepce, and J.-M. Menaud, "Virtual Machine Introspection: Techniques and Applications", in *2015 10th International Conference on Availability, Reliability and Security*, 2015, pp. 676–685. DOI: 10.1109/ARES.2015.43.

[28] H. Xiong, Z. Liu, W. Xu, and S. Jiao, "Libvmi: A Library for Bridging the Semantic Gap between Guest OS and VMM", in *2012 IEEE 12th International Conference on Computer and Information Technology*, 2012, pp. 549–556. DOI: 10.1109/CIT.2012.119.

[29] M. C. Libicki, L. Ablon, and T. Webb, *The Defender's Dilemma.* Rand Corporation, 2015.

[30] M. Gorobets, O. Bazhaniuk, A. Matrosov, A. Furtak, and Y. Bulygin, "Attacking hypervisors via firmware and hardware", *Black Hat USA*, 2015.

[31] S. Razaulla, C. Fachkha, C. Markarian, *et al.*, "The Age of Ransomware: A Survey on the Evolution, Taxonomy, and Research Directions", *IEEE Access*, vol. 11, pp. 40 698–40 723, 2023, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3268535.

[32] C. Beaman, A. Barkworth, T. D. Akande, S. Hakak, and M. K. Khan, "Ransomware: Recent advances, analysis, challenges and future research directions", *Computers & Security*, vol. 111, p. 102 490, Dec. 2021, ISSN: 01674048. DOI: 10.1016/j.cose.2021.102490.

[33] F. Khan, C. Ncube, L. K. Ramasamy, S. Kadry, and Y. Nam, "A Digital DNA Sequencing Engine for Ransomware Detection Using Machine Learning", *IEEE Access*, vol. 8, pp. 119 710–119 719, 2020. DOI: 10.1109/ACCESS.2020.3003785.

[34] W. Ma, P. Duan, S. Liu, G. Gu, and J.-C. Liu, "Shadow attacks: Automatically evading system-call-behavior based malware detection", en, *Journal in Computer Virology*, vol. 8, no. 1-2, pp. 1–13, May 2012. DOI: 10.1007/s11416-011-0157-5.

[35] B. Kolosnjaji, A. Demontis, B. Biggio, *et al.*, "Adversarial Malware Binaries: Evading Deep Learning for Malware Detection in Executables", in *2018 26th European Signal Processing Conference (EUSIPCO)*, Rome: IEEE, Sep. 2018, pp. 533–537. DOI: 10.23919/EUSIPCO.2018.8553214.

[36] J. von der Assen, A. H. Celdrán, J. Luechinger, *et al.*, "RansomAI: AI-Powered Ransomware for Stealthy Encryption", in *GLOBECOM 2023 - 2023 IEEE Global Communications Conference*, 2023, pp. 2578–2583. DOI: 10.1109/GLOBECOM54140.2023.10437393.

[37] S. Garfinkel, "Anti-forensics: Techniques, detection and countermeasures", in *2nd International Conference on i-Warfare and Security*, vol. 20087, 2007, pp. 77–84.

[38] R. R. Branco, G. N. Barbosa, and P. D. Neto, "Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies", *Black Hat*, vol. 1, no. 2012, pp. 1–27, 2012.

[39] J. Singh and J. Singh, “A survey on machine learning-based malware detection in executable files”, *Journal of Systems Architecture*, vol. 112, p. 101 861, Jan. 2021. DOI: 10.1016/j.sysarc.2020.101861.

[40] Y. Nagano and R. Uda, “Static analysis with paragraph vector for malware detection”, en, in *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication*, Beppu Japan: ACM, Jan. 2017, pp. 1–7, ISBN: 978-1-4503-4888-1. DOI: 10.1145/3022227.3022306.

[41] A. Balakrishnan and C. Schulze, “Code obfuscation literature survey”, *CS701 Construction of compilers*, vol. 19, p. 31, 2005.

[42] T. Muralidharan, A. Cohen, N. Gerson, and N. Nissim, “File Packing from the Malware Perspective: Techniques, Analysis Approaches, and Directions for Enhancements”, *ACM Comput. Surv.*, vol. 55, no. 5, Dec. 2022, Place: New York, NY, USA Publisher: Association for Computing Machinery. DOI: 10.1145/3530810.

[43] R. B. Hadiprakoso, H. Kabetta, and I. K. S. Buana, “Hybrid-Based Malware Analysis for Effective and Efficiency Android Malware Detection”, in *2020 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS)*, Jakarta, Indonesia: IEEE, Nov. 2020, pp. 8–12. DOI: 10.1109/ICIMCIS51567.2020.9354315.

[44] H. Razeghi Borojerdi and M. Abadi, “MalHunter: Automatic generation of multiple behavioral signatures for polymorphic malware detection”, in *ICCKE 2013*, 2013, pp. 430–436. DOI: 10.1109/ICCKE.2013.6682867.

[45] Ö. Aslan and R. Samet, “Investigation of possibilities to detect malware using existing tools”, in *2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA)*, IEEE, 2017, pp. 1277–1284.

[46] C. Chio and D. Freeman, *Machine Learning and Security.*

[47] R. Vinayakumar, M. Alazab, K. Soman, P. Poornachandran, and S. Venkatraman, “Robust intelligent malware detection using deep learning”, *IEEE access*, vol. 7, pp. 46 717–46 738, 2019, Publisher: IEEE.

[48] D. Tian, Q. Ying, X. Jia, R. Ma, C. Hu, and W. Liu, “MDCHD: A novel malware detection method in cloud using hardware trace and deep learning”, *Computer Networks*, vol. 198, p. 108 394, 2021, ISSN: 1389-1286. DOI: 10.1016/j.comnet.2021.108394.

[49] M. Sewak, S. K. Sahay, and H. Rathore, “Comparison of Deep Learning and the Classical Machine Learning Algorithm for the Malware Detection”, in *2018 19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, 2018, pp. 293–296. DOI: 10.1109/SNPD.2018.8441123.

[50] T. Shinagawa, H. Eiraku, K. Tanimoto, *et al.*, “BitVisor: A thin hypervisor for enforcing i/o device security”, in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE ’09, event-place: Washington, DC, USA, New York, NY, USA: Association for Computing Machinery, 2009, pp. 121–130, ISBN: 978-1-60558-375-4. DOI: 10.1145/1508293.1508311.

[51] M. Hirano and R. Kobayashi, "Machine Learning Based Ransomware Detection Using Storage Access Patterns Obtained From Live-forensic Hypervisor", in *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, 2019, pp. 1–6. DOI: 10.1109/IOTSMS48152.2019.8939214.

[52] M. Hirano and R. Kobayashi, "Machine Learning-based Ransomware Detection Using Low-level Memory Access Patterns Obtained From Live-forensic Hypervisor", in *2022 IEEE International Conference on Cyber Security and Resilience (CSR)*, 2022, pp. 323–330. DOI: 10.1109/CSR54599.2022.9850340.

[53] K. Gogineni, P. Derasari, and G. Venkataramani, "Foreseer: Efficiently Forecasting Malware Event Series with Long Short-Term Memory", in *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, 2022, pp. 97–108. DOI: 10.1109/SEED55351.2022.00016.

[54] J. Zhang, C. Gao, L. Gong, *et al.*, "Malware Detection Based on Multi-level and Dynamic Multi-feature Using Ensemble Learning at Hypervisor", *Mobile Networks and Applications*, vol. 26, no. 4, pp. 1668–1685, Aug. 2021, ISSN: 1383-469X, 1572-8153. DOI: 10.1007/s11036-019-01503-4.

[55] S. Hong, A. Nicolae, A. Srivastava, and T. Dumitraş, "Peek-a-boo: Inferring program behaviors in a virtualized infrastructure without introspection", *Computers & Security*, vol. 79, pp. 190–207, 2018, ISSN: 0167-4048. DOI: 10.1016/j.cose.2018.08.010.

[56] B. Singh, *Memory Forensics for Virtualized Hosts*, Mar. 2021. [Online]. Available: https://blogs.vmware.com/security/2021/03/memory-forensics-for-virtualized-hosts.html (visited on 02/16/2024).

[57] Y. Zhang, S. Luo, H. Wu, and L. Pan, "Antibypassing Four-Stage Dynamic Behavior Modeling for Time-Efficient Evasive Malware Detection", *IEEE Transactions on Industrial Informatics*, vol. 20, no. 3, pp. 4627–4639, 2024. DOI: 10.1109/TII.2023.3327522.

[58] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection",

[59] M. Hirano, T. Tsuzuki, S. Ikeda, N. Taka, K. Fujiwara, and R. Kobayashi, "WaybackVisor: Hypervisor-Based Scalable Live Forensic Architecture for Timeline Analysis", in *Security, Privacy, and Anonymity in Computation, Communication, and Storage*, G. Wang, M. Atiquzzaman, Z. Yan, and K.-K. R. Choo, Eds., Cham: Springer International Publishing, 2017, pp. 219–230, ISBN: 978-3-319-72395-2. DOI: 10.1007/978-3-319-72395-2_21.

[60] M. Hirano, R. Hodota, and R. Kobayashi, "RanSAP: An open dataset of ransomware storage access patterns for training machine learning models", *Forensic Science International: Digital Investigation*, vol. 40, p. 301 314, 2022, ISSN: 2666-2817. DOI: 10.1016/j.fsidi.2021.301314.

[61] M. Cen, F. Jiang, X. Qin, Q. Jiang, and R. Doss, "Ransomware early detection: A survey", en, *Computer Networks*, vol. 239, p. 110 138, Feb. 2024, ISSN: 13891286. DOI: 10.1016/j.comnet.2023.110138.

[62] *DRAKVUF® Black-box Binary Analysis System.* [Online]. Available: https://drakvuf.com/ (visited on 02/16/2024).

[63] *Volatilityfoundation/volatility: An advanced memory forensics framework.* [Online]. Available: https://github.com/volatilityfoundation/volatility (visited on 02/19/2024).

[64] *VirusShare.com.* [Online]. Available: https://virusshare.com/ (visited on 02/22/2024).

[65] *VirusTotal - Home.* [Online]. Available: https://www.virustotal.com/gui/home/upload (visited on 02/22/2024).

[66] v. Team, *VMware leads the HCI Market in 2022, According to IDC*, Apr. 2023. [Online]. Available: https://blogs.vmware.com/virtualblocks/2023/04/18/vmware-leads-the-hci-market-in-2022-according-to-idc/ (visited on 03/22/2024).

[67] *YARA documentation! — yara 4.4.0 documentation.* [Online]. Available: https://yara.readthedocs.io/en/stable/ (visited on 02/23/2024).

[68] K. Lu, *Analysis of .NET Thanos Ransomware Supporting Safeboot with Networking Mode | FortiGuard Labs*, Section: FortiGuard Labs Threat Research, Jul. 2020. [Online]. Available: https://www.fortinet.com/blog/threat-research/analysis-of-net-thanos-ransomware-supporting-safeboot-with-networking-mode (visited on 02/23/2024).

[69] S. Tanda, *Tandasat/HyperPlatform*, original-date: 2016-02-26T14:52:35Z, Feb. 2024. [Online]. Available: https://github.com/tandasat/HyperPlatform (visited on 02/27/2024).

[70] A. Ionescu, *Ionescu007/SimpleVisor*, original-date: 2016-03-16T16:32:00Z, Feb. 2024. [Online]. Available: https://github.com/ionescu007/SimpleVisor (visited on 02/27/2024).

[71] *HyperDbg.* [Online]. Available: https://docs.hyperdbg.org/ (visited on 03/01/2024).

[72] *Vx Underground.* [Online]. Available: https://vx-underground.org/ (visited on 03/01/2024).

[73] *Guest Introspection.* [Online]. Available: https://docs.vmware.com/en/VMware-NSX-Data-Center-for-vSphere/6.4/com.vmware.nsx.admin.doc/GUID-049EF8ED-224C-4CAF-B6E7-1CD063CCD462.html (visited on 02/15/2024).

[74] *vSphere Web Services API - VMware API Explorer.* [Online]. Available: https://developer.vmware.com/apis/1720/vsphere/ (visited on 02/15/2024).

[75] *Virtual Machine Introspection - Xen.* [Online]. Available: https://wiki.xenproject.org/wiki/Virtual_Machine_Introspection (visited on 02/16/2024).

[76] *Xen Project 4.15 Feature List - Xen.* [Online]. Available: https://wiki.xenproject.org/wiki/Xen_Project_4.15_Feature_List (visited on 02/16/2024).

[77] markruss, *LiveKd - Sysinternals*, Mar. 2021. [Online]. Available: https://learn.microsoft.com/en-us/sysinternals/downloads/livekd (visited on 02/27/2024).

[78] *First Steps in Hyper-V Research | MSRC Blog | Microsoft Security Response Center.* [Online]. Available: https://msrc.microsoft.com/blog/2018/12/first-steps-in-hyper-v-research/ (visited on 02/26/2024).

[79] ifeomaufondu-ms, *Enable Intel Performance Monitoring Hardware in a Hyper-V Virtual Machine*, May 2021. [Online]. Available: https://learn.microsoft.com/en-us/windows-server/virtualization/hyper-v/manage/performance-monitoring-hardware (visited on 02/27/2024).

[80] *Connecting To HyperDbg.* [Online]. Available: https://docs.hyperdbg.org/using-hyperdbg/kernel-mode-debugging/examples/beginning/connecting-to-hyperdbg (visited on 02/29/2024).

[81] M. Jurczyk, *J00ru/windows-syscalls*, original-date: 2018-05-31T07:41:52Z, Apr. 2024. [Online]. Available: https://github.com/j00ru/windows-syscalls (visited on 04/04/2024).

[82] *Operation Modes | HyperDbg Documentation*, Feb. 2022. [Online]. Available: https://docs.hyperdbg.org/using-hyperdbg/prerequisites/operation-modes (visited on 04/26/2024).

[83] *Design of !syscall & !sysret | HyperDbg Documentation*, Oct. 2021. [Online]. Available: https://docs.hyperdbg.org/design/features/vmm-module/design-of-syscall-and-sysret (visited on 04/17/2024).

[84] P. Mishra, V. Varadharajan, E. Pilli, and U. Tupakula, "VMGuard: A VMI-based Security Architecture for Intrusion Detection in Cloud Environment", *IEEE Transactions on Cloud Computing*, pp. 1–1, 2018. DOI: 10.1109/TCC.2018.2829202.

[85] A. Zheng and A. Casari, *Feature engineering for machine learning: principles and techniques for data scientists.* O'Reilly Media, Inc., 2018.

[86] K. Tan and R. Maxion, ""Why 6?" Defining the operational limits of stide, an anomaly-based intrusion detector", in *Proceedings 2002 IEEE Symposium on Security and Privacy*, 2002, pp. 188–201. DOI: 10.1109/SECPRI.2002.1004371.

[87] J. Lüthi, *Cyber-Tracer/HyperDtct*, original-date: 2024-04-08T13:09:01Z, May 2024. [Online]. Available: https://github.com/Cyber-Tracer/HyperDtct (visited on 05/28/2024).

[88] S. Garfinkel, P. Farrell, V. Roussev, and G. Dinolt, "Bringing science to digital forensics with standardized forensic corpora", *Digital Investigation*, vol. 6, S2–S11, 2009, ISSN: 1742-2876. DOI: 10.1016/j.diin.2009.06.016.

[89] *Build & Install | HyperDbg Documentation.* [Online]. Available: https://docs.hyperdbg.org/getting-started/build-and-install (visited on 05/07/2024).

[90] *Scripts/windows/DFIR/process-behavior-logger.ds at master · HyperDbg/scripts.* [Online]. Available: https://github.com/HyperDbg/scripts/blob/master/windows/DFIR/process-behavior-logger.ds (visited on 04/17/2024).

[91] *About PortableApps.com | PortableApps.com.* [Online]. Available: https://portableapps.com/about (visited on 04/19/2024).

[92] Jimmy, *Jimmy-ly00/Ransomware-PoC*, original-date: 2020-09-07T12:41:37Z, Apr. 2024. [Online]. Available: https://github.com/jimmy-ly00/Ransomware-PoC (visited on 04/19/2024).

[93] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, "Scikit-learn: Machine Learning in Python", *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[94] T. p. d. team, *Pandas-dev/pandas: Pandas*, Feb. 2020. DOI: 10.5281/zenodo.3509134. [Online]. Available: https://doi.org/10.5281/zenodo.3509134.

[95] Y. Zhao, Z. Nasrullah, and Z. Li, "PyOD: A Python Toolbox for Scalable Outlier Detection", *Journal of Machine Learning Research*, vol. 20, no. 96, pp. 1–7, 2019. [Online]. Available: http://jmlr.org/papers/v20/19-011.html.

[96] *Pyod.models.iforest - pyod 1.1.4 documentation.* [Online]. Available: https://pyod.readthedocs.io/en/latest/_modules/pyod/models/iforest.html (visited on 05/31/2024).

[97] *Pyod.models.lof - pyod 1.1.4 documentation.* [Online]. Available: https://pyod.readthedocs.io/en/latest/_modules/pyod/models/lof.html (visited on 05/31/2024).

[98] *7-Zip.* [Online]. Available: https://www.7-zip.org/ (visited on 05/27/2024).

[99] M. Lavrijsen, *Mattiwatti/EfiGuard*, original-date: 2019-03-25T19:47:39Z, May 2024. [Online]. Available: https://github.com/Mattiwatti/EfiGuard (visited on 05/27/2024).

[100] wille, *Wille/cry*, original-date: 2017-01-11T13:46:17Z, May 2024. [Online]. Available: https://github.com/wille/cry (visited on 05/17/2024).

[101] C. Ayala, *CesarAyalaDev/RAASNet*, original-date: 2021-03-08T22:14:57Z, May 2024. [Online]. Available: https://github.com/CesarAyalaDev/RAASNet (visited on 05/17/2024).

[102] C. J. May, *Lawndoc/RanSim*, original-date: 2021-09-17T15:53:40Z, May 2024. [Online]. Available: https://github.com/lawndoc/RanSim (visited on 05/14/2024).

[103] P. Drakatos, *PanagiotisDrakatos/JavaRansomware*, original-date: 2016-12-16T23:27:34Z, Mar. 2024. [Online]. Available: https://github.com/PanagiotisDrakatos/JavaRansomware (visited on 05/14/2024).

[104] J. Lüchinger, "AI-powered Ransomware to Optimize its Impact on IoT Spectrum Sensors", M.S. thesis, University of Zurich, Mar. 2023.

[105] J. Lüthi, *Dev-Lj/RAASNet*, original-date: 2024-05-13T14:54:01Z, May 2024. [Online]. Available: https://github.com/Dev-Lj/RAASNet (visited on 05/17/2024).

[106] *Pyinstaller/pyinstaller*, original-date: 2011-11-23T11:05:56Z, May 2024. [Online]. Available: https://github.com/pyinstaller/pyinstaller (visited on 05/17/2024).

[107] S. Padovan, "AI-powered Ransomware to Stay Hidden", M.S. thesis, University of Zurich, Jan. 2024.

[108] G. Lemaître, F. Nogueira, and C. K. Aridas, "Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning", *Journal of Machine Learning Research*, vol. 18, no. 17, pp. 1–5, 2017. [Online]. Available: http://jmlr.org/papers/v18/16-365.html.

[109] *LocalOutlierFactor.* [Online]. Available: https://scikit-learn/stable/modules/generated/sklearn.neighbors.LocalOutlierFactor.html (visited on 06/06/2024).

[110] *MultinomialNB.* [Online]. Available: https://scikit-learn/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html (visited on 06/06/2024).

[111] *RandomForestClassifier*. [Online]. Available: `https://scikit-learn/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html` (visited on 06/10/2024).

[112] petikvx, *MalwareBazaar | Babuk*, Apr. 2023. [Online]. Available: `https://bazaar.abuse.ch/sample/01c647838c374e91e8f9fe967fd25235d72264414bb0d5b82c4fbd4151a9717f` (visited on 06/09/2024).

[113] C. Dong, *Babuk Ransomware*, Jan. 2021. [Online]. Available: `https://cdong1012.github.io//reverse%20engineering/2021/01/03/BabukRansomware/` (visited on 06/09/2024).

[114] 0x0d4y, *MalwareBazaar | LockBit*, May 2024. [Online]. Available: `https://bazaar.abuse.ch/sample/03db266db5b96223ef42206d57e30fa58c59e70b1d14c017422f097af1560ad1` (visited on 06/10/2024).

[115] N. Huỳnh, *Dissecting LockBit v3 ransomware*, Sep. 2023. [Online]. Available: `https://blog.calif.io/p/dissecting-lockbit-v3-ransomware` (visited on 06/10/2024).

[116] Europol, *Law enforcement disrupt world's biggest ransomware operation*, Feb. 2024. [Online]. Available: `https://www.europol.europa.eu/media-press/newsroom/news/law-enforcement-disrupt-worlds-biggest-ransomware-operation` (visited on 06/10/2024).

[117] *Clonezilla*. [Online]. Available: `https://clonezilla.org/` (visited on 06/10/2024).

[118] *Git*. [Online]. Available: `https://git-scm.com/` (visited on 06/21/2024).

[119] *Python*, Jun. 2024. [Online]. Available: `https://www.python.org/` (visited on 06/21/2024).

Abbreviations

AD	Anomaly Detection
API	Application Programming Interface
AUC	Area Under the ROC Curve
CNN	Convolutional Neural Network
DLL	Dynamic Link Library
DNN	Deep Neural Network
Dom0	Control Domain (Xen)
DomU	Unprivileged Domain (Xen)
DSE	Driver Signature Enforcement
FN	False Negative
FP	False Positive
FPR	False Positive Rate
GB	Gigabyte
IForest	Isolation Forest
IPT	Intel Processor Trace
KNN	K-Nearest Neighbors
KVM	Kernel-Based Virtual Machine
LBA	Logical Block Access
LOF	Local Outlier Factor
ML	Machine Learning
NB	Naive Bayes
OS	Operating System
PID	Process Identifier
RaaS	Ransomware-as-a-Service
RF	Random Forest
RFE	Recursive Feature Elimination
RL	Reinforcement Learning
ROAR	Ransomware Optimized with AI for Ressource-constrained devices
ROC	Receiver Operating Characteristic
ROS	Random Oversampling
SMOTE	Synthetic Minority Oversampling Technique
SVM	Support Vector Machine
TF-IDF	Term Frequency-Inverse Document Frequency
TID	Thread ID
TN	True Negative
TP	True Positive

TPR	True Positive Rate
UFW	Uncomplicated Firewall
UNM	University of New Mexico
VM	Virtual Machine
VMI	Virtual Machine Introspection
VMM	Virtual Machine Monitor
VT	Variance Threshold

List of Figures

List of Tables

Appendix A

Installation Guidelines

A.1 Controller

The installation guidelines for the controller machine are written for a Raspberry Pi 3 Model B+ with Ubuntu 22.04.4 LTS installed. Nevertheless, these guidelines should work for an arbitrary Ubuntu system. It is assumed that a user with administrative privileges is already set up, Git [118], and Python [119] installed, and the system is connected to the internet.

An unprivileged user named ”logger” is created, and the repository is cloned. This process is outlined in Listing A.1.

Listing A.1: Create Logger-User and Download Repository

```
# Create the logger user and clone the repository
sudo -s
adduser logger
cd /home/logger/
git clone https://github.com/Cyber-Tracer/HyperDtct.git
```

To configure the controller's network and firewall, execute the commands outlined in A.2.

Listing A.2: Setup the Controller's Network

```
# Configure the network of the controller
cd /home/logger/HyperDtct/controller/setup
chmod u+x ./setup_network.sh
chmod u+x ./setup_ufw.sh
./setup_network.sh
./setup_ufw.sh
```

A.1.1 Start Monitoring

Monitoring the samples considered for V2 to V3 is enabled by bash scripts such as `controller/V2.sh`. To monitor V1, add the executables, downloadable from [91], to their corresponding directory in `controller/input/V1`, and run the commands outlined in Listing A.3. To collect logs for individual inputs, `/controller/start_server.py` also accepts single zip files for the *–input* argument. A zip file must contain a `execute.bat` file with all the instructions on executing the sample and all the executables and samples required by the sample.

Listing A.3: Collect V1 Samples

```
# Create zipped input files for V1
cd /home/logger/HyperDtct/controller
mkdir -p input_zipped/V1
cd input
python3 to_zipped.py --directory V1/ --output_directory ../input_zipped/V1/

# Start logging as the unprivileged logger user
su - logger -c "cd HyperDtct/controller && python3 start_server.py --input
input_zipped/V1/ --log_dir /configured/output/dir"
```

A.2 Client

The client machine must be connected to the internet and adhere to the CPUs supported by HyperDbg. This work used an Intel I7-6700k CPU. Install Git [118], Python [119] and 7-zip [98] (skip 7-zip installation if V1 data is collected, as there it will be installed as part of the monitoring process) for Windows and download HyperDtct according to Listing A.4.

Listing A.4: Clone HyperDtct to C Drive

```
cd C:\
git clone https://github.com/Cyber-Tracer/HyperDtct.git
```

After setting up Windows with an administrative user, HyperDbg must be installed, according to the guide [89]. Move the latest release's directory to `C:\HyperDtct\HyperDbg`. Then, download the latest release of EfiGuard from [99], and follow the instructions provided by [99] to set up a bootable loader thumb drive. Move `EfiDSEFix.exe` to `C:\HyperDtct\client\System\EfiDSEFix.exe`. Configure the BIOS to boot on the thumb drive first.

To configure the administrative user to log in automatically and continue monitoring on startup, execute the scripts as outlined in Listing A.5, where the keywords *USERNAME* and *PASSWORD* are the username and password of the administrative user. The commands should be executed from an elevated shell.

Listing A.5: Configure Autologin and Startup Task

```
C:\HyperDtct\client\System\setup\configure_autologin.bat USERNAME PASSWORD
powershell.exe
Set-ExecutionPolicy Unrestricted
C:\HyperDtct\client\System\setup\setup_logging_task.ps1 -username USERNAME
```

To add the client user, execute the commands outlined in Listing A.6. *PASSWORD* is the password of the newly created client user. To allow executables to run with standard privileges, enter and store the client user's credentials.

Listing A.6: Store Client Credetials

```
net user Client PASSWORD /add
py C:\HyperDtct\client\System\runas.py store_creds
```

To allow monitoring samples over an extended period, set the screen saver and standby timeout to never by running `client/System/setup/setup_power_settings.ps1`. To allow malicious samples to run uninterrupted for monitoring purposes, disable Microsoft Defender by running `client/System/setup/disable_ms_defender.ps1` in an elevated shell.

The client is then populated with documents, fetched from [88], and a backup of these files is stored on the second drive. Additionally, the backup drive and the directory of *HyperDtct* are made inaccessible for the client user. These steps are outlined in Listing A.7.

Listing A.7: Populate Client with Documents and Configure Access Rights

```
rem Populate Client user with documents
py C:\HyperDtct\client\System\setup\download_govdocs.py 0 100 C:\Users\Client\Documents

rem Create file backup
7z.exe a -mx1 D:\FileBackup\Documents.7z C:\Users\Client\Documents

rem Configure access rights
icacls C:\HyperDtct /deny Client:(F) /q
icacls D:\ /deny Client:(F) /q
```

A.3 Detection

Everything related to detection is in the directory `/controller/detection`. First, install the required packages outlined in `/controller/detection/requirements.txt`. To train models based on the collected logs, run `/controller/detection/train_models.py`. The script supports the following attributes:

- *–version*: Version of the algorithms and preprocessors to consider. Supported values are 1 to 3.
- *–log_dir*: Directory where the logs for the different versions are stored
- *–output_dir*: Directory where the trained models should be saved.

Appendix B

Contents of the CD

The following was handed in additionally:

1. The midterm- and final presentation of this thesis.
2. The LaTeX source code and pdf.
3. Links to the GitHub repositories [87], [105].
4. *HyperDtct*'s source code, including serialized models considered during the thesis.