University of Zurich^UZH

MASTER PROJECT – Communication Systems Group, Prof. Dr. Burkhard Stiller

Ransomware Detection with Machine Learning in Storage Systems

Dario Gagulic (18-707-257)
Lynn Zumtaugwald (17-929-340)
Siddhant Sahu (20-744-579)

Supervisors: Dr. Alberto Huertas, Jan von der Assen and Dr. Roman Pletka (IBM Research Lab Zurich)
Date of Submission: February 13, 2023

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

ifi

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

Abstract

Ransomware is widely spread and was the top malware-attack type in 2021 that causes people, businesses, and other institutions to pay millions of ransom every year. Thus, a lot of research has been conducted in the detection of ransomware using different methods on heterogeneous levels of computer systems. One possible dynamic approach that provides many advantages such as the ability of performing data extraction in parallel using numerous computational storage devices is ransomware detection in storage systems using machine learning. Hereby, early work has shown promising results in self-contained environments with well-understood workloads. However, little was known about the generalizability of the proposed machine learning model to unseen ransomware, to different storage setups, to realistic office environments and the performance of other models using additional features. This master project focuses on the safe collection of storage access patterns of six ransomware samples (Sodinokibi, LockFile, Lockbit 2.0, WannaCry, Black-Basta, Conti) and one benign workload using a virtual environment detached from any network access. Following-up, a machine learning ransomware detection pipeline was designed and implemented in python using the before-mentioned access patterns as raw data. An enhanced feature set, proven to increase the model's performance, was proposed and used to train the models. Finally, the performance of three different models (Random Forest, XGBoost, DNN) as well as their generalizability has been evaluated. Random Forest and XGBoost have shown to manifest high generalizability to mixed workloads with an average F1-Score of 91.2 %. Further, all models have shown a high generalizability to different storage system setups (F1-Score: 95.4 %) and to unseen ransomware (F1-Score: 92.8 %). These are important capabilities considering the number of new ransomware samples that are created, the amount of different storage system setups that exist, and the fact, that real-world ransomware attacks are executed in noisy, and not self-contained, environments.

Abstrakt

Ransomware ist weit verbreitet und war im Jahr 2021 die häufigste Malware-Angriffsart, welche Menschen, Unternehmen und andere Institutionen dazu veranlasst, jedes Jahr Millionen an Lösegeld zu bezahlen. Daher wurden zahlreiche Forschungsarbeiten zur Erkennung von Ransomware mit verschiedenen Methoden auf heterogenen Ebenen von Computersystemen durchgeführt. Ein möglicher dynamischer Ansatz, der viele Vorteile bietet, wie zum Beispiel die Möglichkeit der parallelen Datenextraktion unter Verwendung von computational storage, ist die Erkennung von Ransomware in Speichersystemen durch maschinelles Lernen. In diesem Zusammenhang haben frühe Arbeiten vielversprechende Ergebnisse in isolierten Umgebungen mit gut verstandenen Arbeitslasten gezeigt. Es war jedoch wenig über die Generalisierbarkeit des vorgeschlagenen Random-Forest-Modells auf unbekannte Ransomware, auf verschiedene Speichersystemkonfigurationen, auf realistische Umgebungen und auf die Leistung anderer Modelle mit zusätzlichen Features bekannt. Dieses Masterprojekt konzentriert sich auf die sichere Sammlung von Speicherzugriffsmustern von sechs Ransomware-Samples (Sodinokibi, LockFile, Lockbit 2.0, WannaCry, BlackBasta, Conti) und einer gutartigen Arbeitslast unter Verwendung einer virtuellen Umgebung, die von jeglichem Netzwerkzugriff getrennt ist. Anschliessend wurde eine Pipeline für Ransomware Erkennung durch maschinelles Lernen entworfen und in Python implementiert, wobei die zuvor erwähnten Zugriffsmuster als Rohdaten verwendet wurden. Ein erweitertes Feature-Set, das nachweislich die Leistung des Modells erhöht, wurde vorgeschlagen und zum Trainieren der Modelle verwendet. Schliesslich wurde die Leistung von drei verschiedenen Modellen (Random Forest, XGBoost, DNN) sowie ihre Generalisierbarkeit bewertet. Random Forest und XGBoost zeigten eine hohe Generalisierbarkeit für gemischte Arbeitslasten mit einem durchschnittlichen F1-Score von 91.2 %. Darüber hinaus haben alle Modelle eine hohe Generalisierbarkeit für verschiedene Speichersystemkonfigurationen (F1-Score 95.4 %) und für unbekannte Ransomware (F1-Score 92.8 %) gezeigt. Dies sind wichtige Fähigkeiten, wenn man bedenkt, wie viele neue Ransomware-Muster erstellt werden, wie viele verschiedene Speichersystemkonfigurationen es gibt und dass Ransomware-Angriffe in der Realität nicht in isolierten Umgebungen ausgeführt werden.

Acknowledgments

We would like to express our gratitude to Prof. Dr. Burkhard Stiller, Dr. Alberto Huertas, Jan von der Assen, Dr. Slavisa Sarafijanovic, Dr. Roman Pletka, Dr. Dionysios Diamantopoulos and Dr. Haris Pozidis for their support, guidance, and feedback during the research and writing process of this master project. We also wish to thank IBM Research Lab Zurich for providing the infrastructure which was necessary for this project.

Contents

Chapter 1

Introduction

1.1 Motivation

A lot has changed since in the late 1980s fraudsters encrypted victims' documents for the first time and demanded cash to be sent by post in return for the releasing of the documents [1]. Together with the emergence of cryptocurrencies, ransomware attacks are getting more prevalent, especially in recent years, where attackers ruthlessly target companies, individuals, and institutions such as Ireland's Health Service [2] and many more. In 2021, ransomware is considered the top attack with 23% of all malware attacks [3]. The number of attacks has more than doubled from the year 2020 (305 million) to the year 2021 (623 million) with a total ransom of approximately 600 million USD paid [4]. The infection mainly happens through phishing, but also through exposing vulnerabilities [3]. This makes it a lucrative business for hackers and indicates that the number of attacks will not decrease any time soon. In fact, over 100 new ransomware families appear every year, and new variants show even more sophisticated obfuscation mechanisms making detection more difficult.

Therefore, it is of high importance to improve existing and explore new detection mechanisms. Ransomware detection can be done by different approaches and on different levels. Static approaches consist of methods of ransomware detection before the ransomware is executed. Hereby, the syntax or structural features, such as byte representation, opcode, or portable executable (PE) header of malware are used to detect possible attacks. This approach works well for known malware but has the disadvantage that it can not detect zero-day attacks and that it is vulnerable to many obfuscation techniques [5]. In contrast, dynamic approaches consist of methods where information is analyzed during or after the ransomware execution. Since ransomware samples exhibit a set of characteristic features at run-time that is common across families, these approaches allow early detection of new variants. Hereby, anomaly-based dynamic ransomware detection has a set of rules for known benign workloads and assumes that all anomalous activity is malicious. In practice, however, it has shown to be difficult to develop a good set of rules, without having a large positive rate. In behavior-based dynamic ransomware detection, the behavioral patterns of ransomware are analyzed and a workload is evaluated by its intended actions. This approach allows real-time detection and is capable of detecting zero-day attacks. Behavioral patterns of ransomware can be observed on different levels. First, the network traffic generated by ransomware can be used for detection. It provides the advantage that the detection can be executed without a significant impact on the host input/output (IO) traffic. However, a disadvantage is that this method can not detect ransomware that is not generating network traffic, and monitoring the network traffic can produce a high overhead depending on how granular network traffic is monitored [6, 7]. Second, the behavior patterns exhibited by ransomware on the file system level can be used for detection. For example, file opening and closing, file layout information, and metadata of files. However, this approach impacts the host IO traffic [8]. Third, ransomware behavior on the operating systems (OS) level, such as created processes, required permission, run-time dynamic link library (DLL), searched windows and more can be used for detection. This, however, is dependent on the OS itself and the used libraries. Further, monitoring every process on the OS produces an overhead with an impact on the host IO traffic [9]. Fourth, ransomware detection can be executed in storage systems using information from storage activity only. This approach provides the advantages that (1) data extraction can be performed in parallel using numerous computational storage devices. (2) The feature extraction and inference part can be executed directly in the storage system stack using computational storage. And thus, (3) All these tasks can be executed without a significant impact on the host IO traffic. Detecting ransomware using information from storage systems only has the difficulty that only IO operations can be tracked. However, early work performing ransomware detection on block storage systems using machine learning [10, 11] shows promising results in ransomware detection in self-contained, simple environments using a simple feature set.

However, very little is known about the generalizability of the model developed. Especially, the generalizability to new, during training unseen, ransomware is to this point unknown. Considering various new ransomware that is created and the number of ransomware variants, this is an important factor. Further, the generalizability of this model to different storage setups is unknown. Again, an important factor considering the many different setups that are used with storage systems in practice. Moreover, the generalizability of the model to more realistic environments, where ransomware and benign workloads are run at the same time, is uncertain. Additionally, the latest research has focused on Random Forest, Support Vector Machines (SVM) and K-Nearest Neighbors (KNN) models only and the performance of other models is unexplored. Lastly, the feature set used for training the model is narrow and an enhanced feature set provides the opportunity to improve the detection capabilities further.

1.2 Description of Work

Therefore, this masters project focuses on:

1. Storage access pattern collection of ransomware and benign workloads in a safe, self-contained virtual environment:

- Establishment of a process to safely collect ransomware/benign workload storage access patterns by using a virtual environment detached from any network access.
- Design and implementation of a benign workload simulator bash script that performs file conversion in random order includes pauses, and only reads 10% of the files
- Collection of storage access patterns of six ransomware samples (Sodinokibi, LockFile, Lockbit 2.0, WannaCry, BlackBasta, Conti) and one benign workload on two different storage setups. Additionally, collection of storage access patterns of three different ransomware samples and benign workload executed simultaneously on one setup.
- Validation of collected storage access patterns by entropy and time series analysis using extracted entropy values from block storage device pre- and post-infection and the obtained storage access patterns. The entropy is extracted using a program written in C and the analyses are performed using a Python program that provides visualizations and allows for a comparison.

2. Machine learning detection system design and implementation

    - Establishment of an enhanced feature set and analysis of the added information value by comparing the performance results to the results of the previous feature set. Measurement of the impurity-based, permutation-based, and SelectKBest feature importance - together with a feature correlation and a theoretical time complexity analysis of the features led to a final feature selection.
    - Feature extraction design and implementation using a Python script.
    - Training of a Random Forest, an XGBoost, and a seven-layer DNN model on the binary classification task of distinguishing ransomware from benign workload. The before-mentioned labeled storage access patterns serve as training data for the models.

3. Performance evaluation and generalizability analysis of the trained models

    - Evaluation of the models' performance on distinguishing ransomware from benign workload using F1-Score.
    - Evaluation of the models' generalizability to different storage system setups by using access patterns obtained on one setup as training data and access patterns from the other setup as evaluation data.
    - Evaluation of the models' generalizability to more realistic, noisy environments, where one ransomware and one benign workload are executed simultaneously. In this scenario, the training data contains access patterns from separate workloads while the evaluation data contains mixed workloads only.
    - Evaluation of the models' generalizability to (during training) unseen ransomware by excluding one ransomware sample from the training set and evaluating the models' performance on the excluded sample only.

1.3 Thesis Outline

Chapter 2 of this report describes the context of ransomware and its detection. *Chapter 3* explores related work on ransomware detection. *Chapter 4* focuses on the design used in this master project including the ransomware, feature and model selection, and model parametrization. And further presents the implementation of the test environment, benign workload simulator, feature extraction, model training, and evaluation used in this project. *Chapter 5* presents the experiments and their results. Since this report contains numerous experiments, their results are directly discussed to improve the reading flow. The end of the chapter contains a comparative discussion of all experiments and results. *Chapter 6* summarizes the master project, draws conclusions, and provides insights about future work. In the *Appendix A*, the structure of the code repository and the feature extraction code are presented, as well as the external dependencies, and *appendix B* shows the contributions of each student.

Chapter 2

Background

In this chapter, the most important keywords are introduced and explained to ensure a general and common understanding used for further reading.

Malware aims to corrupt or misuse the users' system and stands short for malicious software. There exist different types of malware that all have to bypass the systems' prevention and detection mechanisms and have the goal to remain undiscovered until the desired actions have been successfully completed. Some popular types of malware are *worms* which can self-replicate and infect other connected computers,*trojans* hiding malicious code within legitimate software, *viruses* which spread between computers by actively executing them, and *ransomware* which encrypts users files or locks the users' system and demands a ransom payment to free the system. Most malware makes use of social engineering to invade the users' system and exploit the users' unawareness or good faith [12].

Phishing is one sort of social engineering that exploits people's behavior and their willingness to freely publish sensitive information with very little thought of security and privacy. Users mostly consider their interaction partners as trusted, even though the only identification is an e-mail address. The current trend of ubiquitous communication enhanced the user's unawareness and makes such approaches more effective [13].

Ransomware encrypts or locks users' sensitive information and demands a ransom to be paid for the information to be decrypted and released. Modern ransomware increasingly uses double and triple extortion techniques, as will be shown in subsection 3.1. This type of malware has proved to be a lucrative approach and therefore shows an increasing interest. Furthermore, attackers are working on sophisticated obfuscation techniques by changing the ransomware's appearance without affecting the underlying behavior to evade the detection of anti-virus systems. This makes it more difficult to detect new emerging ransomware families [14], [15].

Storage Systems and Devices are needed for users to store data, regardless of form. Direct area storage (DAS), like floppy disks, hard disk drives (HDD), flash drives, and solid-state drives (SSD) is often directly connected to the computing machine and can provide local backup services. Network-based storage (NAS) is better accessible for data sharing and collaboration since more than one computer can access it through a network [16]. Besides direct vs network-attached storage, there is also the distinction between block, object, and file storage. In block storage, the data is broken up into blocks, which are stored as separate pieces with unique identifiers. Block storage is typically used in Storage Area Networks (SANs) or cloud-based storage environments where fast, efficient, and reliable data transportation is required. In contrast, object storage breaks data files up into pieces called objects, which are stored in a single repository spread across multiple network systems. Similar to block storage, these objects have unique identifiers, which are used by applications to identify the object. The important difference between block and object storage is that metadata is handled differently. In object storage, the metadata includes rich information about the files stored in the object. Whereas in block storage, metadata is limited to basic file attributes. As the name indicates, in file storage all the data is stored in a single file. This type of storage is used to store files on a computer's hard drive and can be used also to store data on a NAS. Other examples include cloud-based file storage systems, network drives, computer hard drives, and flash drives. The files are structured hierarchically with folders and sub-folders which makes retrieving files easier using a file manager. While file storage can be easy to configure, the data access is constrained by a single path to the data, which impacts the performance compared to block and object storage [17].

Machine learning is a common approach used in modern ransomware detection methods. The problem of distinguishing new variants of malware from their predecessors relying only on static analysis like content or signature-based techniques faces limitations and is becoming less effective [9]. Therefore, supervised machine learning is used to learn patterns in labeled data and train models which are capable of distinguishing benign from malicious workloads (binary classification) or even using multi-classification for more than two different classes. Some algorithms used so far to detect ransomware include Random Forest, SVM, and KNN. These algorithms do require some computation to be done but are also able to detect obfuscated malware with higher accuracy [18].

Deep Learning and Deep Neural Networks are a subset of machine learning which show improved performance on more complex classification problems than classical machine learning algorithms. Deep learning can be supervised, semi-supervised and unsupervised. Deep learning architectures include amongst others deep neural networks (DNN), deep reinforcement learning (deep RL), recurrent neural networks (RNN), convolutional neural networks (CNN), and Transformers, each having its application domain. In DNN's, neural networks with many layers having many nodes are stacked and used to progressively extract higher-level features from raw input [19].

Chapter 3

Related Work

This chapter explains related work and research state-of-the-art to ransomware and ransomware detection.

3.1 Ransomware Overview

Although decreasing from 23 % in 2020 to 21 % in 2021, ransomware has still been the top attack type amongst malware in 2021 [3]. The absolute number of attacks has more than doubled from 304.6 million in 2020 to 623.3 million in 2021 [4]. The frequency of ransomware attacks tends to shift through the year, with the highest in May and July and the lowest in January. Phishing has shown to be the most important infection vector in 2021, with more than 41 % of observed ransomware attacks using phishing as its infection vector. Often phishing is conducted through emails, where the victim clicks on a malicious link or appendix and the ransomware gets downloaded to the machine. Targets for phishing are individuals as well as companies and other institutions. Microsoft, Apple, Google, and BMO Harris Bank were the top brands frequently spoofed by phishing emails in 2021. The second highest infection vector, vulnerability exploitation, allows advisories to gain access to victim networks for further operations with often elevated privileges. Because the number of vulnerabilities hit a new record high every year for the past 10 years, it is likely to continue as an attack vector for ransomware in the future. One example of an important vulnerability exposed in 2021 is the vulnerability in Apache's Log4j Library, where advisories can execute arbitrary code (and also ransomware) on the victims' machine with very little expense. This library is widespread and used in a large number of software packages and libraries.

The increasing use of Internet of Things (IoT) devices, opens a wide range of new vulnerabilities to advisories since IoT devices often have worse security mechanisms in place. Once gaining initial access to the victim's machine, ransomware attacks start to understand the victim's local system and domain to acquire additional credentials to enable lateral movement, which describes the process by which attackers spread from an entry point to the rest of the network. Then, the data collection and exfiltration can begin. According to X-Force [3], most ransomware attacks in 2021 used the principle of double extortion tactic of data theft, in which threat actors exfiltrate a victim's sensitive data in addition to encrypting it. In addition to that, some ransomware as the last step also targets a domain controller as a distribution point for the ransomware payload to infect more machines. A new concerning trend in ransomware is the use of triple extortion techniques, where attackers additionally to the double extortion technique also threaten to engage in distributed denial of service (DDoS) attacks against the affected organization. Figure 3.1 shows types of ransomware observed in 2021 by X-Force. The average time before a ransomware family rebrands or shuts down has shown to be 17 months. This means that ransomware is a highly volatile area that is evolving very quickly [3]. As explained, the possible infection vectors for ransomware get larger every year and can not be completely removed, IT Security tends to shift to ransomware detection instead of ransomware protection.

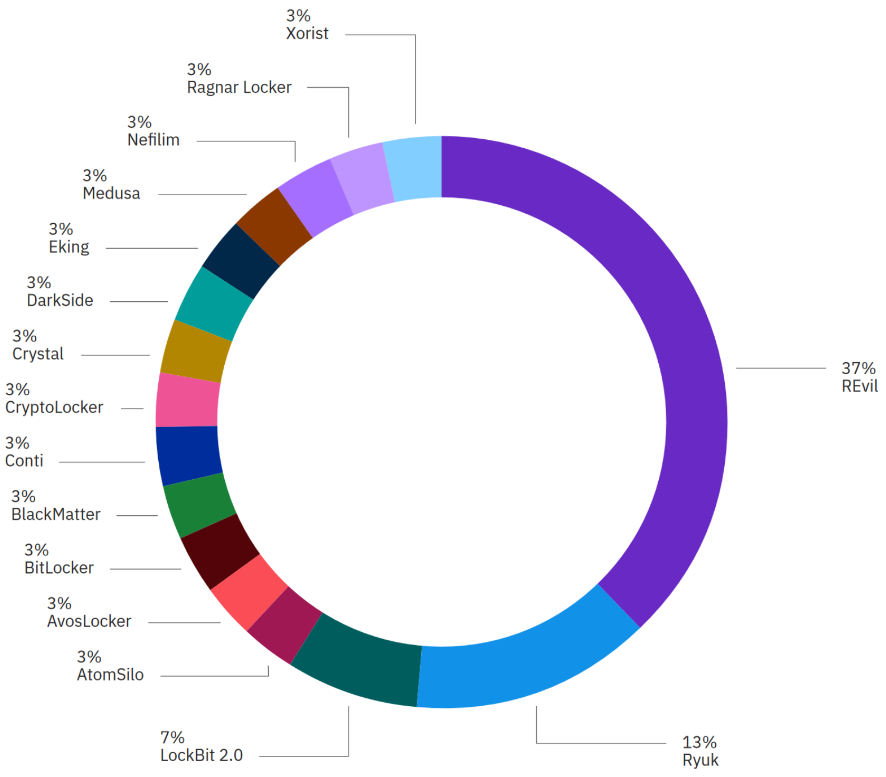


Figure 3.1: Types of ransomware observed in 2021 [3].

3.2 Static and Dynamic Detection Approaches

As ransomware is changing rapidly, also the detection mechanisms are evolving by tackling the problem of ransomware detection from different approaches. Existing ransomware detection techniques can be broadly divided into static and dynamic detection techniques:

3.2.1 Static Detection

The process of examining the malware binary without actually running the executable is called static analysis. In this, the detection of ransomware is done prior to it's execution, i.e. before it is run. Hence, with static analysis it is not possible to detect an ongoing attack. There are several alternatives, signature-based techniques hash the ransomware binary and compare it against a set of already known ransomware binaries [20, 21, 22]. A more sophisticated approach relies on using a disassembler like IDA[23] to perform reverse engineering directly on the malware's binary files. With such tools, it is possible to convert the machine-executable binary into a more human readable assembly language. Having the binary code converted into assembly code give the analyst the ability to inspect algorithm of the ransomware and it's capability.

Maiorca et al. [24] proposed R-PackDroid for Android-based mobile platforms, which looks at the system API packages and labels the Android ransomware executable into three classes as either "Ransomware", "Malware" and "Trusted". The only distinction between the "Ransomware" and the "Malware" classes is that the, "Ransomware" class locks the Graphical User Interface (GUI) of the Android device and the "Malware" class does not. In the very first step they run static analysis on the Android application executable to determine the *packages* from the different Application Programming Interfaces (APIs). Subsequently, the frequency of these API packages are computed, which are used as features for the Random Forest Classifier. Similarly, Abdulrahman et. al. [25] proposed RanDroid, which decompiles Android Application Package (APK) with the APKtool [26], but in contrast to Maiorca et al. [24] which uses the frequency of packages from the APIs as features, it makes use of the strings and images from the GUI as features. All string statements are extracted from the classes.dex files. In addition to this, all the images are also extracted. Following this, DroidBot [27] is used to simulate a user using the Android device. All the images and texts extracted are pre-processed and then standardized into one single unified format. Moreover, the text present in the image can also be extracted with the help of an Optical Character Recognition (OCR) tool Pytesser [28]. Finally, similarity measure, namely the Image Similarity Measurement (ISM) and the String Similarity Measurement (SSM) are calculated respectively and compared against a known set of ransomware samples. Alzahrani et. al. proposed RanDetector [29], which also does static analysis of the Android APK files, but in contrast to the previous two works, it converts the Android Dalvik executable into a Java Archive (JAR) file, which enables them to extract additional features including permissions, intents and API library information, which are used to train various machine learning models, including kNN, Logistic Regression, SVM, XGBoost, and Random Forest. All these models were tested and evaluated on 450 different Android APKs. Khan et al. [30] proposed DNAact-Ran, which argue that newer versions of the same ransomware family come up with more sophisticated obfuscation techniques by hiding the malicious code in more creative ways [31, 32, 33] making it difficult to be detected by signature based techniques. Hence, they suggest to make use of digital DNA sequencing design constraints. The MOGWO [34] and BCS [35] algorithms are used to generate the features and which are then used to produce the digital DNA sequence for each of the features. The digital DNA sequence is in turn used to compute the k-mer frequency vector. These so called "DNA sequences" can be seen as a representation of the ransomware binary artifact. These features are used to train a linear regression model in an active online learning fashion to avoid labeling the dataset.

Although, the detection accuracy with these approaches is sufficient it has a few weak points. Firstly, it is difficult to analyse exactly the same ransomware binary that is being witnessed in the real attack. Additionally, some ransomware are directly loaded into the memory, without creating a file, consequently making it difficult to analyse. Another concern is about the generalizability, in the sense that doing a static analysis and looking at its API calls and byte-code is specific to a particular ransomware and does not transfer well to other ransomware, and consequently making it difficult to run the "same" static analysis for different ransomware. Moreover, new versions of the "same" type of ransomware but using different obfuscation methods fails the static detection. Furthermore, static analysis would also fail when significant code changes are made to previous versions.

3.2.2 Dynamic Detection

It is also possible to do the detection while the ransomware is actually running. These approaches are called dynamic detection approaches. In order to perform the detection, dynamic analysis entails running the malware sample and observing how it behaves on the system. This requires additional software/firmware or hardware components to be added to the system such that the system statistics can be closely monitored and intercepted in order to do the detection of any potential malware running on the system. Table 3.1 at the end of the subchapter provides a summary of the discussed dynamic detection methods.

McIntosh et al. [14] further classified these approaches into subcategories which are elaborated below:

Opcode and API Runtime Analysis - The idea of this detection technique is that since the ransomware manipulates the user files, it would be using the Application Programming Interfaces (APIs) that are provided by the file system. Therefore, when ransomware or a benign workload is running on the system, it would be interfacing with these APIs and then the API calls would further be translated into opcodes. Hence, these approaches aim at analyzing the patterns in these requests to be able to distinguish ransomware from benign workloads. Maniath et al. [36], proposed a Long Short Term Memory (LSTM) network based solution in which the API calls made by either a ransomware process or a benign process are considered as a sequence of words, analogous to natural long sentences. This in turn helps frame the problem as binary classification on natural languages. The way the API calls are intercepted is with the help of Cuckoo sandbox [37] which employs API hooks to the process. 157 ransomware samples are executed and the API calls traced by the Cuckoo sandbox are saved to generate a dataset. Amongst all the API calls, 239 were different only the 38 calls which were common among the runs of the ransomware were kept for training in a data pre-processing step, and subsequently, each sequence was labeled. This processed dataset was then fed into a LSTM of 3 layers, with each layer consisting of 64 LSTM nodes. Aviad et al. [38] used the Volatility framework [39], which is a forensic tool that collects artifacts from volatile memory (RAM). This utility scans memory dump files, such as *.dmp and *.vmem files, and extracts data from the system, including the Windows Registry, lists of active processes, services, drivers, events, connected devices, files in use, network sessions, sockets, and dynamic-link libraries (DLLs) in use. 23 such features were extracted with this framework and a dataset was created, post which Random Forest, Naive Bayes, Bayesian Network, Logistic Regression, LogitBoost, and Sequential Minimal Optimization were employed for training on this dataset. Out of all these classifiers, it was noticed that Random Forest performed the best. The drawbacks of these approaches are that some ransomware can access or modify the sectors and MBR directly circumvent the need to go through the file system. Therefore, for this ransomware, it is not possible to do the detection with this technique.

Daku et al. [9] analyzed behavioral reports of different ransomware families and classified modified variants of ransomware based on their behavior on the OS level. To find the most significant features an iterative approach was conducted, consisting of extensive testing and experimental analysis, combined with a grouping approach where closely depending attributes were combined. All attributes that have shown to contribute to a better accuracy were selected for the final classification, namely the *Runtime DDL, Open Services, Opened Mutexes, Created Mutexes, Opened Service Managers, Searched Windows, Shell Commands, Injected Process, Required permissions, Created Processes, File Access and Family.* Interestingly, their experiments have shown that a reduced feature set leads to improved classification accuracy. They concluded that having closely dependent features can lead to inaccurate results and should be excluded. The best performing model, decision tree classifier J48, reached an accuracy of 78 %.

Encryption Activity Detection - Hill et al. [40] proposed CryptoKnight, which is a Dynamic Convolutional Neural Network (DCNN) deep learning model that aims at extracting and learning patterns from control flow diagnostic data of the dynamic trace. This model is designed such that it is possible to handle variable-length control flow from the traces. The CryptoKnight model can detect malware with encryption algorithms including AES, RC4, Blowfish, MD5 and RSA. Similarly, OS statistics like CPU usage, network traffic, memory usage, disk usage and more can be monitored for the detection. For instance, Ketzaki et al. [41], Sgandurra et al. [42] and Verma et al. [43] showed that during an attack the CPU usage was persistently high. Al-rimy et al. [44, 45] examined applications of cryptographic primitives and consequently their approaches can only detect ransomware by observing the encryption activity, which ignores the problem that encryption can also be hidden, i. e. by avoiding the use of OS cryptographic libraries or working with safe containers (e.g., PowerShell, VM, compression utility software). Another disadvantage of this strategy is the necessity to develop criteria and heuristics to distinguish “normal” from “abnormal” behavior which might be contentious. Additionally, it is also difficult to estimate if some of these indicators from the self proposed heuristics are actually some workloads which are initiated by the user himself.

File System Activity Analysis - Most of the ransomware operate on files by either createing, deleting or modifying the files. It does so in massive numbers i.e., huge number of file accesses, creations, deletions, and modifications. Consequently, this information can be used for detection. Scaife et al. [46] proposed CryptoDrop, which uses three primary indicators, namely file type changes, similarity measurement, and Shannon entropy, and two secondary indicators, deletion and file type funneling in order to characterize the different processes running in the system. The disadvantage of this approach is that it fails in detecting ransomware which can directly access the MBR or perform the encryption through a command container or virtual machine.

Network Activity Analysis - Some researchers have also tried to examine the network traffic from the host system since it is known that the ransomware makes calls to the Command and Control server (C& C) in order to be able to communicate the transfer of keys for encryption. Therefore, the idea of this approach is to detect these network requests which can be an indication of an ransomware attack by observing the network traffic. Similar to other approaches there have been studies with and without the use of machine learning models: Windows-based systems research has shown that supervised learning can be used to detect malicious activities. Alhawi et al. [47] used network traffic information in form of conversations to remote attacker network address to train classifiers like Bayes Network, Multilayer Perceptron, J48, KNN, Random Forest, and Logistic Model Tree and consequently detect existing ransomware attacks. By using features like the protocol type, source and destination IP address, and port they were able to detect current tracked families from the ransomware tracker website *Virus Total* by training a J48 classifier with a TPR of 97.1 % and lowest FPR of 1.6 %. Additionaly, Noorbehbahani et al. [15] compared the effectiveness of the 6 supervised machine learning models Naive Bayes, SVM, KNN, Random Tree, Random Forest, and Decision Tree for differentiating ransomware from benign workloads on Android systems by using a subset of the CICAandMal2017 dataset, containing 80 network-flow features. Their findings show, that Random Forest proved to be the best classifier with an accuracy of 82.8 % to distinguish ransomware from benign operations on Android systems amongst the 6 supervised machine learning models that have been evaluated. However, alternative supervised machine learning models such as XGBoost and also deep learning models have not been taken into consideration in this work, but might improve performance.
This method requires the availability of information on ransomware from a tracker site and could be used as an orthogonal approach to improve the detection capabilities discussed in this master project on the storage system level. However, network activity analysis can not detect ransomware that does not contact C&C servers and thus do not generate network traffic. According to Berrueta et al. [6] 33.9 % of ransomware does not generate network traffic.

Storage Activity Analysis Most existing ransomware detection mechanisms monitor file accesses and other OS activity or network traffic. In storage systems that operate on a block level, a lot of information that is helpful to detect ransomware attacks is not accessible. For instance, file open and close or the file layout information is not accessible. But performing ransomware detection in a storage system has also many advantages: (1) Data extraction can be performed in parallel using a large number of computational storage devices. (2) The feature extraction and inference part can be executed directly in the storage system stack by using a computational storage architecture. And thus,

(3) all these tasks can be executed without a significant impact on the host IO traffic. Nevertheless, interesting insights have also been presented from the research on detecting ransomware on the storage system level. Hirano and Kobayashi [10] were successful in distinguishing ransomware from benign workload by training three different machine learning models only with information gathered from storage systems, such as write and read throughput, the variance of logical block addresses (LBAs) accessed and the Shannon entropy of written sectors. Their model reached an accuracy of 98 %. However, since the data used for training and evaluation is very similar, the validity of the result is unclear because of possible information leakage between the test and the evaluation dataset. As a follow-up, Hirano et al. [11] presented a new open dataset called RanSAP [48] with storage access patterns of ransomware and benign workloads in 2021 and an article published in the Forensic Science International in 2022 which describes the dataset, its collection process, as well as many different evaluations of a Random Forest model trained on this dataset. The model was trained and evaluated with 7 different ransomware and 5 benign software. Their model reached a high F1-Score of 96.2 % in distinguishing two classes - benign workload from ransomware. Further, they showed that this trained model can detect new variants of seen ransomware with a high recall of over 95 %. The evaluation of the model on storage access patterns collected in a more realistic office environment (running different benign software at the same time as ransomware) is limited to 80 % accuracy and the model has not been evaluated in terms of generalizability to new, during training unseen, ransomware.

Sandboxing - This could be one of the most sophisticated way to attack a system with ransomware. Essentially, sandboxing is a technique in which the execution of any workload is done is an artificial environment. This is helps the workload to hide effectively from any detection system which is running on the host system. The following two approaches use sandboxing. Kharraz et al. [49] proposed Unviel, which comprises of three components, reproducing realistic user environments, observe file system activity and observe the graphical user interface. The authors claim that reproducing an artificial but close to real environments is important so that it is durable to ransomware samples which use fingerprinting heuristics from the files stored. The file system monitor is the most crucial part of their approach which operates in the kernel mode. They argues that it is better to the Windows Filesystem Minifilter Driver framework [50] as opposed to API or system call hooking. This module is placed between the I/O scheduler and FileSystem driver, and listen on every user generated I/O, and identifies the process and I/O type, identifies the file output, and looks into the file's data buffer in order to calculate the Shannon entropy [51] of the I/O's. The collected I/O traces are then sorted and checked for patterns like, read, overwrite or delete along with their corresponding entropy. It was noticed that the different ransomware samples within the same ransomware family exhibited the same type access patterns. For instance, for some ransomware family they overwrite the same file and some open, read and encrypt and create and new file and then securely delete the old files. In addition to this, the graphical user interface is constantly scanned to check for some ransom note and the structural image similarity (SSIM) is calculated against a database of screenshots of already known ransomware. Moreover, the texts are also extracted from the graphical user interface and looks for words like lock, pay, ransom, etc.

Similarly, Tang et al. [52] proposed RansomSpector, which is formulated on the virtual machine introspection (VMI) method. This method is set up bellow the OS level, in the hypervisor, thus it is also possible to do the detection ransomware that tries to do the attack by running it as administrator privileges. Since this method can uses the VMI method, it is very well suited to virtualized environments, for instance the cloud. This technique not only closely keep an eye on the file system I/O activities but also the network activities. Similar to Kharraz et al. [49], it also looks for different access patterns, like over-writing file, and reading, encrypting and create a new file and deleting the old one. Although, some ransomware can stores the encryption keys directly in the ransomware code itself, but this approach is rare since antivirus software can easily reverse these encryptoin keys [53]. Therefore, in addition to the file system access pattern, it also monitors the network activity pattern.

Table 3.1: Dynamic Detection Methods Summary

Method	Analyzed Parameters	Research	Advantages	Disadvantages
Operating System Activity Analysis	Opcode, system calls, API calls, Run-time DDL, Required permissions and more	Daku et al. [9]	Possible detection of ransomware at early stage of execution, before it can cause damage; provides specific behavior usable in incident response and forensic analysis	Analysis impacts host IO traffic, resource intensive.
Encryption Activity Detection	Cryptographic primitives, OS statistics like CPU usage, memory and disk usage	Hill et al. [40], Al-rimy et al. [44, 45]	Cryptographic primitives are good features to detection, since most of ransomwares are involved in encrypting user files.	Encryption can also be hidden, i.e., avoiding the use of OS cryptographic libraries, working with safe containers (e.g., PowerShell, VM, compression utility software), difficult to distinguish a benign workload like file compression to encryption.

File System Activity Analysis	File Type Changes, Similarity Measurement, Secondary Indicators(Deletion, File type funneling), Union Indication, Indicator Evasion	Scaife et al. [46]	Easier Implementation, Files are a good indicator to look at for the problem, good accuracy	Difficult to distinguish if the encryption activity was user generated or generated by a ransomware
Network Activity Analysis	protocol type, source & destination IP-Address, port	Alhawi et al. [47], Noorbehbahani et al. [15]	No significant impact on the host IO traffic; detection of communication to C&C-Server to identify the source of an attack and track it's spread; identify and isolate infected devices	Can not detect ransomware that does not generate network traffic.
Sandboxing	Access patterns, like over-writing file, and reading, encrypting and create a new file and deleting the old one	Kharraz et al. [49], Tang et al. [52]	Possible to do the detection ransomware that tries to do the attack by running it as administrator privileges	It is not OS agnostic and depended on the VM's guest OS, dependent on the VMI technology being used
Storage Activity Analysis	IO's	Hirano et al. [10, 11]	parallelizeable data extraction; feature extraction and inference executable on storage system stack; no significant impact on the host IO traffic	limited information, only IO's

3.3 Placement of Ransomware Detection in Computer Systems

The aforementioned static and dynamic detection approaches can be placed at various levels into a computer system. McIntosh et al. [14] classified them into hardware level, as well as kernel and user mode levels.

3.3.1 Hardware Level

The detection of ransomware can be done at a very low level i.e. at the hardware level. This is achieved by leveraging sensors in the hardware or special firmware support running on the hardware that is not directly exposed to the OS. Manaar et. al. introduced RATAFIA [54] which is a unsupervised machine learning model trained on the Hardware Performance Counters (HPC) events such as branch instructions, branch misses, cache references, and cache misses. This model uses a Deep Neural Network and Fast Fourier Transformation (FFT). When this model was introduced it was the first of its kind there has not been previous work on using the HPC events in order to detect ransomware. [55, 10, 11] proposed a similar model but specifically the detection on the storage device level. Their models used access patters in storage devices to be able to distinguish a ransomware workload from a normal/benign workload. They showed that it is indeed possible to use machine learning and other statistical models to extract access patterns in the storage device to determine the type of workload being run on the system. The drawback of this approach is that some ransomware on purpose obscure the encryption activities in order to prevent themselves from being detected by these models. Consequently, it was seen that using just hardware-based detection of ransomware would not be sufficient for the efficient and effective discovery of ransomware [56, 57]. This master project also focuses on tackling the detection problem at this level, more specifically from the storage systems point of view.

3.3.2 Kernel Level

Another alternative approach is to do the detection problem not directly in the hardware level but higher i.e. in the OS kernel level. [58, 59, 60] have addressed the detection problem at the kernel mode level. All these papers collect the features specific to the OS to check for ransomware-like behaviours by looking at the file system. The shortcoming of these approaches were that it was very difficult to distinguish amongst all the activities which were initiated by the user and which were not. Additionally, it was also very difficult intercept the activities that bypassed the file system, for instance for some ransomware which could directly access the physical sectors and could modify the Master Boot Record (MBR). Furthermore, all of these methods aimed at doing the detection in one-shot. Moreover, building ransomware detection solution at the kernel mode level requires comprehensive understanding of the architecture of the operating system and which is operating system specific.

3.4 Findings from Related Work

Ransomware is a widely spread type of malware and a high thread for individuals, companies, and other organizations around the world. Considering the related work, it shows that ransomware detection is widely being researched on different levels. From a storage system perspective, Hirano et al. [11] developed a model capable of distinguishing ransomware from benign software with high accuracy and the model has been shown to be transferable to ransomware variants. However, very little is known about the models' generalizability to new ransomware. This generalization of a model is very important because of the high variability of ransomware and benign software that evolves daily. This makes training a model on every ransomware at any given point in time nearly impossible and demands having a transferable model to new ransomware and also other benign software types essential for effective ransomware detection. Therefore, this master project focuses on improving the detection of ransomware with machine learning models using only storage system access patterns, on evaluating the generalizability of these models to unseen ransomware, to different storage setups and to the models' detection capabilities in more realistic environments, where benign workload and ransomware run at the same time.

Chapter 4

Design and Implementation

This chapter describes the reproduction of the most recent related work in ransomware detection in block storage system and the design and implementation of this master project.

4.1 Reproduction of Related Work

This section starts with an in depth analysis of the latest existing work in ransomware detection in block storage systems and then depicts the necessary steps of reproducing it and discusses the results of the reproduction compared to the original ones.

4.1.1 In Depth Analysis of Latest Existing Work in Ransomware Detection in Block Storage Systems

This subsection provides a detailed description and analysis of the latest existing work in ransomware detection in block storage systems.

Hirano and Kobayashi [10] were successful in classifying ransomware by training three different machine learning models only with information gathered from storage systems. In their research from 2019, their best-performing model reached an accuracy of 98 %. They collected input/output (IO) traces of a storage system where two ransomware (WannaCry and TeslaCrypt) and one benign workload (Zip) were run separately by using Wayback-Visor, a hypervisor that is located between the hardware and the operating system. They started tracing immediately after the workload was started and continued tracing for 30 seconds. From these traces, five features were computed 4.1:

These features are computed over different window sizes T_{window} of one,five,ten,15,20 and 25 seconds with shifting the windows one second until the end of the trace time is reached. Shannon [61] introduced the concept of information entropy and proved that entropy represents an absolute mathematical limit on how data can be losslessly compressed in a noiseless channel. In contrast, when files are not encrypted and partially or not at all

Table 4.1: Features presented by Hirano et al. [10].

Feature	reads/writes
average normalized Shannon entropy (bit)	writes
average throughput [byte/s]	writes
Variance of Logical Block Address (LBA)	writes
average throughput [byte/s]	reads
Variance of Logical Block Address (LBA)	reads

compressed, their entropy is low. The Shannon entropy of a byte is a value between zero and eight and is computed by Equation 4.1. Where x_i is an ith byte in a sector s and $p(x_i)$ represents the probability of a byte x_i. Zero represents a series of the same bytes and hence denotes low randomness and eight represents a perfectly even distribution of byte values and hence high randomness.

$$H(s) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i) \tag{4.1}$$

To render the data unusable but yet recoverable if the encryption key is known, ransomware encrypts files or parts of a file and thus produces data with high entropy. If data would just be replaced with data having similar entropy such that the entropy stays the same, this information could be use to find the original data. This is why encryption is important. The average normalized Shannon entropy of writes $H_{write}(t)$ is computed using the Equation 4.2 where $H_i(s,t)$ is the Shannon entropy $H(s)$ of the i^{th} written sector s in the window t and N is the number of write accesses in the window t of size T_{window}.

$$H_{write}(t) = \frac{1}{N} \sum_{N}^{i=1} H_i(s,t) \tag{4.2}$$

The variance of a logical block address (LBA) is computed using Equation 4.3, where LBA$_i(t)$ is a logical block address of an i^{th} read access or that of an i^{th} write access in window t, $\overline{LBA(t)}$ is the mean over all $LBA_i(t)$ in that window, and N is the number of the read accesses or that of the write accesses in window t of size T_{window}.

$$V(t) = \frac{1}{N-1} \sum_{i=1}^{N} \left(LBA_i(t) - \overline{LBA(t)} \right)^2 \tag{4.3}$$

These windowed features are then used to train and evaluate Random Forest, KNN, SVM models. The best performance was reached by the Random Forest model and $T_{window} =$ 15 seconds as shown in table 4.2 [10]:

These are promising results but should be taken with caution, since having a total trace time T_d of 30 seconds and T_{window} of 15 seconds with a shifting of one second, there are only 16 windows created which might be very similar to each other. Figure 4.1 depicts the overlap created when using the above-mentioned window sizes. In this setting, one might end up having a very similar training and testing set and this could distort the results. Further, it is a very limited setting with only two ransomware and one benign workload.

Table 4.2: Evaluation metric for different window sizes in Random Forest (Macro average) [10].

Window size (seconds)	Accuracy	Precision	Recall	F-measure
1	0.79	0.81	0.77	0.93
5	0.93	0.93	0.92	0.93
10	0.96	0.96	0.95	0.96
15	**0.98**	**0.98**	**0.98**	**0.98**
20	0.98	0.98	0.97	0.98
25	0.96	0.97	0.96	0.96

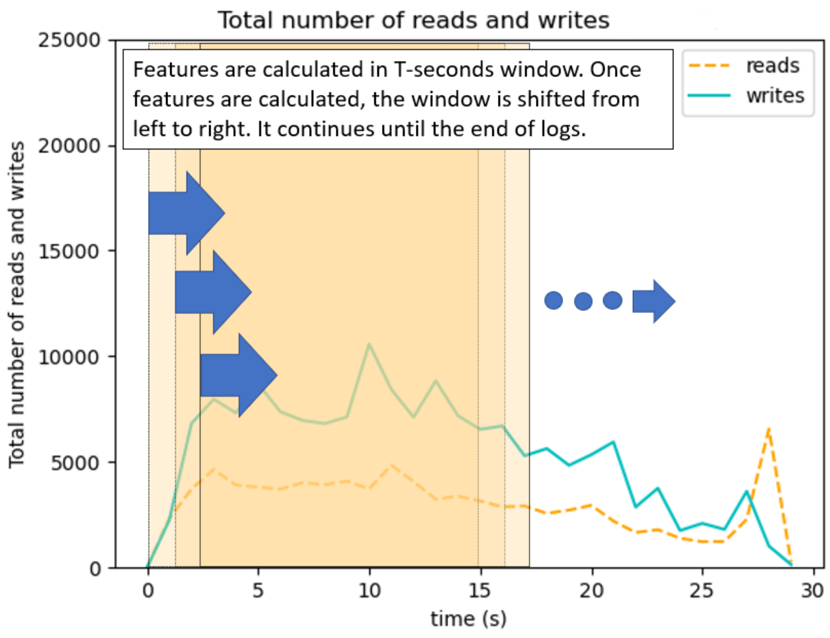


Figure 4.1: Overlapping window sizes used for the feature extraction based on [10].

As a follow-up, Hirano et al. [48] presented a new open dataset called RanSAP with storage access patterns of ransomware and benign workloads in 2021 and an article published in the Forensic Science International in 2022 which describes the dataset, its collection process as well as many different evaluations of this data. The core dataset consists of storage access patterns of the following 7 ransomware *TeslaCrypt, Cerber, WannaCry, Ryuk, GandCrab v4, Sodinokibi, and Darkside* and following five benign workloads *AE-SCrypt v3.10, Zip (Windows 7), SDelete v2.02, Excel 2016, Firefox v90.02* simulating common and harmless operations by the user. For each malicious ransomware sample storage access patterns from a total of three different setups were obtained:

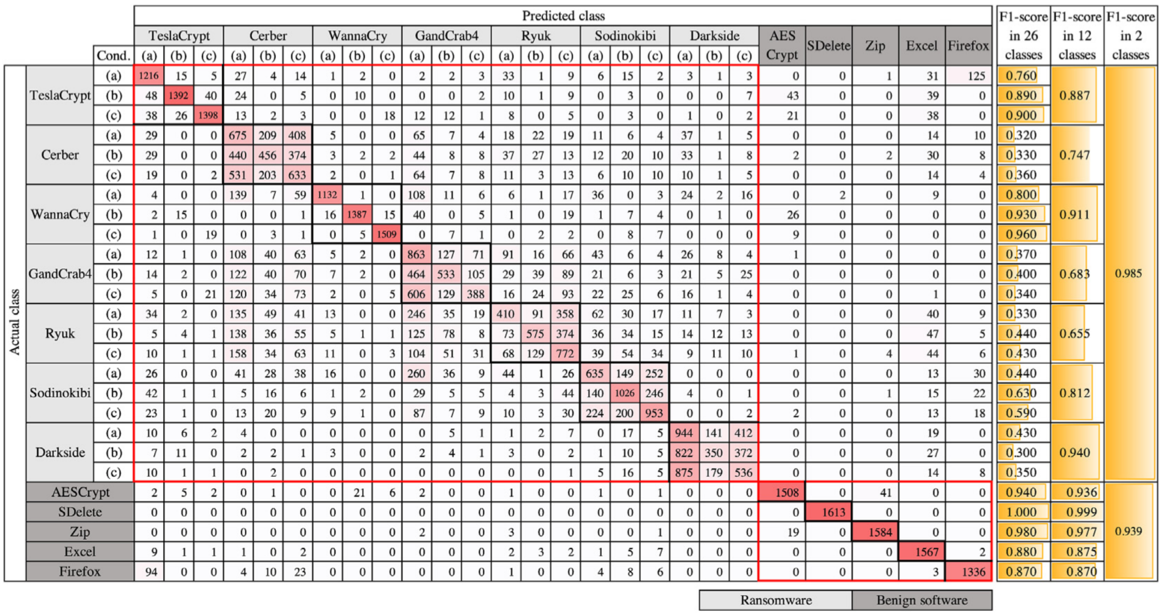
- (a) No decoy files saved on the desktop
- (b) 9872 decoy files in various sizes
- (c) 605 large decoy files

The dataset was analyzed in many different ways. As a first step, t-SNE, a popular variation of the dimensionality reduction algorithm Stochastic Neighbour Embedding (SNE), was applied to the data from the RanSAP dataset with T_{window} of ten seconds and T_d equal to 30 seconds and 60 seconds. Further, a differentiation between 26 classes (all seven ransomware collected under conditions (a) (b) and (c) plus the five benign workloads separated), 12 classes (each ransomware family separate but no differentiation of the collecting conditions (a) (b) and (c) and the benign workloads separated) and in two classes (ransomware and benign). They concluded that in a setting of two classes, there are clusters of data visible on the dimensionality reduction plot and therefore, the developed five-dimensional feature vectors can be used to discriminate ransomware from benign software. As a second step, they compared the performance of Random Forest, SVM, and KNN models applied to different classification settings (24, 12, and two classes) and applied to the traces from different hardware (120GB HDD, 250 GB HDD, 120 GB SSD, 250 GB SSD and all of them combined). They showed, that the performance of Random Forest and KNN was very similar and both better than the performance of SVM. As a third analysis, the F1-Score of a Random Forest model was trained with $T_d = 90$ seconds was plotted against different window sizes from one to 60 seconds. The results show that the higher the window size, the better the results. Hirano et al., however, point out, that using windows with a higher size also means, that the system will take longer to detect an ongoing ransomware attack. The F1 score in binary classification (ransomware/benign) starts converging with a window size of $T_{window} =$ ten seconds or higher and seems to be the best solution regarding the mentioned trade-off. Figure 4.2 shows the confusion matrix of seven ransomware and five benign software on Windows 7 with F1 Scores in 26, 12, and two classes evaluated on a Random Forest model with $T_{window} =$ ten seconds and $T_d = 90$ seconds and using half of the files of the RanSAP dataset for training and half of the files for testing. It shows high F1 Scores of 96.2 % for binary classification, 85.8 % for multi-classification in 12 classes, and significant lower F1 Scores of 58 % in 26 classes. In conclusion, ransomware can effectively be distinguished from benign software using Hirano et al.'s proposed model within the given conditions.

4.1.2 Reproduction of Latest Existing Work in Ransomware Detection in Block Storage Systems

Since this work from Hirano et al. is to our best knowledge the latest in ransomware detection on block storage systems and serves as a starting point for this master project, it is a necessary step to reproduce and validate the results. To achieve that, the RanSAP dataset [48] was used and the same five features mentioned in Table 4.1 were computed over a window size of ten seconds and a window offset of one second using self-developed code written in python and described in Subsection4.3.5. Randomly, half of the dataset was used for training and the other half for testing. The data provided in the RanSAP dataset has been used as is, no preprocessing of the data has been applied (outliers removal, handling of missing data etc.). Using the feature vectors, a Random Forest model was trained and evaluated in a confusion matrix.

The reproduced confusion matrix 4.3 shows a similar F1-Score in 26 classes of 58.2 %, in 12 classes of 83.1 %, and in two classes of 95.2 %. Further, the overall pattern looks very

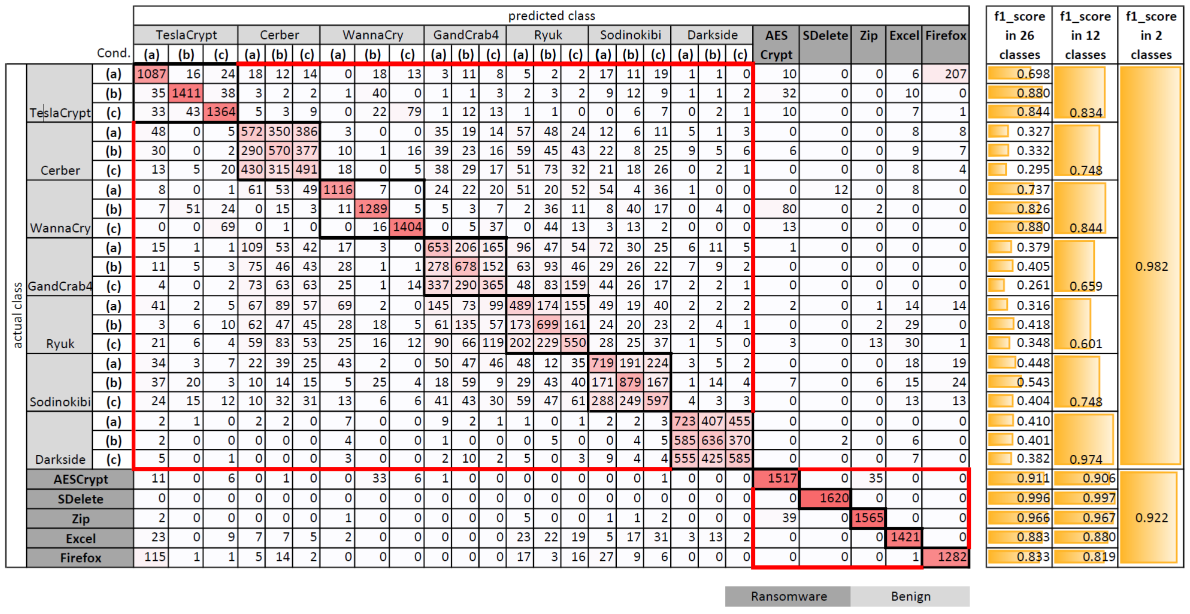


Predicted class

Actual class		TeslaCrypt			Cerber			WannaCry			GandCrab4			Ryuk			Sodinokibi			Darkside			AES Crypt	SDelete	Zip	Excel	Firefox	F1-score in 26 classes	F1-score in 12 classes	F1-score in 2 classes
	Cond.	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)								
TeslaCrypt	(a)	1216	15	5	27	4	14	1	2	0	2	2	3	33	1	9	6	15	2	3	1	3	0	0	1	31	125	0.760		
	(b)	48	1392	40	24	0	5	0	10	0	0	0	2	10	1	9	0	3	0	0	0	7	43	0	0	39	0	0.890	0.887	
	(c)	38	26	1398	13	2	3	0	0	18	12	12	1	8	0	5	0	3	0	1	0	2	21	0	0	38	0	0.900		
Cerber	(a)	29	0	0	675	209	408	5	0	0	65	7	4	18	22	19	11	6	4	37	1	5	0	0	0	14	10	0.320		
	(b)	29	0	0	440	456	374	3	2	2	44	8	8	37	27	13	12	20	10	33	1	8	2	0	2	30	8	0.330	0.747	
	(c)	19	0	2	531	203	633	2	0	1	64	7	8	11	3	13	6	10	10	10	1	5	0	0	0	14	4	0.360		
WannaCry	(a)	4	0	0	139	7	59	1132	1	0	108	11	6	6	1	17	36	0	3	24	2	16	0	2	0	9	0	0.800		
	(b)	2	15	0	0	0	1	16	1387	15	40	0	5	1	0	19	1	7	4	0	1	0	26	0	0	0	0	0.930	0.911	
	(c)	1	0	19	0	3	1	0	5	1509	0	7	1	0	2	2	0	8	7	0	0	0	9	0	0	0	0	0.960		
GandCrab4	(a)	12	1	0	108	40	63	5	2	0	863	127	71	91	16	66	43	6	4	26	8	4	1	0	0	0	0	0.370		
	(b)	14	2	0	122	40	70	7	2	0	464	533	105	29	39	89	21	6	3	21	5	25	0	0	0	0	0	0.400	0.683	0.985
	(c)	5	0	21	120	34	73	2	0	5	606	129	388	16	24	93	22	25	6	16	1	4	0	0	0	1	0	0.340		
Ryuk	(a)	34	2	0	135	49	41	13	0	0	246	35	19	410	91	358	62	30	17	11	7	3	0	0	0	40	9	0.330		
	(b)	5	4	1	138	36	55	5	1	1	125	78	8	73	575	374	36	34	15	14	12	13	0	0	0	47	5	0.440	0.655	
	(c)	10	1	1	158	34	63	11	0	3	104	51	31	68	129	772	39	54	34	9	11	10	1	0	4	44	6	0.430		
Sodinokibi	(a)	26	0	0	41	28	38	16	0	0	260	36	9	44	1	26	635	149	252	0	0	0	0	0	0	13	30	0.440		
	(b)	42	1	1	5	16	6	1	2	0	29	5	5	4	3	44	140	1026	246	4	0	1	0	0	1	15	22	0.630	0.812	
	(c)	23	1	0	13	20	9	9	1	0	87	7	9	10	3	30	224	200	953	0	0	2	2	0	0	13	18	0.590		
Darkside	(a)	10	6	2	4	0	0	0	0	0	0	5	1	1	2	7	0	17	5	944	141	412	0	0	0	19	0	0.430		
	(b)	7	11	0	2	2	1	3	0	0	2	4	1	3	0	2	1	10	5	822	350	372	0	0	0	27	0	0.300	0.940	
	(c)	10	1	1	0	2	0	0	0	0	0	0	0	0	0	1	5	16	5	875	179	536	0	0	0	14	8	0.350		
AESCrypt		2	5	2	0	1	0	0	21	6	2	0	0	1	0	0	1	0	1	0	0	0	1508	0	41	0	0	0.940	0.936	
SDelete		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1613	0	0	0	1.000	0.999	
Zip		0	0	0	0	0	0	0	0	0	2	0	0	3	0	0	0	0	1	0	0	0	19	0	1584	0	0	0.980	0.977	0.939
Excel		9	1	1	1	0	2	0	0	0	0	0	0	2	3	2	1	5	7	0	0	0	0	0	0	1567	2	0.880	0.875	
Firefox		94	0	0	4	10	23	0	0	0	0	0	0	1	0	0	4	8	6	0	0	0	0	0	0	3	1336	0.870	0.870	

Ransomware	Benign software

Figure 4.2: Original confusion matrix of seven ransomware and five benign software on Windows 7 from [11].



predicted class

actual class		TeslaCrypt			Cerber			WannaCry			GandCrab4			Ryuk			Sodinokibi			Darkside			AES Crypt	SDelete	Zip	Excel	Firefox	f1_score in 26 classes	f1_score in 12 classes	f1_score in 2 classes
	Cond.	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)								
	(a)	1087	16	24	18	12	14	0	18	13	3	11	8	5	2	2	17	11	19	1	1	0	10	0	0	6	207	0.698		
	(b)	35	1411	38	3	2	2	1	40	0	1	1	3	2	3	2	9	12	9	1	1	2	32	0	0	10	0	0.880		
TeslaCrypt	(c)	33	43	1364	5	3	9	0	22	79	1	12	13	1	1	0	0	6	7	0	2	1	10	0	0	7	1	0.844	0.834	
	(a)	48	0	5	572	350	386	3	0	0	35	19	14	57	48	24	12	6	11	5	1	3	0	0	0	8	8	0.327		
	(b)	30	0	2	290	570	377	10	1	16	39	23	16	59	45	43	22	8	25	9	5	6	6	0	0	9	7	0.332		
Cerber	(c)	13	5	20	430	315	491	18	0	5	38	29	17	51	73	32	21	18	26	0	2	1	0	0	0	8	4	0.295	0.748	
	(a)	8	0	1	61	53	49	1116	7	0	24	22	20	51	20	52	54	4	36	1	0	0	0	12	0	8	0	0.737		
	(b)	7	51	24	0	15	3	11	1289	5	5	3	7	2	36	11	8	40	17	0	4	0	80	0	2	0	0	0.826		
WannaCry	(c)	0	0	69	0	1	0	0	16	1404	0	5	37	0	44	13	3	13	2	0	0	0	13	0	0	0	0	0.880	0.844	
	(a)	15	1	1	109	53	42	17	3	0	653	206	165	96	47	54	72	30	25	6	11	5	1	0	0	0	0	0.379		
	(b)	11	5	3	75	46	43	28	1	1	278	678	152	63	93	46	29	26	22	7	9	2	0	0	0	0	0	0.405		0.982
GandCrab4	(c)	4	0	2	73	63	63	25	1	14	337	290	365	48	83	159	44	26	17	2	2	1	0	0	0	0	0	0.261	0.659	
	(a)	41	2	5	67	89	57	69	2	0	145	73	99	489	174	155	49	19	40	2	2	2	2	0	1	14	14	0.316		
	(b)	3	6	10	62	47	45	28	18	5	61	135	57	173	699	161	24	20	23	2	4	1	0	0	2	29	0	0.418		
Ryuk	(c)	21	6	4	59	83	53	25	16	12	90	66	119	202	229	550	28	25	37	1	5	0	3	0	13	30	1	0.348	0.601	
	(a)	34	3	7	22	39	25	43	2	0	50	47	46	48	12	35	719	191	224	3	5	2	0	0	0	18	19	0.448		
	(b)	37	20	3	10	14	15	5	25	4	18	59	9	29	43	40	171	879	167	1	14	4	7	0	6	15	24	0.543		
Sodinokibi	(c)	24	15	12	10	32	31	13	6	6	41	43	30	59	47	61	288	249	597	4	3	3	0	0	0	13	13	0.404	0.748	
	(a)	2	1	0	2	2	0	7	0	0	9	2	1	1	0	1	2	2	3	723	407	455	0	0	0	0	0	0.410		
	(b)	2	0	0	0	0	0	4	0	0	1	0	0	0	5	0	0	4	5	585	636	370	0	2	0	6	0	0.401		
Darkside	(c)	5	0	1	0	0	0	3	0	0	2	10	2	5	0	3	9	4	4	555	425	585	0	0	0	7	0	0.382	0.974	
AESCrypt		11	0	6	0	1	0	0	33	6	1	0	0	0	0	0	0	0	1	0	0	0	1517	0	35	0	0	0.911	0.906	
SDelete		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1620	0	0	0	0.996	0.997	
Zip		2	0	0	0	0	0	1	0	0	0	0	0	5	0	0	1	1	2	0	0	0	39	0	1565	0	0	0.966	0.967	0.922
Excel		23	0	9	7	7	5	2	0	0	0	0	0	23	22	19	5	17	31	3	13	2	0	0	0	1421	0	0.883	0.880	
Firefox		115	1	1	5	14	2	0	0	0	0	0	0	17	3	16	27	9	6	0	0	0	0	0	0	1	1282	0.833	0.819	

Ransomware	Benign

Figure 4.3: Reproduction: Confusion matrix of seven ransomware and five benign software on Windows 7.

similar. For example, poor distinguishing capabilities of Cerber, GandCrab v4, Ryuk, Sodinokibi, and Darkside in setup (a), (b), and (c) can be observed in both confusion matrices. Additionally, both confusion matrices show high F1-Scores in detecting benign workloads in 26, 12, and two classes. Only minor differences can be observed. For example, the exact number of TeslaCrypt on setup (a) that has been classified as TeslaCrypt setup (a) does not match. This is also true for other examples in the confusion matrix. These minor differences can be explained by the split of data in the training and testing sets. Hirano et al. stated in their work that half of the files were used for training and the other half for testing [11]. This procedure was followed in the reproduction. Since a random choice of a training and testing set can lead to differences in performance, these minor differences can be explained by that fact. Overall, the conclusion is that the Hirano et al. results can be reproduced and validated using the developed code in this master project.

4.2 Design

This section describes the design choices made in this master project on the selection of ransomware, features and machine learning models.

4.2.1 Ransomware Selection

This subsection explains the choice of ransomware samples for this master project. The ransomware samples of Black Basta, Lockfile, WannaCry and Conti were downloaded form MalwareBazaar which is a global database storing and sharing malware samples with the community. For Lockbit and Sodinokibi, the ransomware samples previously collected by IBM were used [62].

Table 4.3 shows the six ransomware selected for this project and their SHA256 key.

Table 4.3: Chosen Ransomware and SHA256

Ransomware	SHA256	Encryption Algorithm
Sodinokibi	93ce973daa9687f185966b3133f700300665 5ec9d5bf3edb881efaf0e4fbafc7	AES and Salsa20 [63]
Black Basta	5d2204f3a20e163120f52a2e3595db198900 50b2faa96c6cba6b094b0a52b0aa	ChaCha20 [64]
Lockbit 2.0	1adf694880ec194a166905cf1756cdb48ec9 d98b6faaf7cd5d756c97bce93844	AES [65]
Lockfile	bf315c9c064b887ee3276e1342d43637d8c0 e067260946db45942f39b970d7ce	intermittend encryption [66]
WannaCry	6e51fa23db7d2f83f3d380d6d465e3262179 3d8c50e6bde255d996c25288c044	RSA and AES [67]
Conti	e22ff5594ab357da993651bf7decf035400d4 2bf37cca943495e1ac9bd7721c4	AES-256 [68]

Sodinokibi

Sodinokibi also named as Revil has been chosen since it made up 37 % of the ransomware attacks observed by X-Force in 2021 [3] and is still very active in 2022 [69]. This makes Sodinokibi one of the most prevalent used ransomware threats and an important target for ransomware research.

Sodinokibi has a variety of functionalities and uses advanced encryption techniques. But first, Sodinokibi needs to get system privileges [69]. It does so by either exploiting CVE-2018-845, an elevation privilege vulnerability existing on windows machines, where the Win32k component fails to properly handle objects in memory and allows to run arbitrary code in kernel mode or by running in admin mode [70]. Sodinokibi then continues to collect basic system and user data. If it finds that the keyboard layout is set to Russian, Ukrainian, or 18 other languages from post-USSR territories, it terminates the process and does not attack the victim's machine. This fact supports the claim that the authors of Sodinokibi come from a post-USSR country. If the keyboard layout is set to another language, it encrypts files on the victim's machine using Salsa20 and additionally encrypts keys with curve25519/AES-256-CTR. Further, Sodinokibi also utilized the command and control server obfuscation technique and it can operate without having internet access. The ransom note is left on the screen and victims are demanded to pay a ransom to encrypt their files on a website. Victims can use a trial decryptor, created by Sodinokibis authors, at the domain decryptor.top, to decrypt three images for free. It is important to note that in addition to attacking the victim's machine, Sodinokibi also tries to encrypt files on network shares [69].

Sodinokibi is known to be distributed by malicious spam campaigns. It is also spread by compromised managed service providers and utilizes the CVE-2019-2725 vulnerability combined with the RIG exploit kit. Since 2019, it is estimated that Sodinokibi has infected millions of machines and also collected millions of ransom. [69].

Black Basta

Black Basta belongs to the newcomers of ransomware families and has first been detected in April 2022. Even though this ransomware is rather novel, the developers behind Black Basta seem to have been involved with other known ransomwares [64]. A recent attack targeted the American Dental Association (ADA) which rose attention. The ransomware deletes volume shadow copies, hijacks Windows servers and encrypts the users' files using the ChaCha20 algorithm, and encrypts the ChaCha20 key using RSA-4096. According to Micro Trend Research [64], Black Basta is an example of a Ransomware-as-a-Service (RaaS) ransomware and can be bought or rented from the dark web. The paid ransom of a successful attack is split between the developers of the ransomware and the affiliates which spread the ransomware and infected the victim's device. However, unlike most modern ransomware, Black Basta has a more targeted approach and uses spear-phishing to target a specific person or group. For instance, they include current events or financial documents, known to be of interest to the target, to increase their efficiency. Since Black

Basta is currently active and becoming more prevalent, we decided to include it for the training of our machine learning model.

Lockbit

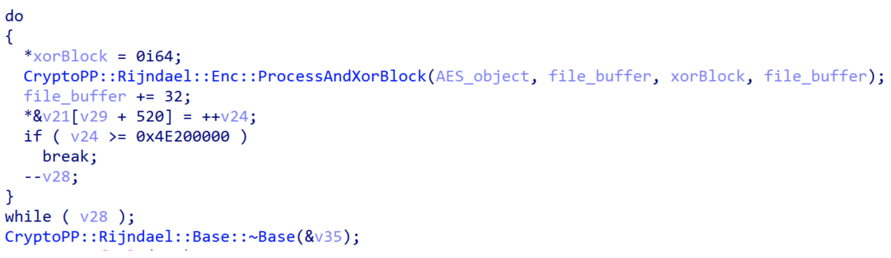
Lockbit first appeared in September 2019 [71] and is also an example of a Ransomware-as-a-Service (RaaS) ransomware. It has gained attention after the attack on Accenture in 2021 [72]. Just like any other ransomware, Lockbit also encrypts the user's files and demands a ransom payment to decrypt them. Interestingly, it only encrypts the first 4KB in place of each file to improve the speed of execution. It uses AES for file encryption and RSA encryption to encrypt the AES key. The AES Key is generated using BCryptGenRandom [72]. Lockbit focuses on business enterprises and government organizations and demands high payouts making it a high profile persistent thread.

Since 2019 this particular ransomware continued its level of activity and the developers came up with new versions. The updated Lockbit version demonstrates both, consistent and versatile operations. Therefore, we have chosen to investigate this ransomware further. In contrast to other ransomware, Lockbit 2.0 (first appearance in June 2021) applies a simultaneous attack and spreading phase [72]. Recent versions can disable safety prompts and employ a double extortion scheme. This double thread mechanism not only intimidates victims to delete the confidential data if the ransom is not paid but also threatens with the public release of the stolen data on the dark web [71]. During the spreading phase, it scans the local network to find additional Windows computers to infect. This self-propagating aspect ensures fast spreading and makes the ransomware even more dangerous. The latest Lockbit 3.0 version (appearance in June 2022) uses anti-analysis techniques to hide itself, requires a password to be executed, and has high similarity to DarkSide and BlackMatter. Lockbit version 2.0 is included in the ransomware selection since this version comprises more sophisticated mechanisms as mentioned above compared to version 1.0. Additionally, the database history from MalwareBazaar [73] shows frequent uploads of version 2.0 samples (*status: 02.12.2022*) which indicates its high prevalence.

LockFile

LockFile is a frequent and new ransomware family first seen in July 2021. It uses intermittent encryption methods that have not been seen before in ransomware attacks where it only encrypts every other 16 bytes of a file, resulting in an encrypted document looking statistically closer to the unencrypted original. This is a method used to avoid being detected by prevalent statistic-based ransomware detection methods and also boosts the encryption speed. Figure 4.4 shows the main encryption loop, which iterates through the whole file in memory [66].

In addition to intermittent encryption, LockFile ransomware uses memory-mapped I/O to encrypt a file. This allows the ransomware to minimize disk I/O that detection mechanisms could spot. After encryption, the files have a .lockfile extension and can not be



```
do
{
  *xorBlock = 0i64;
  CryptoPP::Rijndael::Enc::ProcessAndXorBlock(AES_object, file_buffer, xorBlock, file_buffer);
  file_buffer += 32;
  *&v21[v29 + 520] = ++v24;
  if ( v24 >= 0x4E200000 )
    break;
  --v28;
}
while ( v28 );
CryptoPP::Rijndael::Base::~Base(&v35);
```

Figure 4.4: Main encryption loop of Lockfile ransomware [66].

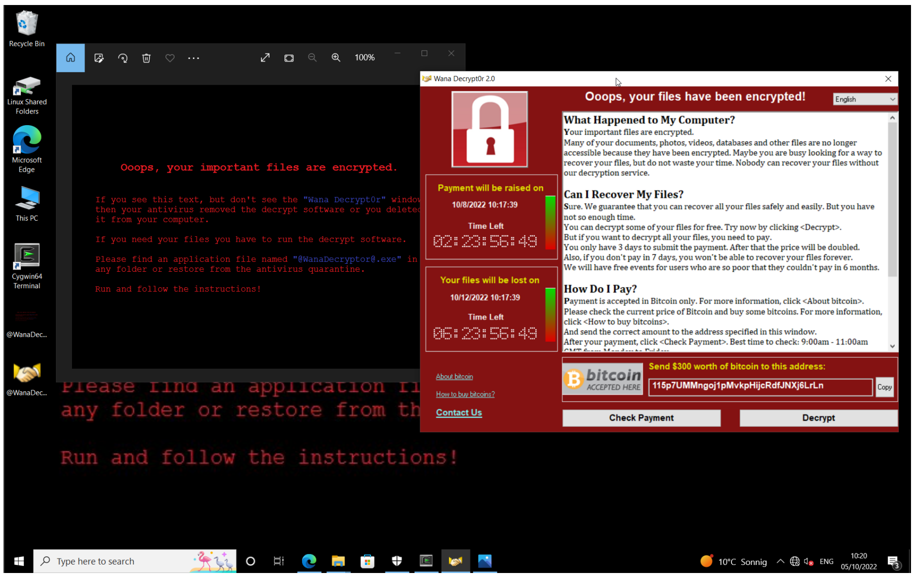
opened. To infect victims, LockFile ransomware exploits the ProxyShell vulnerabilities in Microsoft Exchange servers followed by a PetitPotam BTLM relay attack to gain control of the domain [66]. Like most ransomware, Lockfile leaves a ransom note on the victim's desktop asking for a ransom to be paid to restore the files. Fortunately, on November 3rd, 2021, Avast released a free decryption tool capable of restoring data encrypted by LockFile and the tool can be downloaded on Avast's official website [74]. Despite having a free decryption tool available for LockFile, the new intermittent encryption method makes it an interesting research topic, since more ransomware families using it to avoid detection are likely to arise in near future.

WannaCry

WannaCry is a type of malware which is categorized as crypto-ransomware. It encrypts user's files and requests for money in order to decrpyt the files. This ransomware also has different names like Dharma or Ryuk. In addition to infect the host system, this particular malware also utilizes an SMB vulnerability to spread among other computers in the same network. SMB is the Server Message Block protocol for Windows and enables sharing files or resources like printers over a network. It can also be used to access various remote services in Windows [75].

According to [75, 76] the execution process for WannaCry is as follows. At first WannaCry tries to encrypt all the user's files and after it has done so it sends an HTTP GET request to several pre-defined domains and halts the encryption process if it was able to reach these pre-defined domains. Afterwards, it tries to contaminate other computers on the same network by scanning for port 445 (Microsoft CIFS) and trying a known SMBv1 vulnerability (i.e., the EternalBlue vulnerability) [77, 78, 79] to gain access other computers in the same network. At the same time in the local system, it tries to extract its binary executable and configuration files. These extracted files are hidden from the anti-virus software. Once, the files are extracted, it starts with the encryption process, and encrypts user files and finally adds the ransomware note by changing the wallpaper. Furthermore, it also shows a window with more information on the time and a countdown before which it expects the ransom to be paid. Figure 4.5 shows this ransom note.

According to the dynamic analysis [79, 75, 76], to remain persistent on the compromised system, WannaCry does the following actions: The Windows AutoRun function is made use of by the malware to achieve memory persistence. This creates a new Windows registry item and consequently runs on every reboot of the system. Then, it removes all the backup files and directories on the system. Additionally, it grants itself complete access to all files on the computer via the Windows icacls command. Moreover, if there are any databases on the system like SQL or MS Exchange, it also tries to delete them. Finally, it creates a shortcut on the desktop of the @WanaDecryptor@.exe decrypter file.



Ooops, your important files are encrypted.

If you see this text, but don't see the "Wana Decrypt0r" windo
then your antivirus removed the decrypt software or you delete
it from your computer.

If you need your files you have to run the decrypt software.

Please find an application file named "@WanaDecryptor@.exe" in
any folder or restore from the antivirus quarantine.

Run and follow the instructions!

Ooops, your files have been encrypted!

What Happened to My Computer?
Your important files are encrypted.
Many of your documents, photos, videos, databases and other files are no longer accessible because they have been encrypted. Maybe you are busy looking for a way to recover your files, but do not waste your time. Nobody can recover your files without our decryption service.

Can I Recover My Files?
Sure. We guarantee that you can recover all your files safely and easily. But you have not so enough time.
You can decrypt some of your files for free. Try now by clicking <Decrypt>.
But if you want to decrypt all your files, you need to pay.
You only have 3 days to submit the payment. After that the price will be doubled.
Also, if you don't pay in 7 days, you won't be able to recover your files forever.
We will have free events for users who are so poor that they couldn't pay in 6 months.

How Do I Pay?
Payment is accepted in Bitcoin only. For more information, click <About bitcoin>.
Please check the current price of Bitcoin and buy some bitcoins. For more information, click <How to buy bitcoins>.
And send the correct amount to the address specified in this window.
After your payment, click <Check Payment>. Best time to check: 9:00am - 11:00am

Payment will be raised on
10/8/2022 10:17:39
Time Left
02:23:56:49

Your files will be lost on
10/12/2022 10:17:39
Time Left
06:23:56:49

Send $300 worth of bitcoin to this address:
115p7UMMngoj1pMvkpHijcRdfJNXj6LrLn

Figure 4.5: WannaCry Ransomnote.

Conti

Conti is first observed in 2020 and uses mainly phishing attacks to infect victims' devices. During the infection, Conti installs a Trickbot or BazarLoader Trojan in order to obtain remote access and set up a backdoor to the command-and-control server [3, 80]. The established backdoor allows the monitoring of network traffic, which gives insights into the recovery approaches of the infected device. Further, this ransomware accelerates its spreading phase by using multithreading, which emphasizes the importance of quick detection and efficient countermeasures. Similar to the other selected ransomware, Conti employs a double extortion scheme. Conti's most famous attacks targeted Ireland's Health Service where they were forced to shut down all IT services causing serious problems [80]. According to an article in The Irish Times [2] a significant increase in phone calls from scammers has been detected claiming to be from Ireland's Health Service with the aim to put people under pressure and force them to pay money. Ireland's Health Service confirmed that on the 28th of May 2021, medical information for 520 patients was leaked on publicly accessible websites. They estimated the total damage to be at least 100 million euros.

4.2.2 Feature Selection

This subsection discusses the initial feature set that has been proposed in this master project in order to distinguish benign workload from ransomware. These features are analysed in 5.2.

Hirano et al. [10] showed that ransomware can be distinguished from benign workloads using supervised machine learning models trained with only five features extracted from block-level IO operations. The features are summarized in Table 4.1 and the corresponding formulas to compute them can be found in their original paper. The five features will be reused in this work as a starting point and are computed over a window size t in seconds. The code used to compute the features is described in Subsection 4.3.5.

DNNs and XGBoost will be used in this project additionally to distinguish ransomware from benign workloads and to achieve improvements in generalizability to unseen ransomware and benign software. As these models might show better performance with more features, the following features are used in addition to the five features introduced by Hirano et al. [10]. These features are also computed over a time window t in seconds.

Table 4.4: Additional Features for Machine Learning Models.

Id	Feature	reads/writes
A	Variance of entropy	writes
B	Rewrite rate	both
C	Mean entropy of rewrites	both
D	Difference of variance of LBA writes and variance of LBA reads	both
E	Slope of entropy	writes
F	Kurtosis of entropy	writes
G	Kurtosis of LBA	reads
H	Kurtosis of LBA	writes
I	Stacking of multiple windows	both

Feature A: Variance of Entropy of Writes

The variance of the entropy of writes $V(t)$ in window t is computed by Equation 4.4 where N denotes the number of write IO operations in that window of size T_{window}.

$$V(t) = \frac{1}{N-1}\sum_{i=1}^{N}(H_i(s,t) - H_{write}(t))^2 \tag{4.4}$$

The concept of entropy has been introduced in Section 4.1. Only paying attention to the mean entropy value in a specific window size results in an information loss. Therefore, the variance of the entropy is included as an additional feature. This is especially relevant in real-world environments, where benign workloads and ransomware attacks are executed at the same time. An alternative approach would be creating a histogram of the entropy and using the bins as features.

Feature B. Rewrite Rate

The rewrite rate corresponds to the number of read IO operations to which a subsequent write operation is made to the same LBA location within a short time interval (i.e., the number of reads followed by a write to the same LBA). The rewrite rate is calculated for every window t. To simplify the implementation, only read and write operations happening in the same window are counted in the rewrite rate. When the write operation happens in a different window than the read operation it will not be counted as a rewrite. This feature captures the different read and write behavior between ransomware and benign workloads of first reading every file, then writing it in place or doing read and write in place directly one after another for each file processed. This feature is only relevant for ransomware which does overwrite files, meaning doing writes in place [49]. Writes out of place are not considered as a rewrite.

Feature C. Mean Entropy of Rewrites

The mean entropy of rewrites is the mean of the entropy of the writes in a specific window of size t where there is also a read to the same LBA that occurs earlier in time than the write. The entropy of rewrites indicates the uncertainty of rewrite IO operations.

Feature D. Difference of Variance of LBA Writes and Variance of LBA Reads

The difference between the variance of LBA writes and LBA reads is computed by Equation 4.5 for each window t in seconds and $V_{LBAreads}$ and $V_{LBAwrites}$ are computed by Equation 4.6 for reads and writes respectively. In Equation 4.6, t denoted the window size in seconds, N the total number of IO's in the window, and LBA_i is the corresponding LBA for each IO. The time series plots of the LBA variance of reads and writes in Hirano et al. show that the difference between read and write LBA variance is small for ransomware traces and high for the benign workload. This makes it an interesting feature to distinguish ransomware from benign workloads.

$$Diff(V_{LBAwrites}(t), V_{LBAreads}(t)) = V_{LBAwrites}(t) - V_{LBAreads}(t) \tag{4.5}$$

$$V(t) = \frac{1}{N-1} \sum_{i=1}^{N} \left(LBA_i(t) - \overline{LBA(t)} \right)^2 \tag{4.6}$$

Feature E. The Slope of Entropy

The slope of entropy measures the rate of change in the entropy for write IO's and is computed for a window of size t. This feature can detect trends of raising or falling entropy values. The slope $m(t)$ is computed according to the formula 4.7 by fitting a linear least squares model where N is the number of write IO's in the window size t, x_i is the write IO's timestamp and y_i the corresponding entropy value. $\bar{x}$ denotes the mean of all the timestamps in the window and $\bar{y_i}$ is the mean of the corresponding entropy values.

$$m(t) = \frac{\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{N}(x_i - \bar{x})^2} \tag{4.7}$$

Feature F. Kurtosis of Entropy

Kurtosis is a measure of whether the data are heavy-tailed or light-tailed relative to a normal distribution. Data sets with high kurtosis tend to have heavy tails or outliers. Data sets with low kurtosis tend to have light tails or lack of outliers. A uniform distribution would be the extreme case. The kurtosis of the entropy is computed using Fisher's Kurotsis by subtracting Pearson's Kurtosis by three, where E is the expectation operator, μ_4 is the fourth central moment and σ is the standard deviation [81].

$$Kurt[X] = E[(\frac{x-\mu}{\sigma})^4] = \frac{E[(X-\mu)^4]}{(E[(X-\mu)^2])^2} = \frac{\mu_4}{\sigma^4} \tag{4.8}$$

Features G. and H. Kurtosis of LBA Reads and Kurtosis of LBA Writes

The same method and formula as explained in *F.* are applied for the calculation of the kurtosis of LBA reads and writes. The difference is that it is applied to the LBA and not to the entropy.

Feature I. Stacking of Multiple Windows

The idea of stacking feature vectors from multiple N windows follows the idea of including the time dimension into one input vector and helps to detect changes over time. It allows for increasing the feature space and enables the use of more sophisticated ML models. Further, it also allows using smaller window sizes t whilst still covering the same time range. One feature vector includes all the features explained in A - H and the features presented by Hirano et al. [11] computed over a specific window size t and N of such windows are stacked horizontally and fed into the machine learning model as one feature-vector.

4.2.3 Model Selection and Parametrization

As shown in Section 3 existing work on ransomware detection in storage systems used Random Forest as a model. Expanding from the existing research, this master project evaluates the performance of two additional models on the binary classification task of distinguishing ransomware from benign workload, namely XGboost and DNN. However, hyperparameter tuning is not a part of this project and could be performed in future work to further increase the performance.

Random Forest

As the name implies, the Random Forest model is an extension to the classical decision tree algorithm where the dataset is recursively split in each decision node until a leaf node with the corresponding label is reached. Hereby, the tree structure is defined by maximizing information gain which is calculated using entropy. However, decision trees are highly sensitive to its training data which could result in high variance. Therefore, Random Forest is a collection of multiple random decision trees, which makes it less sensitive to the training data and significantly improves the prediction accuracy [82]. This is achieved through bagging, a combination of bootstrapping, where each tree is trained with only a subset of the original dataset and aggregation, where the prediction of a test sample is conducted through majority voting. Further, Random Forest is relatively fast to train and to predict and does only depend on few tuning parameters. Also, the model gives a fast estimate of generalization error and provides a way to estimate variable importance using the *feature_importances_* attribute. Regarding the implementation, the *RandomForestClassifier* from *sklearn.ensemble* [83] is used with a value of ten for the *n_estimators* parameter.

XGBoost

XGBoost stands for Extreme Gradient Boosting and is a promising machine learning model, which implies tree boosting and has proven to be highly effective for a variety of different problem sets by winning several Kaggle data competitions. According to Nvidia [84], the worlds largest producer of graphic processors, XGBoost ”is the leading machine learning library for regression, classification, and ranking problems” also because its parallel tree boosting, which has been proven by different benchmarking studies and takes the bias-variance trade-off into consideration. The model is implemented using the XGBoost Python package by the Distributed (Deep) Machine Learning Community with the following arguments in the constructor: objective=”binary:logistic”, random_state=42, scale_pos_weight=99, eval_metric=”map”.

DNN

The third model is a seven layer DNN with the Categorical Crossentropy loss function and the ADAM optimizer. The data is fed to the first layer as a twelve dimensional input vector and after each (except the last) layer, an Exponential Linear Unit (ELU) activation function is applied. The softmax activation function in the last layer normalizes the output of the network to a probability distribution over the predicted output classes. The implementation is conducted using the Sequential model from *keras.models*, an Open-Source deep learning library [85]. For the training of 40 epochs, a batch size of 64 is used. The parametrization of the DNN is based on the findings from Sewak et al. [18] paper in *Comparison of Deep Learning and the Classical Machine Learning Algorithm for the Malware Detection*. DNNs are often used in language translation and image recognition or search tools because of their ability for deep abstraction of correlations between input data and output data. In order to truly benefit from its strengths, an immense dataset is needed for training the model [86].

4.3 Implementation

This section elaborates on the test environment used to collect storage access patterns from ransomware. Further, the script used to simulate benign workload is explained and all the collected storage access patterns are listed. The feature extraction process as well as training and evaluation of models is explained and the code implementation is elaborated.

4.3.1 Test Environment for Feature Extraction on IO Operations

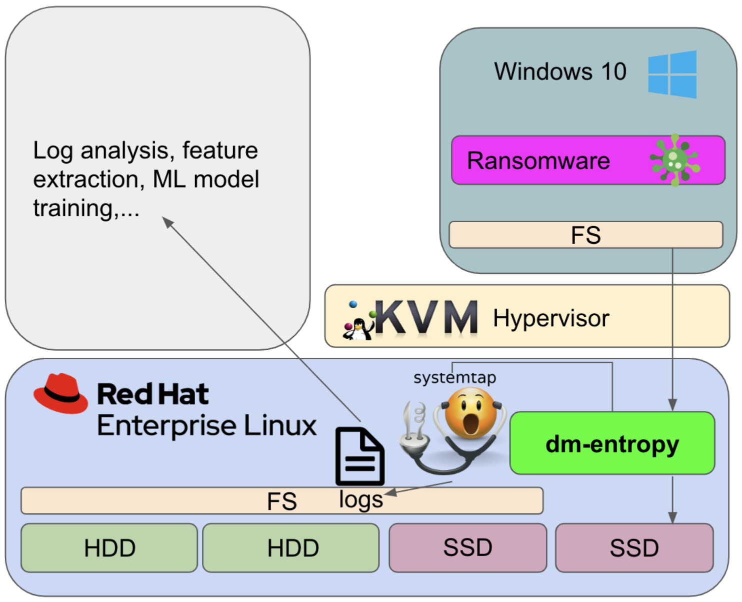


Figure 4.6: KVM-based test environment

In order to run the ransomware workloads and collect traces from it, we have installed and configured two servers each providing a test environment for running benign and ransomware workloads. Figure 4.6 illustrates the test environment used. Both servers are running Red Hat Enterprise Linux (RHEL) 9 with Linux kernel version *5.14.0-70.26.1.el9_0.x86_64*, on top of which we are using the KVM hypervisor to achieve the necessary isolation from the host server to run VMs that run the workloads to generate IO traffic. In contrast to the WaybackVisor hypervisor used by Hirano et al. [10] which automatically saves all the IO traces on to a Hadoop cluster, we decided to use the KVM hypervisor because the WaybackVisor hypervisor is not publicly available. Additionally, the KVM hypervisor also provides advanced virtio functionality. On these servers there are four block devices. Two HDDs and two SSDs.

1. 2× WDC WD10EALX-009BA0 (HDD, ≈ 1 TB)
2. 2× Samsung SSD 850 PRO 512GB (SSD, ≈ 500 GB)

The first HDD is being used to run the actual RHEL 9 host operating system (OS). Each of the two SSDs can be used to serve storage to virtual machines that run the ransomware or benign workloads. The use of dedicated SSDs eases the interception and tracing of only those IO operations that are issued by the guest VM. The remaining and final HDD is mounted in the host OS and used solely for storing the collected traces.

Preparation of SSDs for the VMs

In order to prepare the SSDs for a VM to run on, we follow these steps. We created images for the different partitions along with the MBR and the partition table from a reference VM with installed necessary libraries that is needed to run a specific workload.

1. Clean the block device completely

   ```
   blkdiscard -f /dev/sdb
   ```

2. Write the Master Boot Record (MBR) to the beginning of the block device.

   ```
   dd if=mbr.dat of=/dev/sdb bs=512 count=1
   ```

3. Change the partition table of the block device with the 'sfdisk' command.

   ```
   sfdisk /dev/sdb < sfdisk_partition_table_sdb.dat
   ```

4. Clone each partition with the 'partclone' command. Since in our case we would be running Windows VMs, we clone each of the the partition with the New Technology File System (NTFS) format, 'partclone.ntfs'.

   ```
   for i in 1 2 3 5;
       do partclone.ntfs -r -d -s sdb$i -o /dev/sdb$i;
       date;
   done
   ```

Entropy extraction in the Device Mapper Kernel Module

We used the "dmentropy" kernel module provided by our project supervisor to extract entropy information. This module is designed as a device mapper shim layer and is responsible for intercepting the IO operations and calculating the Shannon entropy [51] of each individual IO operation. The calculation of the Shannon entropy that is implemented here is slightly different from the original definition [51]. This is because the Shannon entropy of characters is a value between 0 and 8. However, in the Linux kernel, no floating point operations can be performed. Therefore, the entropy value is calculated using integer values in a range between 0 and 1000 (per-mille). This enables us to collect the I/O traces with SystemTap with the Shannon entropy information in per-mille for every write operation. It is only possible to collect the Shannon entropy for write IOs and not for read IOs. The code listing 4.1 describes how the calculation of the Shannon entropy per-mille is implemented.

With the "*dm-entropy*" kernel module inserted into the Linux kernel we configure the device mapper "*sirius_win10_dev*" with this inserted kernel module between a specific actual block device and file system of the VM running. And with the help of SystemTap we defined kernel probes for tracing the entropy (per-mille) of every write operation. The code listing 4.2 describes how the "*dm-entropy*" device mapper is designed as a shim layer for Shannon entropy tracing of write operations.

Code Listing 4.1: Entropy extraction code implemented in the Linux device mapper.

```c
/*
 * clz32 - Count the leading number of zeros
 * @param i (IN) 32-bit number to count leading zeros
 * @return number of leading zeros. 32 if no bit set, 0 MSB set, 31 LSB set only
 */
int clz32(uint32_t i)
{
        if (i == 0) {
                return 32;
        }

        return __builtin_clz(i);
}

/*
 * ilog2 - Takes a value and returns the integer log2 value rounded down
 * @param value (IN) value to calculate log 2 for
 * @return integer log2 value rounded down
 */
int ilog2(uint32_t value)
{
        if (value == 0) return 0;
        return(32 - __builtin_clz(value) - 1);
}

#define CHAR_BINS       (0x1 << CHAR_BIT)
#define LOG2_EBS

int64_t _get_entropy(const char *data, const unsigned int len)
{
        /*
         * Note, if len is not a power of 2, then the calculated entropy
         * is lower than it actually is as the ilog2() only looks at the
         * first bit set.
         */
        uint16_t c[CHAR_BINS] = { 0 };
        unsigned int pos;
        int64_t e = 0;

        if (data == NULL) return 0;

        for (pos = 0; pos < len; pos++) {
                c[(uint8_t)*(data + pos)]++;
        }

        for (pos = 0; pos < CHAR_BINS; pos++) {
                e += (ilog2(c[pos]) - ilog2(len)) * -c[pos];
        }

        return (e);
}

int64_t _get_entropy_4k(const char *data)
{
        uint16_t c[CHAR_BINS] = { 0 };
        unsigned int pos;
        int64_t e = 0;

        if (data == NULL) return 0;

        for (pos = 0; pos < ENTROPY_BLOCK_SIZE; pos++) {
                c[(uint8_t)*(data + pos)]++;
        }

        for (pos = 0; pos < CHAR_BINS; pos++) {
                /*
                 * Given:
                 *   p_x = symbol probability of symbol x
                 *   e_x = the entropy contribution from symbol x
                 *   c_x = Number of occurrences of symbol x
                 *   X = number of symbols in the data
                 *     = 2^ENTROPY_BLOCK_BITS for 4KB block
                 *
                 * The Shannon entropy is calculated as follows:
                 *   e_x = -p_x * log2(p_x)
                 *   e   = sum(e_x for x=0->X)
                 * Hence, the returned result must be divided by X.
                 *
                 * In order to keep the entropy value returned as an integer,
                 * this is transformed to:
                 *   e'_x = -c_x * log2(p_x)
                 *   e'   = sum(e'_x for x=0->X)
                 *
                 * Below, (ilog2(c[pos]) - ENTROPY_BLOCK_BITS) corresponds to
                 * the symbol probability log2(p_x).  As the probability is
                 * at most 1, the log2() of the symbol probability will be a
                 * negative number.
                 *
                 * Using
                 *   p_x = c_x / X
                 * we simplify log2(p_x):
                 *   log2(p_x) = log2(c_x/X)
                 *             = log2(c_x) - log2(2^ENTROPY_BLOCK_BITS)
                 *             = log2(c_x) - ENTROPY_BLOCK_BITS
                 *
                 */
                e += (ilog2(c[pos]) - ENTROPY_BLOCK_BITS) * -c[pos];
        }
        return (e);
}

/**
 * Get block entropy in range 0 to 8
 * @param data (IN)
 * @param len  (IN)
 * @return entropy in range 0 to 8
 */
unsigned int get_entropy(const char *data, const unsigned int len)
{
        int64_t e = 0;
        e = MAX(0, _get_entropy(data, len));
        e /= len;
        return((unsigned int)e);
}

/**
 * Get entropy in per-mille
 * The calculated entropy is approximative and might be ~5% higher than in
 * reality due to the log2 being approximated by counting leading zeros.
 * @param data (IN)
 * @param len  (IN)
 * @return entropy as integer in per-mille
 */
unsigned int get_entropy_pm(const char *data, const unsigned int len)
{
        int64_t e = 0;
        e = MAX(0, _get_entropy(data, len)) * 125;
        e /= len;
        return((unsigned int)MIN(1000, e));
}
```

Code Listing 4.2: The "*dm-entropy*" device mapper designed as a shim layer for Shannon entropy tracing of write operations.

```
#pragma GCC optimize ("O0")
noinline void trace_entropy(struct bio_vec bv, struct bvec_iter iter, struct bio *bio,
     unsigned int e)
{
}
#pragma GCC optimize ("O2")
/*
 * entropy_target_map - Handles new bio requests to extract entropy information
 * @ti:         The dm_target structure representing the entropy target
 * @bio:        The block I/O request from upper layer
 * Returns:
 *  DM_MAPIO_SUBMITTED :  The target has submitted the bio request to underlying
    request.
 */
static int entropy_target_map(struct dm_target *ti, struct bio *bio)
{
        struct entropy_dm_target *edt = (struct entropy_dm_target *) ti->private;
#ifdef VANILLA_KERNEL
        unsigned long flags;
#endif
        struct bio_vec bv;
        struct bvec_iter iter;
        unsigned int e;

        bio->bi_bdev = edt->dev->bdev;

        if(bio_data_dir(bio) == WRITE) {
                bio_for_each_segment(bv, bio, iter) {
#ifdef VANILLA_KERNEL
                        char *data = bvec_kmap_irq(&bv, &flags);
                        flush_dcache_page(bv.bv_page);
                        bvec_kunmap_irq(data, &flags);
#else
                        char *data = kmap_local_page(bv.bv_page);
                        e = get_entropy_pm(data, iter.bi_size);
                        kunmap_local(data);
                        trace_entropy(bv, iter, bio, e);
#endif
                }
        }
        if(bio_data_dir(bio) == READ) {
                bio_for_each_segment(bv, bio, iter) {
                        /* Cannot get entropy on reads here */
                        trace_entropy(bv, iter, bio, 0);
                }
        }
        submit_bio(bio);
        return DM_MAPIO_SUBMITTED;
}
```

Adding this “*sirius_win10_dev*” device mapper in between the file system and the block device also adds an overhead in terms of IO rate and latency impact. In an analysis before the start of this project these metrics had been characterised and measured with and without the device mapper. Using 100 % random reads at 16 KiB with no compression on an Intel 1TB SSD DC P4510 the performance penalty from trace collection was around 2.3 - 2.6$\times$ drop in throughput and a 2.6 - 3$\times$ increase in latency.

Collection of IO traces using SystemTap

SystemTap [87] uses a C-like scripting language for dynamically measuring and monitoring the running Linux kernel code. This enables system programmers to write scripts that inspect and diagnose the internal functioning of the Linux systems. The SystemTap script is compiled into a Linux kernel module which can then be inserted into the kernel to extract information. The advantage of using this scripting language is that it imposes certain limitations and special tests can be performed before the script is executed in the kernel [88], hence minimizing the risk to crash the kernel. The code listing 4.3 demonstrates how the SystemTap kernel probes are defined with the “*dm-entropy*” module.

Code Listing 4.3: SystemTap kernel probe definition.

```
/**
 * probe ioblock_trace.entropy - Fires whenever entropy of a block IO request had been
    extracted.of a block IO request.
 * @name       : name of the probe point
 * @devname    : device for which a buffer bounce was needed.
 * @ino        : i-node number of the mapped file
 * @rw         : binary trace for read/write request
 * @sector     :
 * @size       : total size in bytes transferred
 */
probe module("/home/ahu/dm/dmentropy.ko").function("trace_entropy")
{
//        name = "ioblock_trace.entropy"
        devname = __bio_devname($bio)
        if(devname == "nvme0n1") {
                sector = @choose_defined($bio->bi_iter->bi_sector, $bio->bi_sector)
                size = @choose_defined($bio->bi_iter->bi_size, $bio->bi_size)
                if (@defined($bio->bi_opf)) {
                        rw = bio_op($bio)
                }
                else {
                        rw = $bio->bi_rw
                }
                if((bio_rw_num(rw) == BIO_WRITE) || (bio_rw_num(rw) == BIO_READ)){
                        printf("%u %u %u %u %u\n", count_ms, bio_rw_num(rw), sector,
    size, $e)
                }
        }
}
```

Trace Collection Steps

These are the steps that are preformed upon collection of traces:

1. Preparation of the block device as described above. Check steps 1 through 4 of the “Preparation of SSDs for the VMs” in the section 4.3.1.

2. Entropy extraction of the whole block device before starting the VM (Needs to be run only once for each setup configuration, Setup 1 and Setup 2).

   ```
   ./ent -t -x 4096 -y `blockdev --getsize64 /dev/sdb2` /dev/sdb2 >
       entropy_sdb2_pre.csv
   ```

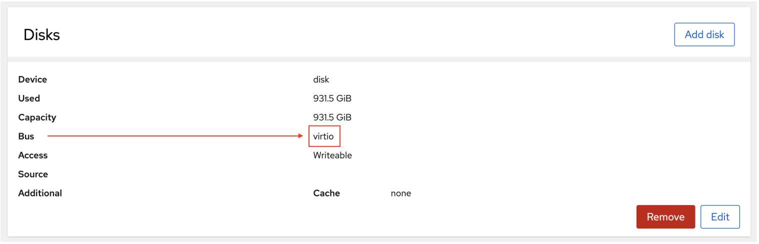
3. Setting up the device mapper block device for entropy extraction:
   First with the ‘insmod’ command we insert the compiled dmentropy module in the Linux kernel.

   ```
   sudo insmod ./dmentropy.ko
   ```

   and after that we create a device mapper block device ‘sirius_win10_dev’ on top a specific block device (in this case ‘sdb’) with the inserted ‘dmemtropy’ kernel module.

```
echo 0 $(sudo blockdev --getsize /dev/sdb) entropy /dev/sdb 0 | sudo
    dmsetup create sirius_win10_dev --readahead=none
```

4. RHEL Cockpit configurations for the VM: After installing the "cockpit" package with yum, the Cockpit server can be accessed at port number 9090. We set the bus type of the block device the VM is running on to 'virtio' since it is faster than the default 'sata'. See the figure 4.7 for reference.



Disks

Device	disk		
Used	931.5 GiB		
Capacity	931.5 GiB		
Bus	virtio		
Access	Writeable		
Source			
Additional	Cache	none	

Figure 4.7: Set the bus type of the block device to 'virtio'.

5. Starting the VM: The VM can be started by clicking on the 'Run' button in the RHEL Cockpit user interface.

6. Proper configuration of the VM:
   The first important step after turning on the VM is to turn off the Windows Defender settings. It might also happen that the Windows Defender gets turned on after few minutes, therefore, it is necessary to check that this setting is turned off before running the ransomware workloads.
   Secondly, the access right to the Govdocs directory should also be properly set such that the logged in user has full control over the files. This can be done as follows: Open Cygwin (as an administrator) and move to the path where the (Govdoc) data folder is located and execute the following commands to recursively adjust the access rights:

   ```
   chmod -R a+rwx data
   chmod -R ugo+rwx data
   ```

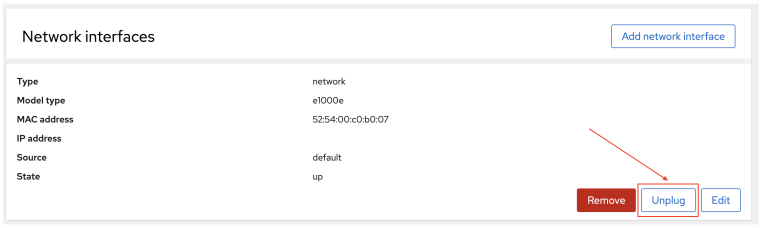
   Similarly, open the command prompt (CMD) (as an administrator) and go to the path where the (Govdoc) data folder is located and execute the following commands to recursively adjust the access rights:

   ```
   icacls data /grant zrl-rsd-user:(OI)(CI)F /T
   ```

   These commands are considering that the Govdocs files are saved in the directory 'data'.

7. Disabling network access to the running VM:
   On the host OS, we enable the firewall to disable all the network traffic from the VMs.

   ```
   firewall-cmd --zone=drop --add-source=192.168.122.0/24
   ```



Network interfaces

Type	network
Model type	e1000e
MAC address	52:54:00:c0:b0:07
IP address	
Source	default
State	up

Figure 4.8: Unplug of the Network Interface of the VM.

Additionally, we also unplug the network interface of the VM on RHEL Cockpit. See figure 4.8 for reference.

8. Starting the trace collection with SystemTap.

   ```
   sudo stap -v -g -o IO_trace_with_entropy.dat entropy-trace.stp
   ```

9. Running the ransomware:
   Depending of the type of the executable of the ransomware, we either double click the executable or we execute it from a 'Cygwin' terminal. In the latter case, we first turn the ransomware file into an executable by 'chmod'

   ```
   chmod +x <ransomware_file_name>
   ```

10. Stop the trace collection after reasonable amount of time: In our case we left the trace collection running for about an hour for all the workloads.

11. Shutdown the VM: This should be done from the from the GUI of the Windows VM and clicking on the 'Shutdown' and not from the RHEL Cockpit interface.

12. Post extraction of the entropy of the whole device.

    ```
    ./ent -t -x 4096 -y ‘blockdev --getsize64 /dev/sdb2‘  /dev/sdb2  >
        entropy_sdb2_post.csv
    ```

13. Proper deletion of the block device: Check step 1 of the "Preparation of SSDs for the VMs" in the 4.3.1 section.

4.3.2 Modification of the ENT tool

In order to extract the entropy of the whole block device on which the VM was running before and after running a ransomware sample, we modified the ENT tool [89]. Given a stream of bytes, from either an input file or the standard input, the tool applies various tests like Entropy, Chi-square Test, Arithmetic Mean, Monte Carlo Value for Pi and Serial Correlation Coefficient on these stream of bytes. Out of all these metrics our concern was only Entropy.

However, it was needed to calculate the entropy in a block size of 4096 bytes with the motivation that some ransomware like Lockbit encrypted data at 4KB chucks [65]. Therefore, initially the "dd" command was being using inside a for loop for the same. But, this turned out to be very slow and cumbersome to extract the entropy with block size of 4096 bytes for the complete SSD which is $\approx$ 500 GB. Consequently, we modified the source of the ENT code and added two new command line flags '-x' and '-y' corresponding of the block size and complete scan size of the whole SSD. This enabled to extract the entropy of the whole SSD within couple of minutes. Check step 2 and step 12 of 4.3.1 for running the ENT tool from the command line with the appropriate flags

4.3.3 Benign Workload Simulator

To extract storage access patterns simulating benign workloads executed on a computer, a file converter bash scripts was implemented. The script recursively traverses over the folders and files, located at the provided path, and converts the file types depicted in Table 4.5. Converting most common file types serves the goal to simulate benign workload commonly executed on a computer.

Table 4.5: Supported File Type Conversions.

Original File Type	Converted To File Type
.doc	.pdf
.docx	.pdf
.xls	.csv
.xlsx	.pdf
.jpeg	.png
.jpg	.png
.png	.jpg
.html	.pdf
.txt	.docx
.ppt	.pdf
.pptx	.pdf
.json	.csv

The file converter can be configured by passing different parameters. Users of the script have the possibility to either traverse folders and files located in the provided path randomly by setting the `-t` flag to 1, or sequentially otherwise. This provides the possibility to simulate benign software, since a person would convert files based on their perceived relevance, instead of going over the files in a strictly sequential manner. Further, the file converter can be configured, such that approximately 10 % of the files are only read instead of converted by setting the `-r` flag to 1. The *read only* capability again serves the purpose of simulating a person's or software's benign behavior on a computer. The last possible configuration is, to include pauses randomly after approximately every 10th file touched by setting the `-p` flag to 1. These pauses serve the purpose to simulate an environment where no specific actions are executed on the machine.

The benign workload storage access patterns were collected by running the benign file converter bash script with all the flags set to 1 and by passing the path to the govdocs dataset folder as argument. The storage access patterns were collected for one hour in the test environment and on both setups as explained in Subsection 4.3.1.

4.3.4 Collected Storage Access Patterns

The storage access patterns, also referred as traces, that were collected in this master project are depicted in Table 4.6. The storage access pattern have been collected by the process and test environment described in Subsection 4.3.1.

Table 4.6: Collected Storage Access Patterns.

Workload	Setup 1	Setup 2
Sodinokibi	x	x
BlackBasta	x	x
Lockbit 2.0	x	x
LockFile	x	x
WannaCry	x	x
Conti	x	x
File Converter (Benign)	x	x
File Converter (Benign) + Lockbit	x	
File Converter (Benign) + BlackBasta	x	
File Converter (Benign) + WanaCry	x	

4.3.5 Feature Extraction, Training and Evaluation of Models

Feature extraction and computation is implemented in Python 3.9. The feature extractor is implemented as a class and can be instantiated by passing the instance variables described in Table 4.7 to the constructor.

After instantiating an object of the feature extractor class, the *extract()* function is executed to extract the feature vectors. The *extract()* function then iterates over all combinations of window sizes and offsets which are specified in the according object attributes (values passed to the constructor) and creates Comma Separated Value (CSV) files in the specified directory. To create clarity and avoid confusion, each feature vector CSV file, containing features extracted over different window sizes and window offsets, follows an intuitive naming scheme. The *extract()* function automatically appends the endings `_ws{window_size}_wo{window_offset}` to each file. Then the function iterates over all dataframes containing read IO's and dataframes containing write IO's and calls the windowing() function. The *windowing()* function iterates over all combinations of window sizes and window offsets and computes the number of windows as well as the specific seconds that belong to each window for all combinations. Once, the number of windows is computed, the *windowing()* function fills the windows with the corresponding IO's by iterating over all the computed windows (e.g. with a window size of ten seconds and a window offset of one second, the first window goes from zero to ten seconds, the

Table 4.7: Feature Extractor: Instance Variables.

Instance Variable Name	Variable Type	Description
windows	list[int]	The window sized in milliseconds for which one wants to extract features; the list can contain one or multiple window sizes but must have the same length as offsets
offsets	list[int]	The window offsets by which each window is shifted; the list can contain one or multiple window sizes but must have the same length as windows
datasets_reads	list[pandas.DataFrame]	Dataframes per workload containing all the read IO's from the traces and labels
datasets_writes	list[pandas.DataFrame]	Dataframes per workload containing all the write IO's from the traces and labels
save_file	String	the path to the file where the final feature vectors will be stored
stacking	int	The number of windows that are stacked horizontally (feature I 4.4)

second window from one to 11 seconds and so on) and taking the corresponding IO's from the dataframes. With these IO's per window, features get computed by calling the *compute_features()* function. The implemented features in the feature extractor are the features presented by Hirano et al. [11] depicted in Table 4.1 and the newly suggested features in this master project depicted in Table 4.4. If stacking is set, the specified number of feature vectors, computed for all windows, are stacked horizontally. All the feature rows get concatenated and the final feature vector is written to the CSV file created in the beginning for each combination of window size and window offset. The feature extraction code can be viewed in Appendix A.3.

After the extraction of features per window, the dataset is split into five folds for each workload separately, where a fold contains 20 % of the windows. In the first split, the first fold is used as test data, and the last four folds as train data. In the second split, the second fold is used as test data and the first and last three are used as train data, and so on. Since overlapping windows are used in most of the experiments, one needs to ensure that no information is leaked between the train and the test data. Thus, the according windows are removed at the split point from the train data. How many windows have to be removed depends on the specific window size and window offset used. An illustration of this process is depicted in Figure 4.9. This example shows a five-fold split where fold one is used as test data and folds two to five are used as train data. Fold one and fold two are split between windows 18 and 19, which are depicted in orange. Windows 14 to 23 are computed using partly the same data (IO's) shown as hatched orange area. To avoid information leakage between the train and the test set, windows 19 to 23 are thus removed from the train set and windows 14 to 18 remain in the test set. Overlapping windows are removed from the train set since it contains much more data (80 %) compared to the test data (20 %) to ensure enough windows remain in the test set for accurate testing.

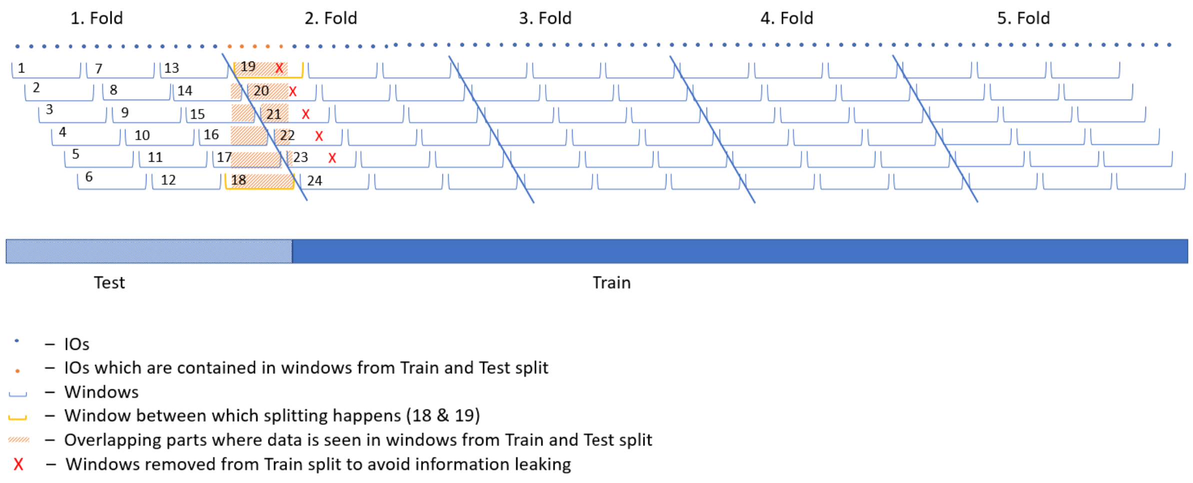

1. Fold
2. Fold
3. Fold
4. Fold
5. Fold
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
19 X 20 X 21 X 22 X 23 X 24
Test
Train
• – IOs
• – IOs which are contained in windows from Train and Test split
⎵ – Windows
⎵ – Window between which splitting happens (18 & 19)
▨ – Overlapping parts where data is seen in windows from Train and Test split
X – Windows removed from Train split to avoid information leaking

Figure 4.9: Illustration of Overlapping Window Removal.

The models are then trained and evaluated for each of those five different splits and the average of the evaluation metrics is calculated as final evaluation metric.

4.3.6 Code Implementation

During the process of this master project, different code has been developed to perform a variety of experiments. All code is well documented with the aim of easy re-usability. Docstrings indicate the functions' responsibility and describe inputs and outputs. Comments within the code guide future developers, to easily understand what specific parts of the code aim to accomplish and what parameters can be adjusted. The repository is enhanced with ReadMe files for each experiment conducted, which provide a general overview and describe the steps needed in order to reproduce the experiment. Further, the ReadMe files contain references to Boxnotes with additional information on where the output of the experiment can be found for further analysis without the need to re-execute the experiment. The exact structure of the repository and useful installation guidelines can be found in Appendix A.1.

Chapter 5

Experiments, Results and Discussion

The following chapter contains the most important experiments conducted in this master project and their corresponding results. The listed experiments mostly follow a semantic structure and build up until the last experiment, where different generalizability aspects of the implemented models are evaluated. Since this master project contains numerous experiments, the results are directly discussed of each experiment to facilitate the reading flow. In the end, a comparative discussion of all experiments is provided.

5.1 Trace Validation

To verify, that the execution of a ransomware sample and the corresponding trace collection had been executed successfully, different analyses need to be conducted. This section exemplary describes the evaluation of the impact caused on the system during the ransomware execution on the example of Black Basta by first comparing the reference entropy of the system (before) with the entropy of the infected partition (after). Secondly, the values and additional computed metrics of the newly collected traces are plotted and evaluated in a profound Time Series Analysis. These steps ensure that the collected data is valid and can be used for further machine learning experiments.

5.1.1 Entropy Analysis

As described in Section 4.1, entropy is an important measurement for describing the degree of disorder or randomness in the system. Therefore, a successful run of a particular ransomware sample shows an increase of entropy over the extracted partition after the infection compared to the reference entropy values. Figure 5.1 depicts the side-by-side comparison of the entropy analysis for Setup 1, where the OS and data are located on separate partitions and Figure 5.2 shows Setup 2, where the OS and data are located on the same partition (see Section 4.3.1). The entropy analysis for each setup consists of a histogram plot and a delta plot.

The histogram plot with 50 bins shows the range of possible entropy values from zero to eight on the x-axis and the corresponding frequency of each value on the y-axis. The blue vertical bars indicate the frequency of a value before the infection, whereas brown bars indicate the frequency after the infection. Hereby, changes and shifts in the entropy frequency can be detected and interpreted. For instance, the data partition histogram plot for Setup 1 shows a clear increase in the frequency at entropy value eight. Further, the frequency within the range of two to five decreases, whereas the frequency within the range of five and half to seven increases. This also indicates a clear shift towards higher entropy values. The same is true for Setup 2 where a clear increase in frequency around the entropy value seven is observed. A decrease of the frequency at entropy value zero is recognizable. This is also true for the histogram plot of the OS partition for Setup 1 where the frequency decreases from 56 % to 5 %. A possible explanation for this observation is, that the executed workload is writing new (possibly encrypted) files to LBA sectors, where before nothing was stored. This behavior is called *writing out of place*. The decrease from 15 % to 10 % in entropy value eight in the histogram plot of the OS partition for Setup 1, is assumed to be caused by writing small files out of place to a location where the previous information was not zeros and the block cannot be populated fully. These observations all confirm the correct execution of the ransomware sample and proves that it did affect the data stored on the storage device. Additionally, the percentage of the modified LBAs is calculated and displayed below each histogram plot. In the case of Black Basta using Setup 1, the data partition has been modified by over 31 %.

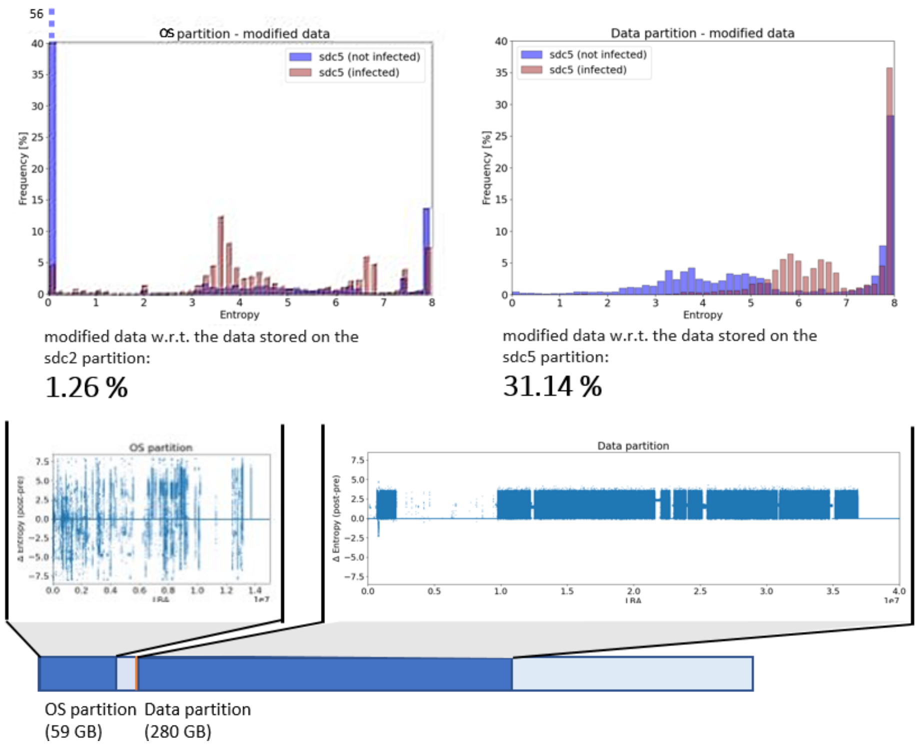


Figure 5.1: Example entropy analysis for the Black Basta ransomware using Setup 1.

The entropy delta plot shows the LBA range for each partition on the x-axis and the delta entropy (post-pre) on the y-axis. The horizontal blue line marks the delta entropy. A

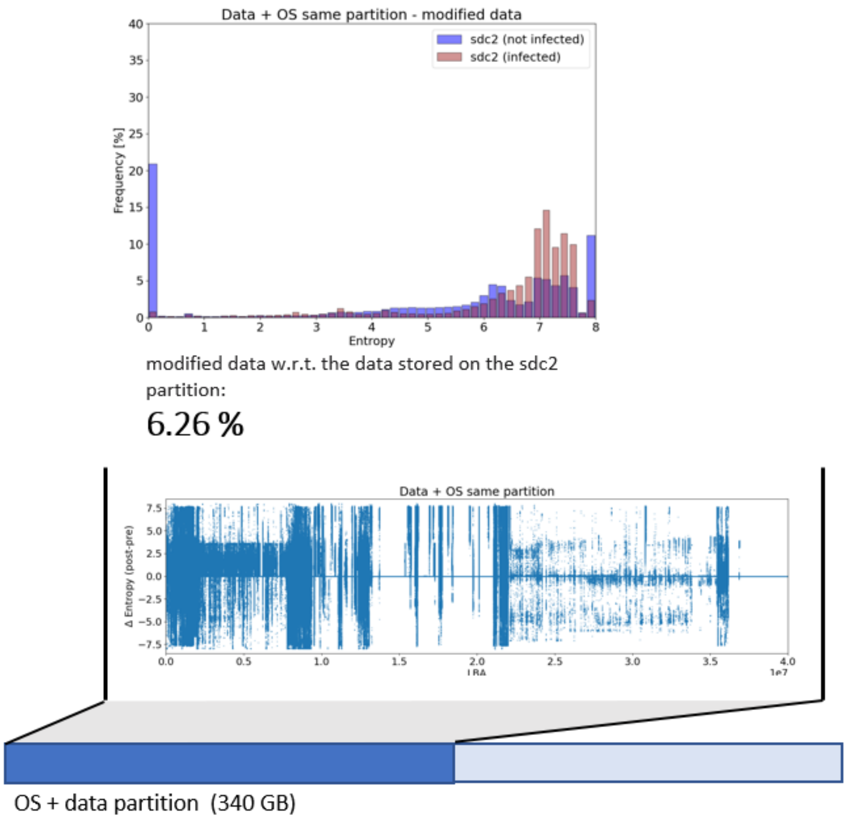


Figure 5.2: Example entropy analysis for the Black Basta ransomware using Setup 2.

zero delta entropy value means that the entropy in that particular LBA did not change. Every point above this blue line, indicates that the entropy for this LBA did increase. A point below this line means a negative delta entropy. The delta plot for the data partition in Setup 1 shows a clear increase in entropy with a pronounced upper limit at a value around four. The OS partition in Setup 1 looks more sparse and shows an increase, as well as an decrease over the whole LBA range. Interestingly, the delta plot of Setup 2 encodes both patterns from the delta plots in Setup 1.

The example of Black Basta shows, that it is possible to indicate whether the ransomware impacted the system and caused any changes during its execution or not by looking at the plots from the entropy analysis.

5.1.2 Time Series Analysis on IO Operations

After the entropy analysis showed that the executed ransomware attack was successful and the device was impacted, the time series analysis on IO operations is used to confirm that the duration of the IO operations is correct and that the traces are not corrupted. Further it provides valuable insights into the ransomware specific behavior and its active parts. Figure 5.3 depicts three plots, all of them having time in seconds on the x-axis and different metrics on the y-axis. Starting at the top, the entropy plot shows the minimum entropy value of a IO within a second in orange, the average entropy value in green and the maximum entropy value in blue.

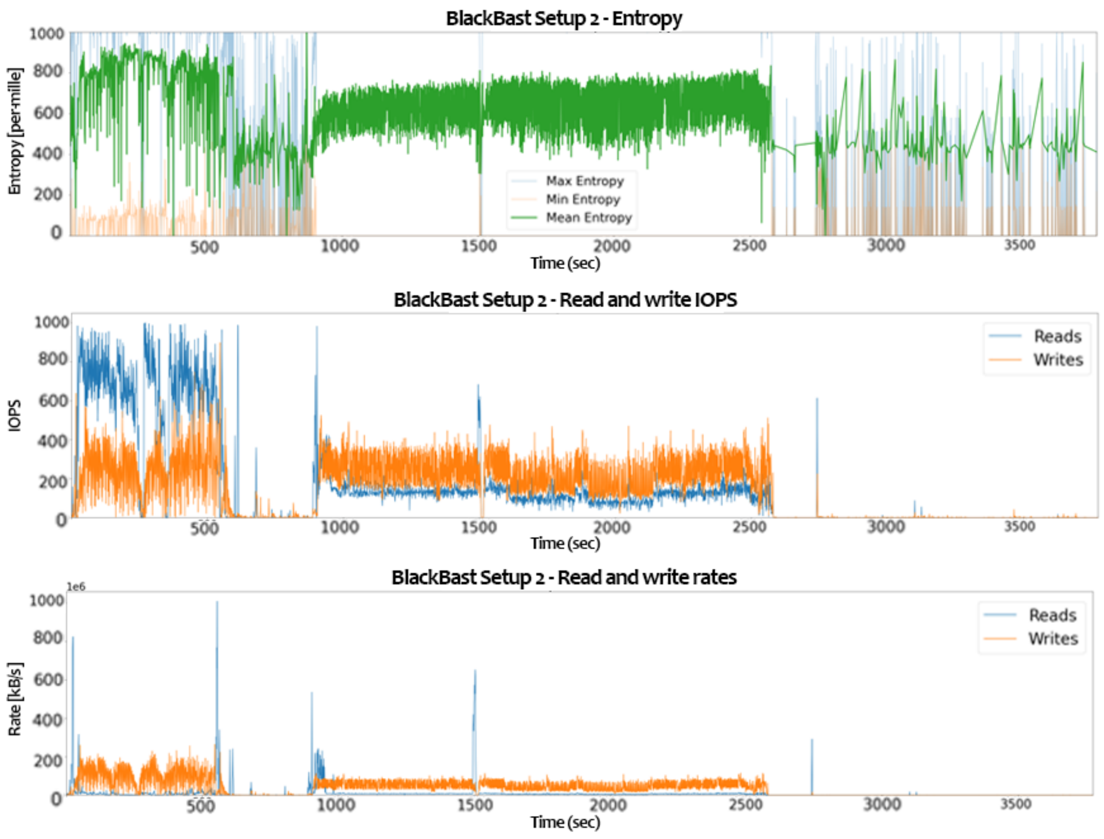


Figure 5.3: IO Time Series Analysis for Black Basta Setup 2.

The middle plot contains the number of Input/Output Operations Per Second (IOPS) for reads in blue and for writes in orange. The plot in the bottom contains the read rate in kilobit per second (e.g., 1.28×10^6 kbit/s) in blue and the write rate in orange. From the middle plot, a clear decrease of number of IOPS can be detected after 500 seconds. The same pattern is also recognizable in the top plot, where the mean entropy (green line) is decreasing and the bottom plot, where both read rate and write rate are close to zero most of the time. After approximately 900 seconds the contrary can be observed, which indicates that the ransomware increasingly encrypts files again. After 2500 seconds the values of the IOPS as well as read/write drop to zero, which indicates that the ransomware has finished its actions. Further, the green line in the entropy plot starts showing a linear behavior where the slope changes suddenly in certain points, whereas during the second active period (900 seconds to 2500 seconds) the change was rather smooth. The reason for this observation lies in the small number of IO's after the ransomware attack is terminated.

Figure 5.4 shows a side-by-side comparison of the *LBA scatter plot* for the benign workload (file conversions) and the WannaCry ransomware. The *LBA scatter plot* is used to detect access patterns of the executed workload and to evaluate what sectors have been modified. The x-axis reflects the duration of the collected traces from 0 to 60 minutes and the y-axis reflects the stacked LBA sectors, reassembling the entire disk. It is important to indicate, that the y-axis is intentionally squeezed unproportionally to allow a more

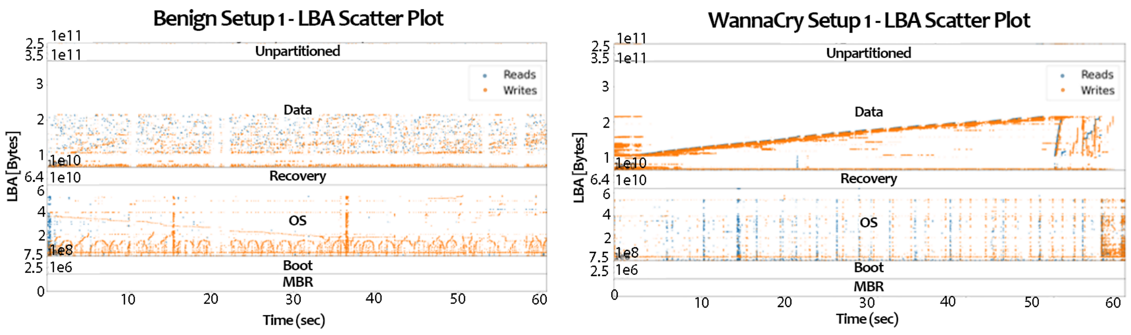


Figure 5.4: LBA Scatter Plot, Comparison Of Benign Setup 1 To WannaCry Setup 1.

intuitive visualization where the entire disk is shown. Hence, distances on the y-axis within the plot cannot be compared. Also, the coloring of read and writes does not exactly reflect the number of read and write operations as the data is plotted on top of each other (with some transparency) but gives a good indication of what type of IO operations are ongoing. The side-by-side comparison, simplifies the analysis and therefore, differences in the data sector can be easily identified. On the left graph, a random pattern can be detected, which indicates that the IO operations over a short period of time cover a broad range in the LBA space (high variance). Differently, the plot on the right shows a continuous increasing line over time, which indicates a sequential access pattern behavior.

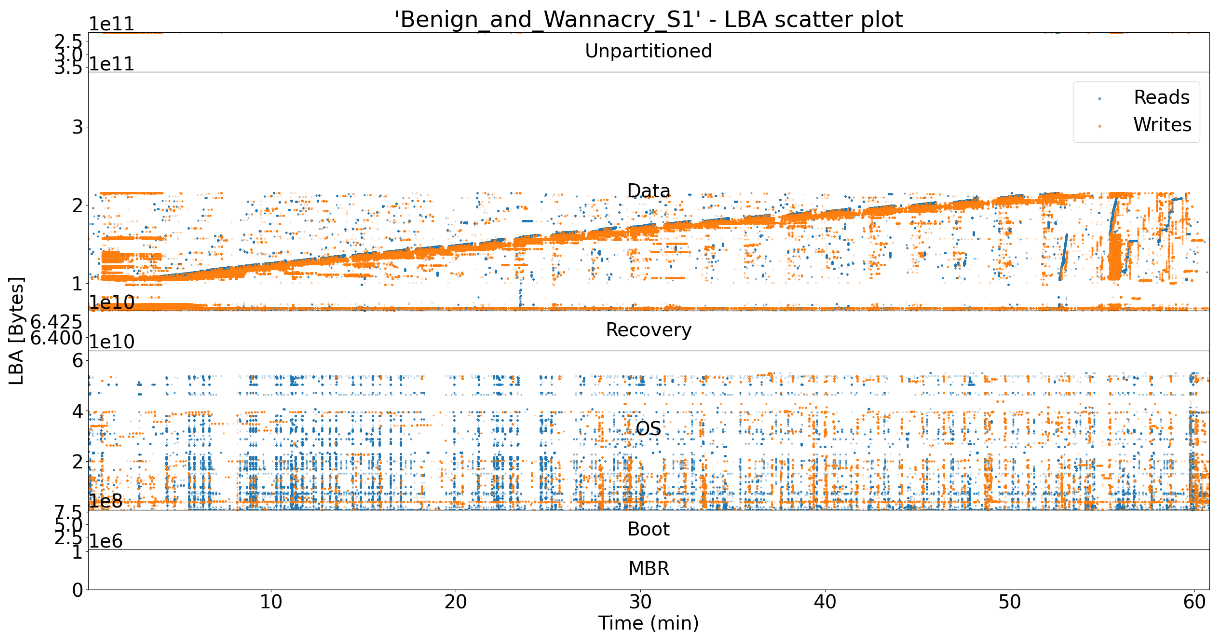


Figure 5.5: LBA scatter plot, Benign + WannaCry Setup 1.

Figure 5.5 shows the *LBA scatter plot* of the mixed workload trace where benign workload and WannaCry ransomware was executed simultaneously. The plot shows the expected result, since both patterns previously described from the individual workload plots (see Figure 5.4) can be recognized.

5.1.3 Active Parts

Based on the time series analysis on IO operations, the active parts (starting from the beginning) of each collected trace was selected. Table 5.1 gives an overview for all ransomware families and how long their active parts take, starting from the beginning. To balance the data for Setup 1 (S1) and Setup 2 (S2) for each ransomware, the smaller active part those have been taken for both.

Table 5.1: Active parts of different ransomware.

Ransomware	Duration of Active Part
Sodinokibi (S1 and S2)	300 Seconds
Lockfile (S1 and S2)	300 Seconds
BlackBasta (S1 and S2)	500 Seconds
WannaCry (S1 and S2)	1500 Seconds
Lockbit (S1 and S2)	90 Seconds
Conti (S1 and S2)	1000 Seconds

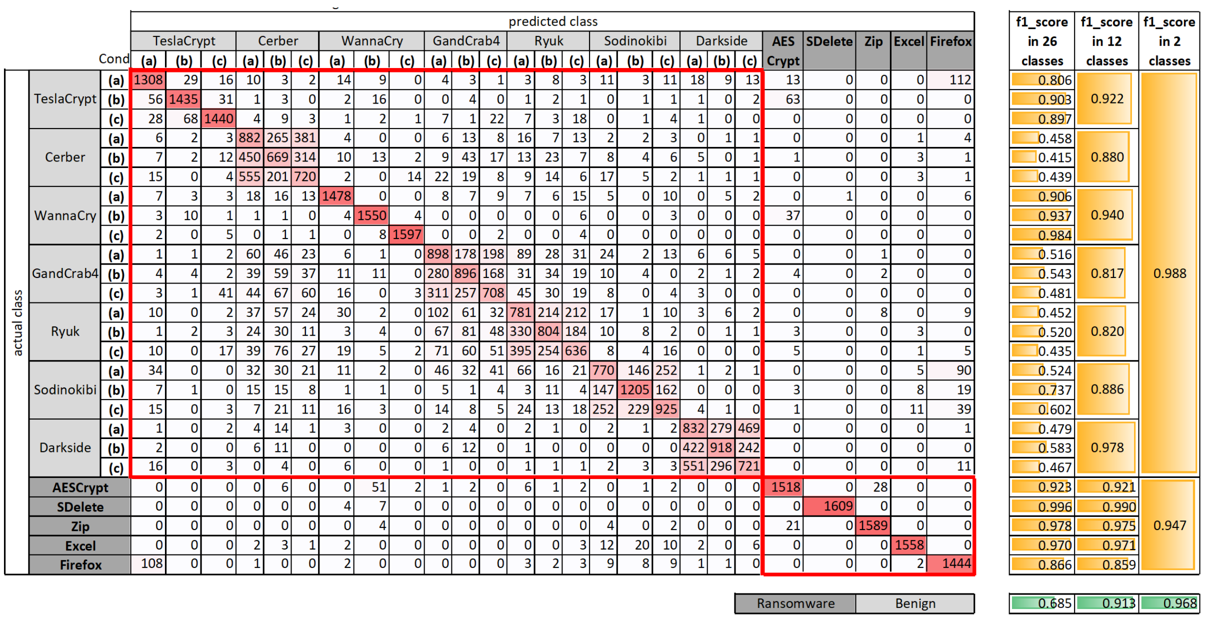
5.2 Feature Analysis

This section describes the evaluation of the newly proposed features depicted in Table 4.4 by (1) evaluating their performance compared to the original features, (2) performing a feature correlation analysis, (3) evaluating the effect of the stacking feature on the model's classification performance, (4) executing a feature importance analysis, and (5) analyzing the theoretical overhead that is introduced by each feature.

5.2.1 Performance Evaluation of New Features A to H

To evaluate the classification performance change caused by the additional features A to H depicted in Table 4.4, feature vectors including the additional features A to H were extracted using the *Feature Extractor* documented in Subsection 4.3.5 from the original dataset proposed by Hirano et al. [11]. The window size of 10 seconds and window offset of one second were kept the same as in the analysis of Hirano et al. for a valid comparison. Further, feature I (Stacking) is not included in this analysis for fair comparison by looking at the same time frame and will be evaluated in Subsection 5.2.3. With the newly computed feature vector, the Random Forest model proposed by Hirano et al. was trained and evaluated following the procedure explained in 4.1. The confusion matrix 5.6 depicts the results of the evaluation. In Table 5.2 the F1-Score from Hirano's et al. evaluation and this master projects reproduction are compared to the F1-Scores from the evaluation including the new features A to H.

In all three cases, the F1-Score of the model evaluation is increased when the features A to H are included. In 26 classes, there is the biggest increase of the F1-Score by 10.3 % to reproduction and by 10.5 % to the original. In the 12 classes case, the F1-Score increased



		predicted class																															
		TeslaCrypt			Cerber			WannaCry			GandCrab4			Ryuk			Sodinokibi			Darkside			AES Crypt	SDelete	Zip	Excel	Firefox	f1_score in 26 classes	f1_score in 12 classes	f1_score in 2 classes
actual class	Cond	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)								
TeslaCrypt	(a)	1308	29	16	10	3	2	14	9	0	4	3	1	3	8	3	11	3	11	18	9	13	13	0	0	0	112	0.806	0.922	0.988
	(b)	56	1435	31	1	3	0	2	16	0	0	4	0	1	2	1	0	1	1	1	0	2	63	0	0	0	0	0.903		
	(c)	28	68	1440	4	9	3	1	2	1	7	1	22	7	3	18	0	1	4	1	0	0	0	0	0	0	0	0.897		
Cerber	(a)	6	2	3	882	265	381	4	0	0	6	13	8	16	7	13	2	2	3	0	1	1	0	0	0	1	4	0.458	0.880	
	(b)	7	2	12	450	669	314	10	13	2	9	43	17	13	23	7	8	4	6	5	0	1	1	0	0	3	1	0.415		
	(c)	15	0	4	555	201	720	2	0	14	22	19	8	9	14	6	17	5	2	1	1	1	0	0	0	3	1	0.439		
WannaCry	(a)	7	3	3	18	16	13	1478	0	0	8	7	9	7	6	15	5	0	10	0	5	2	0	1	0	0	6	0.906	0.940	
	(b)	3	10	1	1	1	0	4	1550	4	0	0	0	0	0	6	0	0	3	0	0	0	37	0	0	0	0	0.937		
	(c)	2	0	5	0	1	1	0	8	1597	0	0	2	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0.984		
GandCrab4	(a)	1	1	2	60	46	23	6	1	0	898	178	198	89	28	31	24	2	13	6	6	5	0	0	1	0	0	0.516	0.817	
	(b)	4	4	2	39	59	37	11	11	0	280	896	168	31	34	19	10	4	0	2	1	2	4	0	2	0	0	0.543		
	(c)	3	1	41	44	67	60	16	0	3	311	257	708	45	30	19	8	0	4	3	0	0	0	0	0	0	0	0.481		
Ryuk	(a)	10	0	2	37	57	24	30	2	0	102	61	32	781	214	212	17	1	10	3	6	2	0	0	8	0	9	0.452	0.820	
	(b)	1	2	3	24	30	11	3	4	0	67	81	48	330	804	184	10	8	2	0	1	1	3	0	0	3	0	0.520		
	(c)	10	0	17	39	76	27	19	5	2	71	60	51	395	254	636	8	4	16	0	0	0	5	0	0	1	5	0.435		
Sodinokibi	(a)	34	0	0	32	30	21	11	2	0	46	32	41	66	16	21	770	146	252	1	2	1	0	0	0	5	90	0.524	0.886	
	(b)	7	1	0	15	15	8	1	1	0	5	1	4	3	11	4	147	1205	162	0	0	0	3	0	0	8	19	0.737		
	(c)	15	0	3	7	21	11	16	3	0	14	8	5	24	13	18	252	229	925	4	1	0	1	0	0	11	39	0.602		
Darkside	(a)	1	0	2	4	14	1	3	0	0	2	4	0	2	1	0	2	1	2	832	279	469	0	0	0	0	1	0.479	0.978	
	(b)	2	0	0	6	11	0	0	0	0	6	12	0	1	0	0	0	0	0	422	918	242	0	0	0	0	0	0.583		
	(c)	16	0	3	0	4	0	6	0	0	1	0	0	1	1	1	2	3	3	551	296	721	0	0	0	0	11	0.467		
AESCrypt		0	0	0	0	6	0	0	51	2	1	2	0	6	1	2	0	1	2	0	0	0	1518	0	28	0	0	0.923	0.921	0.947
SDelete		0	0	0	0	0	0	4	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1609	0	0	0	0.996	0.990	
Zip		0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	4	0	2	0	0	0	21	0	1589	0	0	0.978	0.975	
Excel		0	0	0	2	3	1	2	0	0	0	0	0	0	0	3	12	20	10	2	0	6	0	0	0	1558	0	0.970	0.971	
Firefox		108	0	0	1	0	0	2	0	0	0	0	0	3	2	3	9	8	9	1	1	0	0	0	0	2	1444	0.866	0.859	
																							Ransomware		Benign			0.685	0.913	0.968

Figure 5.6: Confusion matrix of seven ransomware and five benign software on Windows 7 using Hiranos traces, additionally including newly proposed features A to H

Table 5.2: F1-Score Comparison of newly proposed features A to H.

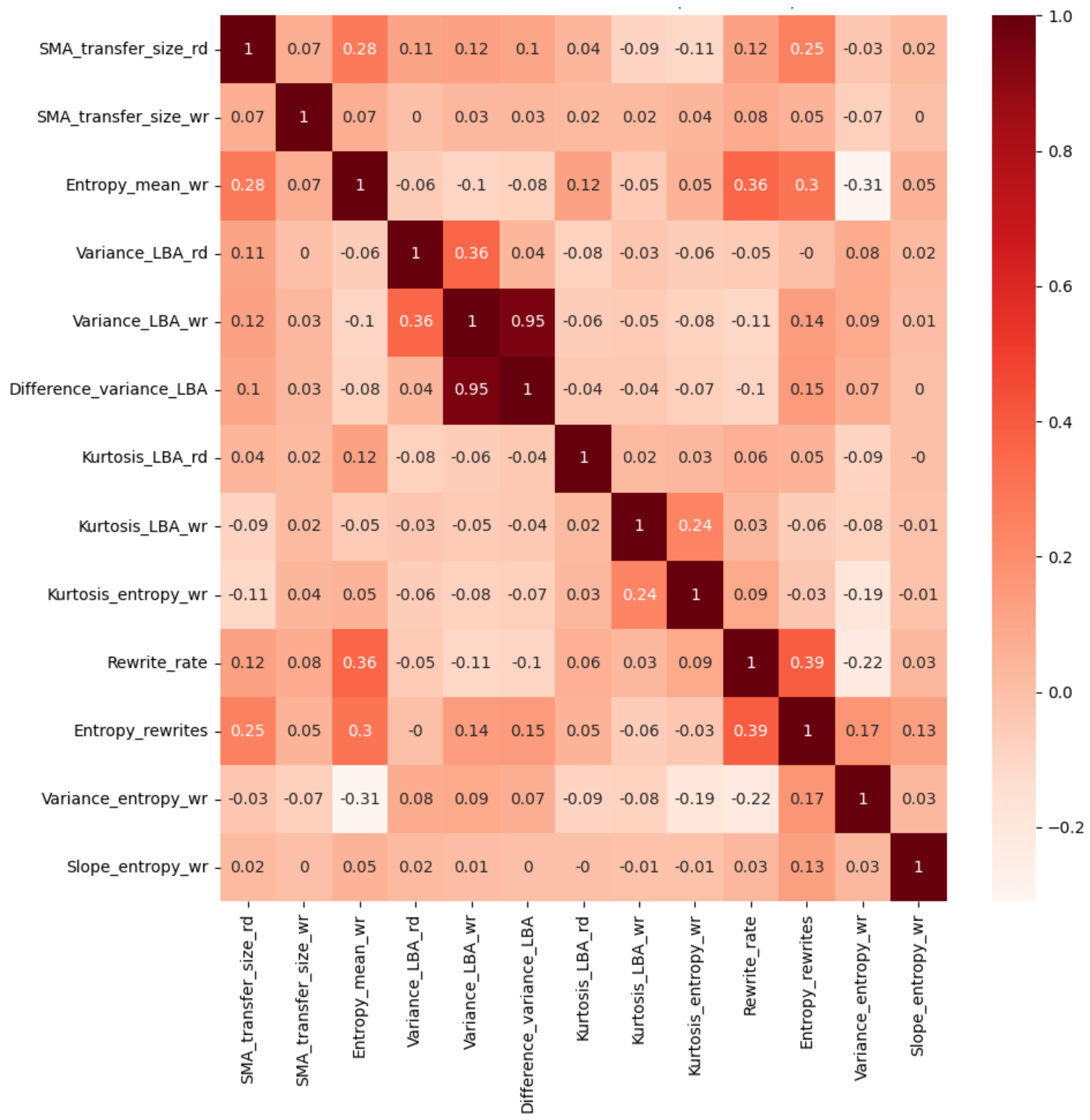
Number of Classes	F1-Score Hirano et al.	F1-Score Reproduction	F1-Score including Features A - H
26	58.0 %	58.2 %	68.5 % (+10.5 % / +10.3 %)
12	85.8 %	83.1 %	91.3 % (+5.5 % / +8.2 %)
2	96.2 %	95.2 %	96.8 % (+0.6 % / +1.6 %)

by 8.2 % to reproduction and by 5.5 % to the original. The 2 classes case shows the smallest increase by 1.6 % to the reproduction and by 0.6 % to the original. The smallest increase in the binary classification case is reasoned by the already high F1-Scores in the reproduction and original dataset.

5.2.2 Feature Correlation

The next conducted experiment in the feature analysis is the Pearson correlation between the features (original and A to H) using Pandas Pearson correlation function and the feature vector extracted from Hirano et al. original dataset with window size 10 seconds and window offset 1 second. The results are shown in Figure 5.7 and show a high positive correlation between the original feature variance of LBA of writes and the newly introduced feature Difference of variance of LBA writes and variance of LBA writes (feature D) of 95 %.

Further, rerunning the performance evaluation experiment explained in Subsection 5.2.1 without feature D, showed similar performance metrics compared to running it including feature D. Therefore, the feature D is excluded from the new feature set and not contained



	SMA_transfer_size_rd	SMA_transfer_size_wr	Entropy_mean_wr	Variance_LBA_rd	Variance_LBA_wr	Difference_variance_LBA	Kurtosis_LBA_rd	Kurtosis_LBA_wr	Kurtosis_entropy_wr	Rewrite_rate	Entropy_rewrites	Variance_entropy_wr	Slope_entropy_wr
SMA_transfer_size_rd	1	0.07	0.28	0.11	0.12	0.1	0.04	-0.09	-0.11	0.12	0.25	-0.03	0.02
SMA_transfer_size_wr	0.07	1	0.07	0	0.03	0.03	0.02	0.02	0.04	0.08	0.05	-0.07	0
Entropy_mean_wr	0.28	0.07	1	-0.06	-0.1	-0.08	0.12	-0.05	0.05	0.36	0.3	-0.31	0.05
Variance_LBA_rd	0.11	0	-0.06	1	0.36	0.04	-0.08	-0.03	-0.06	-0.05	-0	0.08	0.02
Variance_LBA_wr	0.12	0.03	-0.1	0.36	1	0.95	-0.06	-0.05	-0.08	-0.11	0.14	0.09	0.01
Difference_variance_LBA	0.1	0.03	-0.08	0.04	0.95	1	-0.04	-0.04	-0.07	-0.1	0.15	0.07	0
Kurtosis_LBA_rd	0.04	0.02	0.12	-0.08	-0.06	-0.04	1	0.02	0.03	0.06	0.05	-0.09	-0
Kurtosis_LBA_wr	-0.09	0.02	-0.05	-0.03	-0.05	-0.04	0.02	1	0.24	0.03	-0.06	-0.08	-0.01
Kurtosis_entropy_wr	-0.11	0.04	0.05	-0.06	-0.08	-0.07	0.03	0.24	1	0.09	-0.03	-0.19	-0.01
Rewrite_rate	0.12	0.08	0.36	-0.05	-0.11	-0.1	0.06	0.03	0.09	1	0.39	-0.22	0.03
Entropy_rewrites	0.25	0.05	0.3	-0	0.14	0.15	0.05	-0.06	-0.03	0.39	1	0.17	0.13
Variance_entropy_wr	-0.03	-0.07	-0.31	0.08	0.09	0.07	-0.09	-0.08	-0.19	-0.22	0.17	1	0.03
Slope_entropy_wr	0.02	0	0.05	0.02	0.01	0	-0	-0.01	-0.01	0.03	0.13	0.03	1

Figure 5.7: Feature Correlation original and A to H

in all the following results. Another positive correlation of 39 % is observed between the rewrite rate and the entropy of rewrites. This correlation is obvious since the entropy of rewrites is zero when the rewrite rate is zero. When the rewrite rate is not zero, however, the entropy of rewrites is not correlated with the rewrite rate anymore. Thus, none of these two features are excluded.

The feature correlation analysis completes the first part of the feature analysis executed on Hirano et al. traces for comparison. The following experiments are using the self-collected storage access patterns from six different ransomware and one benign workload as explained in 4.3.1.

5.2.3 Analysis of Stacking Feature *I*

This experiment analyzes the impact of the feature *I* (horizontal stacking of multiple windows) following the idea of including the time axis in a feature row and looking at more granular time windows whilst still looking at the same time frame. The method of horizontal stacking multiple windows is evaluated in a separate experiment because it differentiates itself from all other features. Feature *I* is a method of horizontally stacking all the other features *original + A-C + E-H* computed over windows. Thus, the impact of including the time aspect in the feature vector using horizontal stacking is evaluated in this experiment. Therefore, different feature vectors were extracted using the *Feature Extractor* explained in Subsection 4.3.5 from the self collected storage access patterns from six different ransomware and one benign workload as explained in Subsection 4.3.1. The configurations of the stacking feature analysis experiment (no stacking, Setup (A), Setup (B), Setup (C), Setup (D)) are depicted in Table 5.3. With these configurations, the Random Forest model introduced in Subsection 4.2.3 was trained and evaluated using 5-fold cross validation, and the results are presented in Figure 5.8. For all five setups (no stacking, Setup (A), Setup (B), Setup (C), Setup (D)) the time in seconds evaluated in one feature vector row is ten seconds for a valid and fair comparison. Looking at different time intervals in different setups, could possibly distort the results, since training and evaluating with higher window sizes tend to show higher F1-Scores as shown by Hirano et al. [11].

Table 5.3: Configurations for Stacking Feature Analysis Experiment.

Parameter	No Stacking	Setup (A)	Setup (B)	Setup (C)	Setup (D)
window size	10s	5s	2s	5s	9s
window offset	1s	5s	2s	1s	1s
windows stacked	0	2	5	6	2
dataset	self-collected traces active parts (Subsection 5.1.3)				
dataset balance	binary classification				
features	original + *A-C* + *E-H*	original + *A-C* + *E-I*			
training set	80 %				
evaluation set	20 %				
evaluation method	5-fold cross validation				

In none of the tested setups (A to D), where the stacking feature was included, did the model's performance improve. Instead, the performances decreased in all setups. For example, setup (C) is looking at five second windows and is stacking two of them horizontally, thus overall looking at 10 seconds which is the same window size as in the no stacking example. Also, window offset of 1 second is the same as in the no stacking example. Nonetheless, setup C shows an 1.2 % lower F1-Score compared to the case where no stacking was applied. These results show, that looking at storage access patterns in a more granular way and stacking them horizontally does not lead to improved results, instead decreases the performance of the model. A possible reason for this could be that the increased dimensionality affects the weights of the model negatively and lowers the model's performance. Another possibility could be that the included information with using the stacking feature, namely the change of the feature values over time, does not provide useful information for the task of detecting ransomware. Since including the stacking of features did not increase the model's performance, the feature *I* horizontal stacking of multiple windows is excluded from the feature set and from the following experiments.

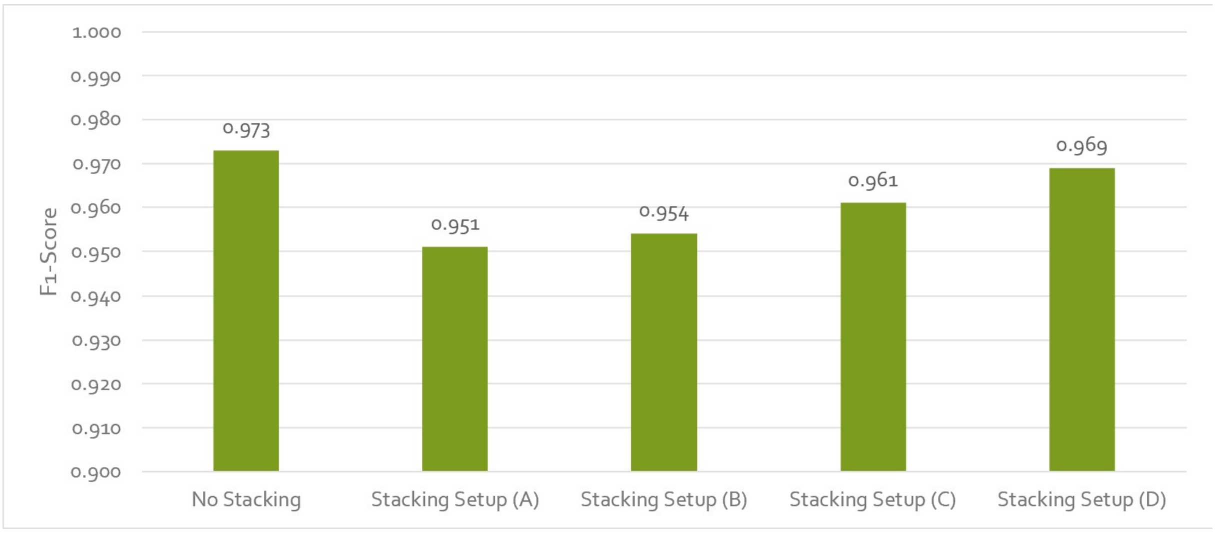


Figure 5.8: Comparing Performance No Stacking vs. Stacking

5.2.4 Feature Importance

In order to evaluate the contribution of our hand designed features towards the classification task, we run three different feature importance algorithms, namely the mean decrease in impurity of a trained random forest model [90], feature permutation [91] based and the SelectKBest[92].

Given the random forest model on the dataset of the features extracted with window size of 10 seconds and with 1 second offset for the the binary classification problem, the mean and the standard deviation of accumulation of the impurity decrease in each of tree of the complete forest was extracted. This method of calculating feature importance is not model agnostic and specific to the random forest model [90]. From Figure 5.9 it can be seen that the top four most contributing features in decreasing importance according to this method are the simple moving average on the write transfer size `SMA_transfer_size_wr`, the mean entropy of write operations `Entropy_mean_wr`, the variance of the LBAs written `Variance_LBA_wr`, and the the simple moving average on the read transfer size `SMA_transfer_size_rd`.

Another way to calculate the feature importance is by performing feature permutation. This method first does the inference of the trained model on the test dataset and subsequently picks a feature and permutes all the rows in the same feature to create a new test dataset. Based on this new test dataset it does another inference of the same trained model. From each inference, the accuracy is measured, and the difference between them is calculated [91]. The advantage with this method of calculating the feature importance is that, this method is model agnostic and not just constrained to a specific model type. The dataset used were features extracted over a window size of 10 seconds with a window

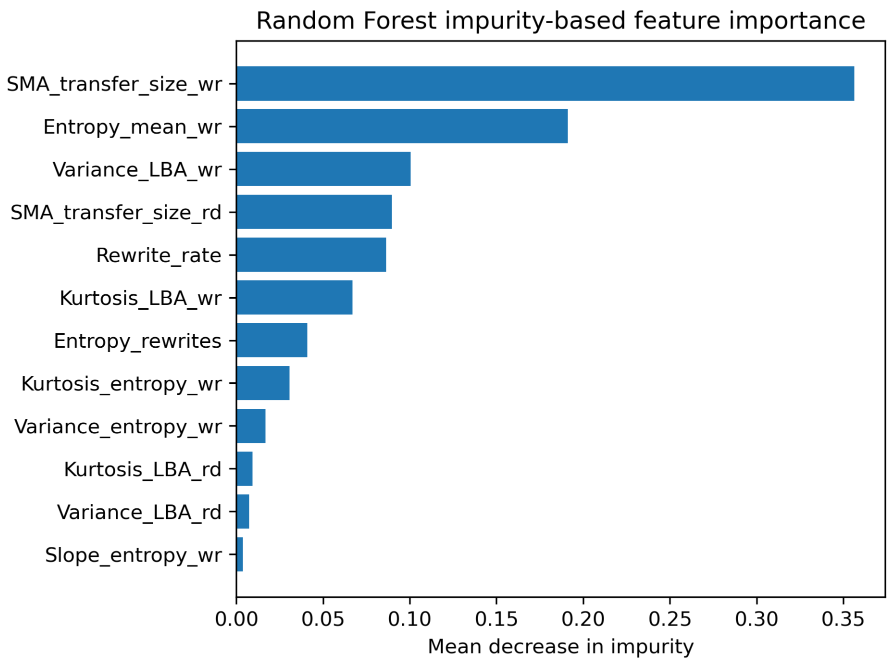


Figure 5.9: Random forest importance of features based on mean decrease in impurity.

offset of 1 second, and the model used was the same trained random forest model as in the previous case. On Figure 5.10 it can be seen that the top four most contributing features according to this method are the mean entropy of write operations `Entropy_mean_wr`, the simple moving average on the write transfer size `SMA_transfer_size_wr`, the entropy of rewrites `Entropy_rewrites` and the rewrite rate `Rewrite_rate`. The black lines indicate the variance of the feature importance between different runs with the feature selected for the permutation.

Finally, the feature importance was calculated directly from the dataset without the need of a trained model using the SelectKBest scikit-learn method [92]. Therefore, this method of calculating the feature importance is also model agnostic and not dependent on the specific type of the model being used. It uses different scoring functions like *f_classif*, *mutual_info_classif*, *chi2*, *f_regression*, and others between the features and labels. In our case, the *f_classif* scoring function was used which computes the ANOVA F-value for the individual features. In Figure 5.10 it can be seen that the top four most contributing features according to this method are the mean entropy of writes `Entropy_mean_wr`, the variance of the LBAs written `Variance_LBA_wr`, the entropy of rewrites `Entropy_rewrites` and the kurtosis of the entropy of write operations `Kurtosis_entropy_wr`.

The first feature collection method is model specific i.e., specific to the random forest model whereas the permutation importance and the SelectKBest methods are model agnostic and therefore can be seen as more meaningful.

From analysing the results of the feature importance of these three methods, the following interpretations are made.

1. The *mean entropy on write operations* and *variance of LBAs written* showed up in

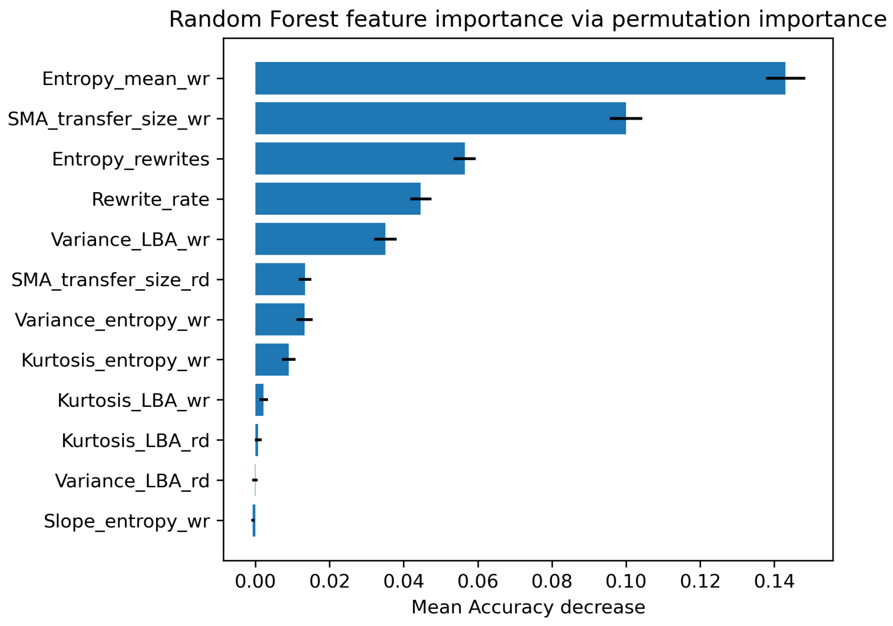


Figure 5.10: Feature permutation based feature importance.

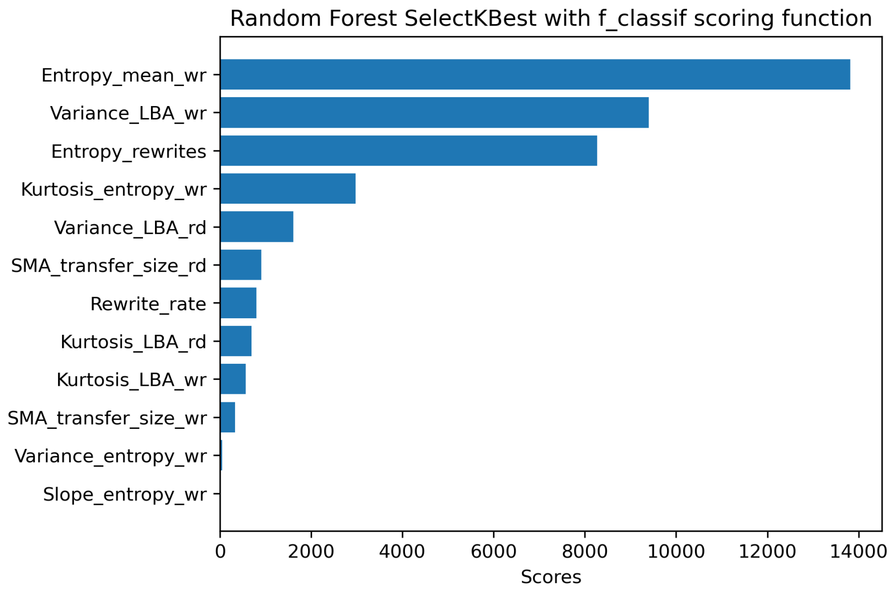


Figure 5.11: SelectKBest based feature importance.

the top 4 features in all the feature importance calculation methods.

2. The *simple moving average of the write transfer size* showed up as the top 4 feature in the first and second feature importance calculation method but not in the third.

3. The *entropy of rewrites* showed up as the top 4 feature in the second and third feature importance calculation methods but not in the first.

4. Out of our own hand designed features, the *re-write rate* and *mean entropy of re-*

writes added good enough importance to the classification task.

5. Out of our own hand designed features, the *slope of the entropy of write operations* added the least value to the classification task.

5.2.5 Theoretical Overhead Analysis

Time overhead analysis is important, since it provides information about the worst-case theoretical run-time complexity and indicates the performance of the feature extraction unit according to the resource. This analysis shows the big $O()$ time complexity of computing features out of the storage access patterns collected when tracing the workloads. Tracing storage access patterns with SystemTab and extracting the entropy also creates overhead which is only present in test environments as introduced above in Section 4.3.1. When implementing the ransomware detection machine learning pipeline into a final storage system product using computational storage devices , dedicated hardware assist will be available to significantly reduce this type of overhead. Hence, this analysis includes the overhead of computing the features only. Since the entropy of each write is already present in the traces, computing the average normalized Shannon entropy consists of summing up these values and dividing it through the number of IO operations, which corresponds to a big O time complexity of $O(n)$. Similarly, computing the features average throughput and mean entropy of rewrites is computing the average and $O(n)$. The features variance of LBA and variance of entropy both include computing the variance of a number of samples. This consists of computing the average of all samples, which is $O(n)$ and summing up the square of difference of each sample and the average, which again is $O(n)$. Therefore, the time complexity is $O(n) + O(n) = O(2n) = O(n)$. The slope of the entropy is computed by fitting a linear least squares fit. The time complexity of this is dependent on the method taken to compute the linear regression. Many different solutions exist, where efficient ones using matrix multiplication perform the computation in $O(k^2 \cdot (n+k))$ where n is the number of rows and k is the number of columns. Since the linear regression is computed over one column, namely the entropy, k is one and the linear regression computation follows complexity of $O(1^2 \cdot (n+1)) = O(n)$. The kurtosis of entropy and LBA is computed with formula (4.8) for the fourth centralized moment, which can be disassembled as $Kurt(X) = E((X-\mu)^4) = E(X^4) - 4E(X)E(X^3) + 6E(X)^2E(X^2) - 3E(X)^4$ where X is denoting the random variables entropy and LBA. Lower central movements are directly related to the mean, variance and skewness which have shown to correspond to a time complexity of $O(n)$. Therefore, the computational time complexity for the kurtosis is $O(6n) = O(n)$. Computing the rewrite rate requires to iterate over the data containing reads and for each of that reads iterate over the data containing writes to check if there are writes coming after reads to the same LBA. This procedure corresponds to a complexity of $O(n^2)$. However, there is a computationally more sophisticated method to perform this, which is implemented in this master project. An inner join between the data containing the reads and the writes is performed, which corresponds to $O(n \cdot \log(n))$. Then, one can iterate over the joined data, which is $O(n)$. In practice, this joined data mostly contains significantly less than n rows. However, the worst case time complexity is $O(n \cdot \log(n) + O(n)) = O(n \cdot \log(n))$.

Table 5.4: Big Oh Time Complexity of Features.

Feature	Complexity
Average normalized Shannon Entropy of writes	$O(n)$
Average Throughput [bytes/s]	$O(n)$
Variance of LBA	$O(n)$
Variance of Entropy	$O(n)$
Rewrite rate	$O(n \cdot \log(n))$
Mean Entropy of Rewrites	$O(n)$
Slope of Entropy	$O(n)$
Kurtosis of Entropy	$O(n)$
Kurtosis of LBA	$O(n)$

5.3 Impact of Trace Data Selection on the Model Performance

Storage access patterns from six different ransomware and one benign workload have been collected for a maximum of one hour each. The time series analysis explained in Subsection 5.1.2 has shown that not all workloads show the same activity over this hour. As explained in Subsection 5.1.3, the most active timeframe from each workload has been identified. To determine how much time in seconds should be used from each workload as the dataset for training and evaluating the machine learning models in order to achieve the best possible performance, a data selection experiment was performed. In this experiment, the machine learning models were trained and evaluated using feature vectors created from different duration of traces as the dataset as outlined in Table 5.5. The feature vectors were extracted using 1.5 min, 3 min, 12 min, and the defined active parts for each ransomware on each setup (Setup 1 and Setup 2). To balance the dataset for binary classification, the sum of the time taken for all ransomware traces has been taken for the benign workload. In the 12-minute case, however, it was not possible to perfectly balance the dataset for binary classification since there is a maximum of 60 min of benign workload traces and 6 times 12 min would require 72 min of trace. Instead, the maximum possible amount of 60 min was taken. With these feature vectors, the models were trained and evaluated using 5-fold cross validation method described in Subsection4.3.5 and the average of evaluation F1-Scores are shown in Figure 5.12.

The results show, that taking the defined active parts as dataset show the highest F1-Score of 97.3 % for Random Forest and XGBoost and 95.6 % for the DNN. Selecting 1.5 min of each ranomware trace shows an F1-Score decrease compared to the active parts by 3.2 % for random forest, by 4.4 % for XGBoost and by 4.9 % for DNN. Selecting 3 min of each ransomware trace shows an F1-Score decrease compared to the active parts by 3.2 % for random forest, by 3.1 % for XGBoost and by 2.7 % for DNN. The worst classification performance show the models that were trained and evaluated using 12 min of each ransomware trace, with an F1-Score of 92.7 % for Random Forest, 92.8 % for XGBoost and 90.4 % for the DNN. Thus, the active parts will be used as dataset for all the following experiments. Further, it can be observed that the Random Forest and XGBoost reached the highest F1-Score in all three data setups of this experiment and show

Table 5.5: Configurations for Data Analysis Experiment.

Parameter	1.5 min	3 min	12 min	Active Parts
window size	0 s			
window offset	1 s			
dataset each ransomware on each setup	1.5 min	3 min	12 min	active parts
dataset benign workload on each setup	9 min	18 min	60 min	60 min
dataset balance	binary classification			
features	original + *A*-*C* + *E*-*H*			
training set	80 %			
evaluation set	20 %			
evaluation method	5-fold cross validation			
models	Random Forest, XGBoost and DNN			

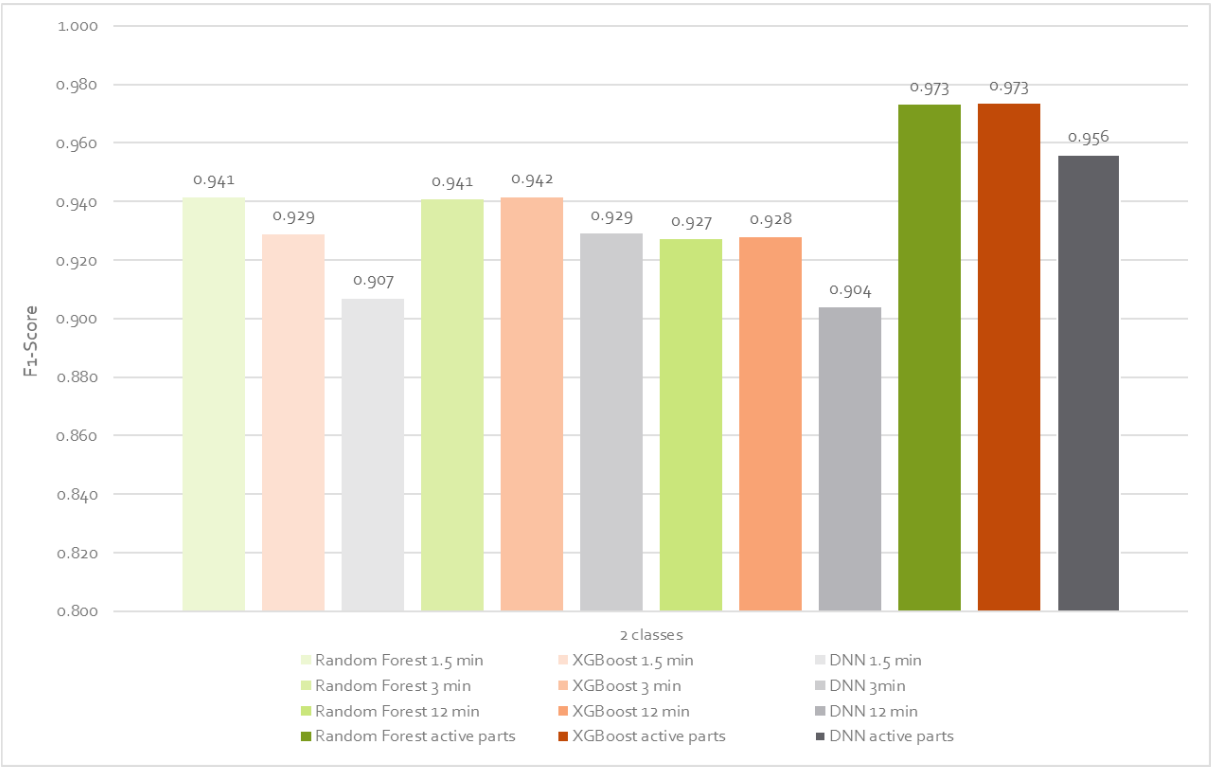


Figure 5.12: Model Performance (F1-Score) Random Forest, XGBoost and DNN on different amounts of trace data on binary classification.

a high capability of distinguishing ransomware from benign workload. The DNN shows a lower best F1-Score of 95.6 %. All three models are evaluated on their generalizability in the next section, since they might show different performance in those settings.

5.4 Generalizability

This section describes the experiments conducted in this master thesis to evaluate different aspects generalizability of the models.

5.4.1 Storage System Setups

Not every system is setup the same way. In fact, it is very likely that systems differ with regards to the used storage device as well as the partitioning of the disk. In a sliced disk, each partition is managed separately. Depending on the number of disk partitions, the content stored as well as the partition size of each partition can vary. As explained in Subsection 4.3.1, two different setups have been created. As a reminiscence, setup 1 is referred to as the setup where the OS and Data is stored on different partitions. In Setup 2 the OS and Data is stored on a single partition. This allows the evaluation of the trained machine learning models' generalizability, to detect ransomware executed on a different setup. For this experiment, the machine learning models were first trained on collected traces from Setup 1 and tested on traces from Setup 2. Second, the models were trained on traces from Setup 2 and tested on traces from Setup 1. In both cases, the models were trained with traces of six ransomware and one benign workload and the configurations shown in Table 5.6. The average F1-Score over the 5 folds is depicted in Figure 5.13. The first three bars serve as a reference, where traces from both setups were used for training as well as for testing the models. When training on setup 1 and testing on setup 2, the

Table 5.6: Configurations for Data Setup Experiment.

Parameter	Value
window size	10 s
window offset	1 s
dataset for each ransomware on each setup	active parts
dataset for benign workload on each setup	60 min
dataset balance	binary classification
features	original+ A-C + E-H
training set	6 ransomware and benign workload from one setup
evaluation set	6 ransomware and benign workload from other setup
evaluation method	5-fold cross validation
models	Random Forest, XGBoost and DNN

maximal observed performance drop over all three models consists of 2.3 % in the case of XGBoost.

Random Forest generalized the best with a F1-score of 95.1 %. In the other case, training on Setup 2 and testing on Setup 1, the maximal performance drop contains only 1.5 %

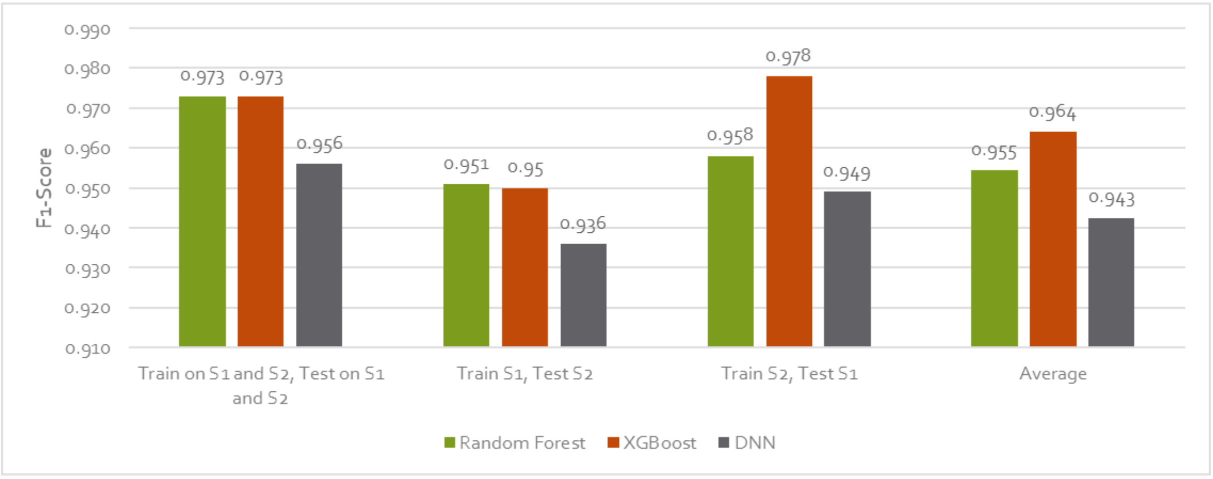


Figure 5.13: Genrealizability of Random Forest, XGBoost and DNN to different setups.

for the Random Forest model, while XGBoost even showed an improvement of 0.5 % and generalized the best. Despite the differences in the traces from Setup 1 compared to the traces from Setup 2, only a minor decrease in performance is observed which indicates an excellent generalizability (F1-Score above 93 s for all three models in both cases) to different storage system setups. Overall, XGBoost generalized best with a performance drop of 0.9 s on average (1.8 s for Random Forest, 1.35 s for DNN).

5.4.2 Mixed Workloads

In a real world scenario, ransomware is not executed in an isolated environment where only one workload is running at the same time. More likely, the user is actively interacting with the device while simultaneously the attack is taking place in the background. To test our models capability of detecting ransomware behavior in more realistic noisy environments, mixed workload traces used for testing have been collected for Lockbit (+Benign), BlackBasta (+Benign) and WannaCry (+Benign). The bash script (see Subsection 4.3.3) is executed just before the ransomware is run. As depicted in Table 5.7, all parameter of the bash script are set to 1 (random pauses, random traversing and random read only) to effectively simulate typical user behavior.

Table 5.7: Configurations For Mixed Workload Experiment.

Parameter	Value		
window size	10 s		
window offset	1 s		
features	original + *A*-*C* + *E*-*H*		
evaluation data	Lockbit (+Benign)	BlackBasta (+Benign)	WannaCry (+Benign)
bash script	-p 1, -t 1, -r 1		
evaluation	5-fold cross validation		
models	Random Forest, XGBoost, DNN: trained on active parts, binary		

The average F1-Scores over 5 folds are depicted in Figure 5.14. The results show, that both Random Forest and XGBoost generalized well for all three mixed workloads, with an average F1-Score above 91 %. The DNN model clearly shows a lower capability in distinguishing ransomware specific behavior from noise created by the file converter bash script with an average F1-Score of 78 %. Interestingly, the DNN relatively performs better on the mixed workload traces of WannaCry, F1-Score of 84.9 %, compared to the mixed workload traces from LockBit, F1-Score of 71.8 %, whereas in the case of Random Forest and XGBoost Lockbit was better detected.

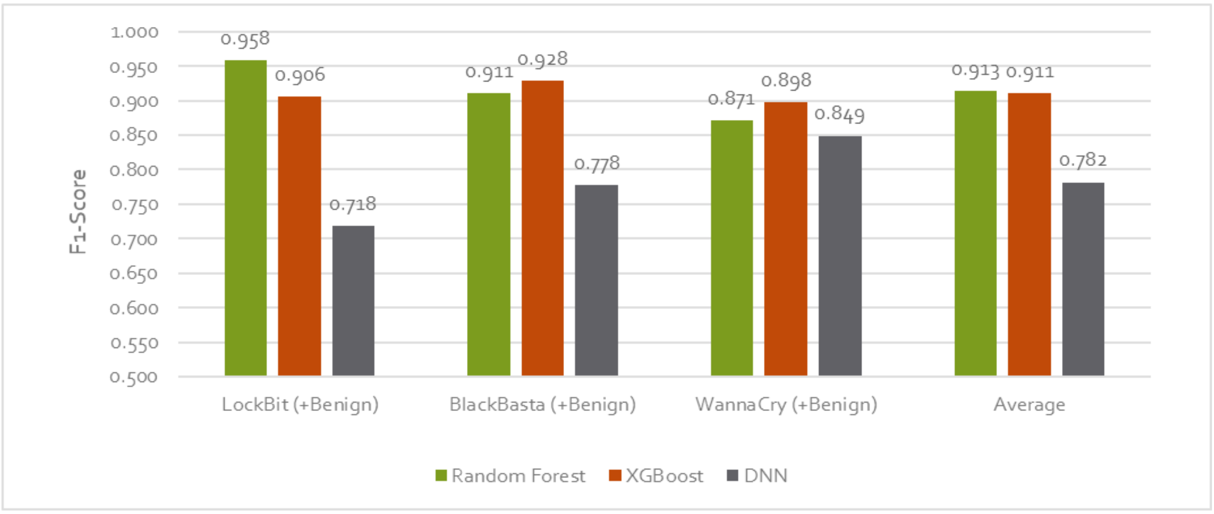


Figure 5.14: Generalizability of Random Forest, XGBoost and DNN to mixed workloads.

5.4.3 Unseen Ransomware

As discussed in Section 3 that evaluates the generalizability of models to distinguish ransomware from benign workloads when the a particular ransomware has not been seen during training is important since new ransomware types are created frequently and one would not be able to update and re-train the models whenever a new ransomware is found. To evaluate this generalizability, the models were trained on five ransomwares only and one benign workload and then evaluated on the sixth ransomware not included in the training using the configurations shown in Table 5.8. The average F1-Score over the 5 folds is depicted in Figure 5.15.

The results show that Lockbit, Conti and WannaCry are very well detected, even when the models were not trained on their storage access patterns with an F1-Score higher than 98 % for all of these three ransomware. The recognition of BlackBasta as ransomware is a lower compared to Lockbit, Conti and WannaCry but still very high with an F1-Score of 9 % on average of the models. The models were able to detect Sodinokibi with an average F1-Score of 90.5 %. For LockFile however, the F1-Score drops to 70.5 % on average. The reason for this lower F1-Score in the case of LockBit is expected to be the different behavioural patterns LockBit exhibits compared to all the other ransomware included in this master project. As shown in Subsection 4.2.1, LockFile uses the sophisticated method

Table 5.8: Configurations for Unseen Ransomware Experiment.

Parameter	Value
window size	10 s
window offset	1 s
dataset for each ransomware on each setup	active parts
dataset for benign workload on each setup	60 min
dataset balance	binary classification
features	original+ A-C + E-H
training set	5 ransomware and benign workload
evaluation set	the sixth ransomware is excluded from the training set
evaluation method	5-fold cross validation
models	Random Forest, XGBoost and DNN

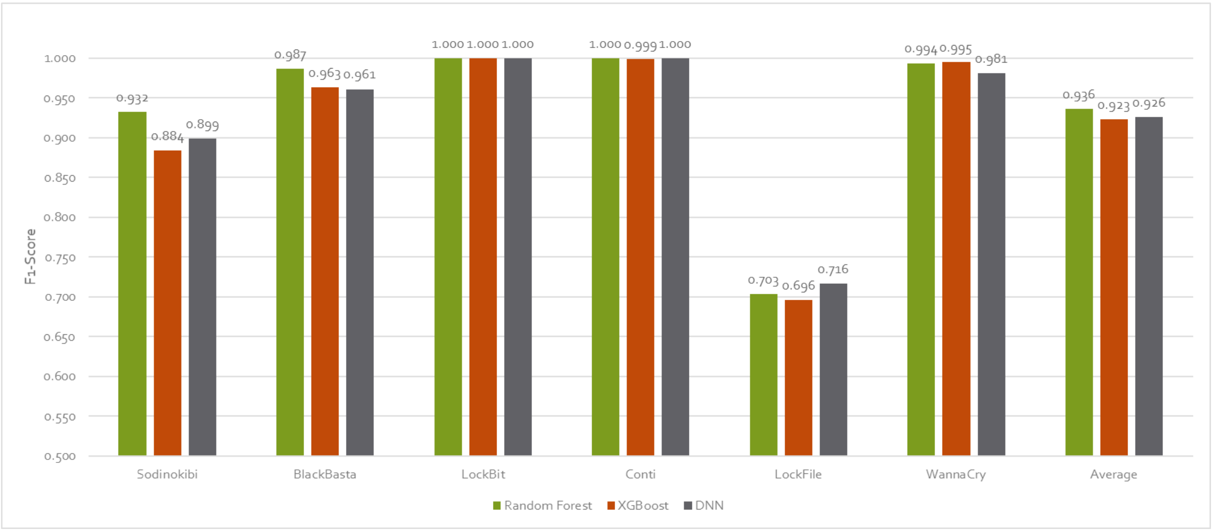


Figure 5.15: Model Performance (F1-Score) Random Forest, XGBoost and DNN to unseen ransomware.

of intermittent encryption which produces less increase in the entropy than encrypting the entire file. The entropy has shown to be an important feature for the models in the feature importance analysis in Subsection 5.2.4. This experiment shows that the generalizability to unseen, during training, ransomware of all three models is good on average with F1-Scores above 92.8 %. However, it also shows that the current models trained on the current features are able to generalize reliably for many but not every ransomware. This also implies the importance of training the final model that goes into products on a great variety of existing ransomware samples that exhibits different behavioral patterns.

5.5 Comparative Discussion of all Experiments and Results

The newly introduced features A - H improve the F1-Score of the Random Forest model in multi-classification as well as in binary classification compared to the original features proposed by Hirano et al. [11]. This shows, that the newly introduced features improve the model's capability to distinguish ransomware from the benign workload. However, not all of the newly proposed features are needed for this performance improvement. The feature correlation analysis has shown that feature D, which denotes the difference of the variance of LBA from writes and the variance of LBA from reads, has a high positive Pearson correlation (0.95) with the feature variance of LBA of writes. Removing this feature did not decrease the model's F1-Score. Further, in the second iteration of the feature engineering process proposed Feature I (horizontal stacking of windows) did not improve the models performance as either as is shown in Subsection 5.2.3 and is therefore not included in the final feature set. Thus, the final feature set proposed in this master project contains the features depicted in Table 5.9.

Table 5.9: Final feature set.

Source	Feature	reads/writes
Hirano et al. [11]	Average entropy	writes
Hirano et al. [11]	SMA transfer size [byte/s]	writes
Hirano et al. [11]	SMA transfer size [byte/s]	reads
Hirano et al. [11]	Variance of LBA	writes
Hirano et al. [11]	Variance of LBA	reads
master project	Variance of entropy	writes
master project	Rewrite rate	both
master project	Mean entropy of rewrites	both
master project	Slope of entropy	writes
master project	Kurtosis of entropy	writes
master project	Kurtosis of LBA	reads
master project	Kurtosis of LBA	writes

Besides the feature Rewrite Rate, with has a big Oh time complexity $O(n * log(n))$, all features proposed in this master project and proposed by Hirano et al. [10] have a time complexity of $O(n)$. Thus, the overhead introduced by these features is linear and manageable. If the final model is implemented in a computational storage device or in firmware of a storage devices - with an aim to produce the smallest possible overhead while still having high accuracy in terms of F1-Score - one could sacrifice the features slope of entropy, kurtosis of entropy, and kurtosis of LBA, since they showed to be the least important features for the models as analyzed in Subsection 5.2.4. Omitting the rewrite rate feature would spare the most computing time, however, it would come with a cost of a notable reduction of the model's accuracy, since it showed to be an important feature as discussed in Subsection 5.2.4. Time complexity and accuracy are a trade-offs and the best solution depends on the needs.

Training and evaluating the models on the active parts of each ransomware and the sum of the active parts from benign workload shows the best results as proven in the experiment in Section 5.3. However, it is unclear how these models behave if they are tested on the entire trace time. It is expected that the F1-Score would drop, because the entire trace time of a ransomware sample is labeled as such, regardless of it containing also parts where the ransomware runs idle, and hence the trace looks more like a benign type of workload. However, in a final product, the aim is to detect ransomware at the beginning, when it is the most active, to appropriately take measures for optimal damage control. Therefore, training the model on the active parts seems to be a reasonable approach. Another benefit of training the models only on the active parts of ransomware traces might include a decrease in the false positive rate, where a workload is declared as ransomware while it is in fact only benign software since in the less active parts ransomware resemble benign workload more. This, however, is an assumption and more experiments must be conducted in the future to prove it.

As a novelty in ransomware detection on storage systems, the models introduced in this master project have been evaluated on their generalizability to different storage system setups and to unseen ransomware. Random Forest and XGBoost have shown good generalizability to mixed workloads, where ransomware and benign workload is executed at the same time. On average, both stayed above an F1-Score of 91 %. For implementing these models in a final product, this is an important capability, since, in real-world environments, most of the time users interact with the computer when a ransomware attack starts. However, the models have only been evaluated on the combination of ransomware and one benign workload, namely a file converter. To evaluate the effect of mixed workloads including other benign workloads, further experiments have to be conducted. All three models have shown to be generalizable with regard to the two different storage system setups tested in this master project (data and OS on separate and on the same partition) with a high F1-Score of minimum 93.6 % in all cases. On average, all models also show great generalizability to (during training) unseen ransomware with an average F1-Score above 92.3 %. This is an important capability since new ransomware gets released very frequently and one would not be able to train the model on every ransomware at any given point in time. However, for one ransomware tested, namely LockFile, all three models recognized it as being ransomware with an F1-Score of only 70.5 % on average. This is significantly lower than the models' performance for other ransomware it has not seen during training. The reason is expected to be the different behavioral patterns that LockFile exhibits through performing intermittent encryption and in-memory encryption. This shows that all three models are well generalizable to unseen ransomware, which shows similar behavioral patterns but not to ransomware that exhibits remarkably different behavior. This also shows the necessity of training a ransomware detection model on a great variety of ransomware demonstrating different behavior.

The three different models (Random Forest, XGBoost, and DNN) evaluated on their performance in detecting ransomware have shown different performance metrics. While Random Forest and XGBoost both reached an F1-Score of 97.3 %, the DNN showed a lower F1-Score of 95.6 %. Also in the generalizability experiments, Random Forest and XGBoost performed better than the DNN. However, no additional parameter tuning has been performed in this master project. This needs to be evaluated in future work, since it might further improve the performance of the models and might change the difference in performance between those models.

Chapter 6

Summary and Conclusion

The effective detection of ransomware attacks is more important than ever, and is gaining increasingly high attention. This is true especially, since in 2021 ransomware counts to the top attack type amongst malware. The change in number of attacks and money paid for ransom indicate, that ransomware attacks will continue being a high thread in the future. Previous research has shown, that ransomware can be detected by various static and dynamic approaches and can be conducted on different levels of computer systems with different advantages and disadvantages. This master project pursued a machine learning based detection of ransomware, using behavioral patterns from storage systems only. Advantages of this approach are, that in a final product implementation, efficient data extraction can be performed in parallel using a large number of computational storage devices. Further, the feature extraction and inference part can be executed directly in the storage system stack, without a significant impact on the host IO traffic.

Early work on ransomware detection in storage systems with machine learning has shown promising results in self-contained environments with simple well understood workloads using a simple feature set. However, very little was known about the generalizability of the developed Random Forest model and the performance of other, more sophisticated, machine learning models and an evolved feature set has not been researched yet.

Therefore, in this master project an extended feature set for machine learning has been proposed and analyzed in many different ways, converging in a final feature set. This final feature set showed to improve the performance (F1-Score) of Random Forest model trained on the feature set proposed by Hirano et al. by 8.2 % in the 12 class multi-classification task of classifying workloads separately and by 1.6 % in the binary classification task of distinguishing benign workload from ransomware.

Furthermore, a process and environment for collecting storage access patterns of ransomware and benign workload safely has been established. Subsequently, different storage access patterns of various workloads have been traced and analyzed for their validity.

Additional to Random Forest, a XGBoost and a DNN model were used in this project. All three models showed the best performance when they were trained and evaluated on the active parts of the collected traces. Experiments proved, that ransomware can be detected reaching an F1-Score of 97.3 % for Random Forest and XGBoost in the binary classification task and in the case of DNN a F1-Score of 95.6 % was reached. As a novelty in ransomware detection in storage systems using machine learning, the models have been evaluated on different generalizability aspects. All the models showed a high generalizability to different storage system setups with an average F1-Score of 95 %. Additionally, all three models manifest a high generalizability to, during training, unseen ransomware with an average F1-Score above 92 %. This is an important capability considering that new ransomware samples are created every day. However, the analysis showed that the models performance decreases when evaluating on a ransomware that exhibits very different behavioral patterns than the ransomware the model was trained on. These insights emphasize the necessity of training the models on a variety of ransomware exhibiting different behaviors. Further, Random Forest and XGBoost showed high generalizability to mixed workloads, where ransomware and benign workload were executed at the same time with an average F1-Score above 91 %. This is an important capability since in real world scenarios, there will likely be other workloads running on the device when a ransomware attack is executed. However, only one type of benign workload, namely file conversion, has been researched. How the models generalizability to mixed workloads including other benign types of workloads behaves, remains open.

In future work, hyperparameter tuning for the models can be performed to further increase the performance. Additionally, the evaluation of the mixed workload generalizability to other types of benign workloads is an interesting step to take. Further, one could train the models on mixed workloads to further improve the models performance in a noisy environment.

Bibliography

[1] C. Srinivasan, "Hobby hackers to billion-dollar industry: the evolution of ransomware," *Computer Fraud & Security*, vol. 2017, no. 11, pp. 7–9, 2017.

[2] T. I. Times, "All about conti ransomware," https://www.irishtimes.com/news/crime-and-law/data-of-520-patients-published-online-hse-confirms-1.4578136, accessed: 13.12.2022.

[3] I. Security, "X-force threat intelligence index 2022," 2022.

[4] SonicWall, "Cyber threat report," 2021.

[5] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Twenty-third annual computer security applications conference (ACSAC 2007).* IEEE, 2007, pp. 421–430.

[6] E. Berrueta, D. Morato, E. Magaña, and M. Izal, "A survey on detection techniques for cryptographic ransomware," *IEEE Access*, vol. 7, pp. 144 925–144 944, 2019.

[7] G. Cusack, O. Michel, and E. Keller, "Machine learning-based detection of ransomware using sdn," in *Proceedings of the 2018 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, 2018, pp. 1–6.

[8] A. Kharraz, W. Robertson, D. Balzarotti, L. Bilge, and E. Kirda, "Cutting the gordian knot: A look under the hood of ransomware attacks," in *International conference on detection of intrusions and malware, and vulnerability assessment.* Springer, 2015, pp. 3–24.

[9] H. Daku, P. Zavarsky, and Y. Malik, "Behavioral-based classification and identification of ransomware variants using machine learning," in *2018 17th IEEE international conference on trust, security and privacy in computing and communications/12th IEEE international conference on big data science and engineering (TrustCom/BigDataSE).* IEEE, 2018, pp. 1560–1564.

[10] 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).* IEEE, 2019, pp. 1–6.

[11] 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. 301314, 2022.

[12] A. P. Namanya, A. Cullen, I. U. Awan, and J. P. Disso, “The world of malware: An overview,” in *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud).* IEEE, 2018, pp. 420–427.

[13] K. Krombholz, H. Hobel, M. Huber, and E. Weippl, “Advanced social engineering attacks,” *Journal of Information Security and applications*, vol. 22, pp. 113–122, 2015.

[14] T. McIntosh, A. Kayes, Y.-P. P. Chen, A. Ng, and P. Watters, “Ransomware mitigation in the modern era: A comprehensive review, research challenges, and future directions,” *ACM Computing Surveys (CSUR)*, vol. 54, no. 9, pp. 1–36, 2021.

[15] F. Noorbehbahani, F. Rasouli, and M. Saberi, “Analysis of machine learning techniques for ransomware detection,” in *2019 16th International ISC (Iranian Society of Cryptology) Conference on Information Security and Cryptology (ISCISC).* IEEE, 2019, pp. 128–133.

[16] I. Corporation, “What is data storage?” https://www.ibm.com/topics/data-storage, accessed: 04.09.2022.

[17] I. C. Education, “Block storage,” https://www.ibm.com/cloud/learn/block-storage, accessed: 11.11.2022.

[18] 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).* IEEE, 2018, pp. 293–296.

[19] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016, http://www.deeplearningbook.org.

[20] D. Venugopal and G. Hu, “Efficient signature based malware detection on mobile devices,” *Mobile Information Systems*, vol. 4, no. 1, pp. 33–49, 2008.

[21] A. Ojugo and A. Eboka, “Signature-based malware detection using approximate boyer moore string matching algorithm,” *International Journal of Mathematical Sciences and Computing*, vol. 5, no. 3, pp. 49–62, 2019.

[22] M. Zheng, M. Sun, and J. C. Lui, “Droid analytics: a signature based analytic system to collect, extract, analyze and associate android malware,” in *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications.* IEEE, 2013, pp. 163–171.

[23] Ida disassembler. [Online]. Available: https://hex-rays.com/ida-pro/

[24] D. Maiorca, F. Mercaldo, G. Giacinto, C. A. Visaggio, and F. Martinelli, “R-packdroid: Api package-based characterization and detection of mobile ransomware,” in *Proceedings of the symposium on applied computing*, 2017, pp. 1718–1723.

[25] A. Alzahrani, A. Alshehri, H. Alshahrani, R. Alharthi, H. Fu, A. Liu, and Y. Zhu, "Randroid: Structural similarity approach for detecting ransomware applications in android platform," in *2018 IEEE International Conference on Electro/Information Technology (EIT)*. IEEE, 2018, pp. 0892–0897.

[26] Apktool. [Online]. Available: https://ibotpeaches.github.io/Apktool/

[27] Droidbot. [Online]. Available: https://github.com/honeynet/droidbot

[28] Pytesser. [Online]. Available: https://github.com/RobinDavid/Pytesser

[29] A. Alzahrani, H. Alshahrani, A. Alshehri, and H. Fu, "An intelligent behavior-based ransomware detection system for android platform," in *2019 First IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. IEEE, 2019, pp. 28–35.

[30] 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.

[31] C. K. Behera and D. L. Bhaskari, "Different obfuscation techniques for code protection," *Procedia Computer Science*, vol. 70, pp. 757–763, 2015.

[32] K. A. Roundy and B. P. Miller, "Binary-code obfuscations in prevalent packer tools," *ACM Computing Surveys (CSUR)*, vol. 46, no. 1, pp. 1–32, 2013.

[33] K. Coogan, S. Debray, T. Kaochar, and G. Townsend, "Automatic static unpacking of malware binaries," in *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 167–176.

[34] S. Mirjalili, S. Saremi, S. M. Mirjalili, and L. d. S. Coelho, "Multi-objective grey wolf optimizer: a novel algorithm for multi-criterion optimization," *Expert Systems with Applications*, vol. 47, pp. 106–119, 2016.

[35] D. Rodrigues, L. A. Pereira, T. Almeida, J. P. Papa, A. Souza, C. C. Ramos, and X.-S. Yang, "Bcs: A binary cuckoo search algorithm for feature selection," in *2013 IEEE International symposium on circuits and systems (ISCAS)*. IEEE, 2013, pp. 465–468.

[36] S. Maniath, A. Ashok, P. Poornachandran, V. Sujadevi, P. S. AU, and S. Jan, "Deep learning lstm based ransomware detection," in *2017 Recent Developments in Control, Automation & Power Engineering (RDCAPE)*. IEEE, 2017, pp. 442–446.

[37] Cuckoo. [Online]. Available: https://cuckoo.readthedocs.io/en/latest/

[38] A. Cohen and N. Nissim, "Trusted detection of ransomware in a private cloud using machine learning methods leveraging meta-features from volatile memory," *Expert Systems with Applications*, vol. 102, pp. 158–178, 2018.

[39] Volatility framework. [Online]. Available: https://www.volatilityfoundation.org/

[40] G. Hill and X. Bellekens, “Cryptoknight: generating and modelling compiled cryptographic primitives,” *Information*, vol. 9, no. 9, p. 231, 2018.

[41] E. Ketzaki, P. Toupas, K. M. Giannoutakis, A. Drosou, and D. Tzovaras, “A behaviour based ransomware detection using neural network models,” in *2020 10th International Conference on Advanced Computer Information Technologies (ACIT).* IEEE, 2020, pp. 747–750.

[42] D. Sgandurra, L. Muñoz-González, R. Mohsen, and E. C. Lupu, “Automated dynamic analysis of ransomware: Benefits, limitations and use for detection,” *arXiv preprint arXiv:1609.03020*, 2016.

[43] M. Verma, P. Kumarguru, S. B. Deb, and A. Gupta, “Analysing indicator of compromises for ransomware: Leveraging iocs with machine learning techniques,” in *2018 IEEE International Conference on Intelligence and Security Informatics (ISI).* IEEE, 2018, pp. 154–159.

[44] B. A. S. Al-Rimy, M. A. Maarof, M. Alazab, F. Alsolami, S. Z. M. Shaid, F. A. Ghaleb, T. Al-Hadhrami, and A. M. Ali, “A pseudo feedback-based annotated tf-idf technique for dynamic crypto-ransomware pre-encryption boundary delineation and features extraction,” *IEEE Access*, vol. 8, pp. 140 586–140 598, 2020.

[45] B. A. S. Al-rimy, M. A. Maarof, M. Alazab, S. Z. M. Shaid, F. A. Ghaleb, A. Almalawi, A. M. Ali, and T. A. Hadhrami, “Redundancy coefficient gradual up-weighting-based mutual information feature selection technique for crypto-ransomware early detection,” *Future Gener. Comput. Syst.*, vol. 115, pp. 641–658, 2021. [Online]. Available: https://doi.org/10.1016/j.future.2020.10.002

[46] N. Scaife, H. Carter, P. Traynor, and K. R. Butler, “Cryptolock (and drop it): stopping ransomware attacks on user data,” in *2016 IEEE 36th international conference on distributed computing systems (ICDCS).* IEEE, 2016, pp. 303–312.

[47] O. M. Alhawi, J. Baldwin, and A. Dehghantanha, “Leveraging machine learning techniques for windows ransomware network traffic detection,” in *Cyber threat intelligence.* Springer, 2018, pp. 93–106.

[48] M. Hirano, “Open repository of the ransap dataset,” https://github.com/manabu-hirano/RanSAP/, accessed: 22.08.2022.

[49] A. Kharaz, S. Arshad, C. Mulliner, W. Robertson, and E. Kirda, “{UNVEIL}: A {Large-Scale}, automated approach to detecting ransomware,” in *25th USENIX security symposium (USENIX Security 16)*, 2016, pp. 757–772.

[50] Windows filesystem minifilter driver framework. [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/filter-manager-concepts

[51] J. Lin, “Divergence measures based on the shannon entropy,” *IEEE Transactions on Information theory*, vol. 37, no. 1, pp. 145–151, 1991.

[52] F. Tang, B. Ma, J. Li, F. Zhang, J. Su, and J. Ma, “Ransomspector: An introspection-based approach to detect crypto ransomware,” *Computers & Security*, vol. 97, p. 101997, 2020.

[53] Z. A. Genç, G. Lenzini, and P. Y. Ryan, "Security analysis of key acquiring strategies used by cryptographic ransomware," in *Proceedings of the Central European Cybersecurity Conference 2018*, 2018, pp. 1–6.

[54] M. Alam, S. Bhattacharya, S. Dutta, S. Sinha, D. Mukhopadhyay, and A. Chattopadhyay, "Ratafia: ransomware analysis using time and frequency informed autoencoders," in *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2019, pp. 218–227.

[55] J.-y. Paik, J.-H. Choi, R. Jin, J. Wang, and E.-S. Cho, "Buffer management for identifying crypto-ransomware attack in environment with no semantic information," in *2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. IEEE, 2019, pp. 443–450.

[56] T. McIntosh, J. Jang-Jaccard, P. Watters, and T. Susnjak, "The inadequacy of entropy-based ransomware detection," in *International conference on neural information processing*. Springer, 2019, pp. 181–189.

[57] J. Pont, B. Arief, and J. Hernandez-Castro, "Why current statistical approaches to ransomware detection fail," in *International Conference on Information Security*. Springer, 2020, pp. 199–216.

[58] A. Continella, A. Guagnelli, G. Zingaro, G. De Pasquale, A. Barenghi, S. Zanero, and F. Maggi, "Shieldfs: a self-healing, ransomware-aware filesystem," in *Proceedings of the 32nd annual conference on computer security applications*, 2016, pp. 336–347.

[59] A. Kharraz and E. Kirda, "Redemption: Real-time protection against ransomware at end-hosts," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2017, pp. 98–119.

[60] S. Mehnaz, A. Mudgerikar, and E. Bertino, "Rwguard: A real-time detection system against cryptographic ransomware," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2018, pp. 114–136.

[61] C. E. Shannon, "A mathematical theory of communication," *The Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[62] IBM, "Boxnote containing used ransomware samples," https://ibm.ent.box.com/folder/170481751288?s=pqo2wue3r73y97ae0b46y2w5jg1g083h, accessed: 13.12.2022.

[63] Acronis, "Taking a deep dive into sodinokibi ransomware," https://www.acronis.com/en-us/cyber-protection-center/posts/sodinokibi-ransomware/, accessed: 11.11.2022.

[64] T. M. Research, "Ransomware spotlight - black basta," https://www.trendmicro.com/vinfo/us/security/news/ransomware-spotlight/ransomware-spotlight-blackbasta, accessed: 19.09.2022.

[65] Unit42, "Lockbit 2.0: How this raas operates and how to protect against it," https://unit42.paloaltonetworks.com/lockbit-2-ransomware/, accessed: 11.11.2022.

[66] I. Security, "X-force threat intelligence index malware analysis report lockfile malware profile," 2022.

[67] secureworks, "Wcry ransomware analysis," https://www.secureworks.com/research/wcry-ransomware-analysis, accessed: 11.11.2022.

[68] P. SecurityPro, "Decoding conti," https://security.packt.com/conti/, accessed: 13.12.2022.

[69] any.run, "Malware-trends: Sodinokibi," https://any.run/malware-trends/sodinokibi, accessed: 15.09.2022.

[70] Microsoft, "Win32k elevation of privilege vulnerability," https://msrc.microsoft.com/update-guide/vulnerability/CVE-2018-8453, accessed: 15.09.2022.

[71] I. Security, "X-force threat intelligence malware analysis report lockbit ransomware," 2022.

[72] T. M. Research, "Ransomware spotlight - lockbit," https://www.trendmicro.com/vinfo/us/security/news/ransomware-spotlight/ransomware-spotlight-lockbit, accessed: 19.09.2022.

[73] Malwarebazaar database. [Online]. Available: https://bazaar.abuse.ch/browse.php?search=tag%3Alockbit

[74] T. Meskauskas, "What is lockfile ransomware?" https://www.pcrisk.com/removal-guides/21620-lockfile-ransomware, accessed: 15.09.2022.

[75] W. Alraddadi and H. Sarvotham, "A comprehensive analysis of wannacry: technical analysis, reverse engineering, and motivation."

[76] Q. Chen and R. A. Bridges, "Automated behavioral analysis of malware: A case study of wannacry ransomware," in *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2017, pp. 454–460.

[77] P. Prasad, N. Sowmya, K. Reddy, and P. Bala, "Introduction to dynamic malware analysis for cyber intelligence and forensics," *International Journal of Mechanical Engineering and Technology*, vol. 9, no. 1, pp. 10–21, 2018.

[78] S. Algarni, "Cybersecurity attacks: Analysis of "wannacry" attack and proposing methods for reducing or preventing such attacks in future," in *ICT Systems and Sustainability*. Springer, 2021, pp. 763–770.

[79] M. Akbanov, V. G. Vassilakis, and M. D. Logothetis, "Wannacry ransomware: Analysis of infection, persistence, recovery prevention and propagation mechanisms," *Journal of Telecommunications and Information Technology*, 2019.

[80] H. Security, "All about conti ransomware," https://heimdalsecurity.com/blog/what-is-conti-ransomware/, accessed: 13.12.2022.

[81] L. T. DeCarlo, "On the meaning and use of kurtosis." *Psychological methods*, vol. 2, no. 3, p. 292, 1997.

[82] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[83] Scikitlearn, "Sklearn.ensemble.randomforestclassifier," https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html, accessed: 13.12.2022.

[84] Nvidia, "What is xgboost?" https://www.nvidia.com/en-us/glossary/data-science/xgboost/, accessed: 13.12.2022.

[85] Keras, "The sequential model," https://keras.io/guides/sequential_model/, accessed: 13.12.2022.

[86] Kaggle, "Random forest vs xgboost vs deep neural network," https://www.kaggle.com/code/arathee2/random-forest-vs-xgboost-vs-deep-neural-network/report, accessed: 13.12.2022.

[87] Systemtap sourceware. [Online]. Available: https://sourceware.org/systemtap/

[88] Using systemtap. [Online]. Available: https://sourceware.org/systemtap/SystemTap_Beginners_Guide/using-systemtap.html

[89] Ent tool. [Online]. Available: https://www.fourmilab.ch/random/

[90] E. Scornet, "Trees, forests, and impurity-based variable importance," *arXiv preprint arXiv:2001.04295*, 2020.

[91] A. Altmann, L. Toloşi, O. Sander, and T. Lengauer, "Permutation importance: a corrected feature importance measure," *Bioinformatics*, vol. 26, no. 10, pp. 1340–1347, 2010.

[92] Selectkbest. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html

Abbreviations

ADA	American Dental Association
APK	Android Application Package
API	Application Programming Interface
CNN	Convolutional Neural Networks
CSV	Comma Separated Value
DAS	Direct Area Storage
DDL	Dynamic-Link Library
DDoS	Distributed Denial Of Service
DNN	Deep Neural Network
deep RL	Deep Reinforcement Learning
FFT	Fast Fourier Transformation
GUI	Graphical User Interface
HDD	Hard Disk Drives
HPC	Hardware Performance Counters
IO	Input/Output
IOPS	Input/Output Operations Per Second
IoT	Internet Of Things
ISM	Image Similarity Measurement
JAR	Java Archive
KNN	K-Nearest Neighbour
LBA	Logical Block Addressing
MBR	Master Boot Record
NAS	Network-based storage
NTFS	New Technology File System
OCR	Optical Character Recognition
OS	Operating Sytstem
RaaS	Ransomware-as-a-Service
RHEL	Red Hat Enterprise Linux
RNN	Recurrent Neural Networks
SAN	Storage Area Network
SSD	Solid-state Drives
SSM	String Similarity Measurement
SVM	Support Vector Machine
VMI	Virtual Machine Introspection

List of Figures

List of Tables

Appendix A

Code and Installation Guidelines

This Chapter provides the code structure and installation guidelines of the repository developed in this master project. It further contains the code used to perform the feature extraction.

A.1 Structure of the Code Repository

The code developed to perform the experiments of this master project is stored in a repository which follows the following structure. For each folder, only the contained experiment ReadMe's are shown. Each experiment ReadMe's describes how to recreate the experiment and which code files need to be used for it. The code files and results are also contained in the repository, but not shown in this overview since the experiment readMe's refer to them clearly:

- benign_workload_file_converter
  - EXP005_readme: Bash script for benign workloads (Contributors Dario Gagulic and Lynn Zumtaugwald)
  - EXP017_readme: Detect Data LBA (Sequential Reader) (Contributors Dario Gagulic and Lynn Zumtaugwald)
- ent_extended
  - README: ENT Fourmilab Random Sequence Tester (Modified by Siddhant Sahu)
- entropy_pre_post_analyse
  - README: Trace Analysis (Contributor Siddhant Sahu)
- hirano_2019_recreation

  - EXP002_readme: Windowing Information leak analysis (Contributors Dario Gagulic and Lynn Zumtaugwald)
  - EXP003_readme: Train and Evaluate Hirano2019 models with own traces (Contributors Dario Gagulic and Lynn Zumtaugwald)
  - EXP004_readme: Plot Features with own traces (Contributors Dario Gagulic and Lynn Zumtaugwald)

- hirano_2022_recreation
  - EXP001_readme: Recreate Hirano2022 confusion matrix (Contributors Dario Gagulic and Lynn Zumtaugwald)

- ML_pipeline
  - EXP015_readme: Feature Extractor Script (Contributors Dario Gagulic and Lynn Zumtaugwald)
  - EXP030_readme: Extract new features from own traces (Method 2: 5fold on windows) (Contributors Dario Gagulic and Lynn Zumtaugwald)
  - EXP031_readme: Generalizability to unseen ransomware (Method2: 5fold on windows) (Contributors Dario Gagulic and Lynn Zumtaugwald)
  - EXP032_readme: Generalizability to different setups s1 and s2 (Method2: 5fold on windows) (Contributors Dario Gagulic and Lynn Zumtaugwald)
  - EXP033_readme: Performance on different amounts of trace time (Method2: 5fold on windows) (Contributors Dario Gagulic and Lynn Zumtaugwald)
  - EXP034_readme: Evaluation of horizontal window stacking feature (Method2: 5fold on windows) (Contributors Dario Gagulic and Lynn Zumtaugwald)
  - EXP035_readme: Evaluation of mixed workloads (Method2: 5fold on windows) (Contributors Dario Gagulic and Lynn Zumtaugwald)
  - EXP037_readme: Feature Importance for Random Forrest with Impurity-Based and Feature Permutation (Method 2 feature extraction) (Contributor Siddhant Sahu)
  - hirano
    - EXP018_readme: Script to extract new features from Hirano's traces (Contributors Dario Gagulic and Lynn Zumtaugwald)
    - EXP019_readme: Confusion matrix Hirano's traces new features (Contributors Dario Gagulic and Lynn Zumtaugwald)
    - EXP020_readme: Merge workloads (Contributors Dario Gagulic and Lynn Zumtaugwald)
    - EXP021_readme: Hirano mixed workloads confusion matrix (Contributors Dario Gagulic and Lynn Zumtaugwald)
    - EXP022_readme: Feature Correlation Analysis (Contributors Dario Gagulic and Lynn Zumtaugwald)

    * EXP023_readme: Latency Analysis (Contributors Dario Gagulic and Lynn Zumtaugwald)
  - method1
    * EXP024_readme: Extract new features from own traces (Method 1: Sampling of IO's) (Contributors Dario Gagulic and Lynn Zumtaugwald)
    * EXP025_readme: Confusion matrix own traces features A,B,C,E,F,G,H (Method1: Sampling of IO's) (Contributors Dario Gagulic and Lynn Zumtaugwald)
    * EXP026_readme: Generalizability to unseen ransomware (Method1: Sampling of IO's) (Contributors Dario Gagulic and Lynn Zumtaugwald)
    * EXP027_readme: Generalizability to different setups s1 and s2 (Method1: Sampling of IO's) (Contributors Dario Gagulic and Lynn Zumtaugwald)
    * EXP028_readme: Performance on different amounts of trace time (Method1: Sampling) (Contributors Dario Gagulic and Lynn Zumtaugwald)
    * EXP029_readme: Evaluation of horizontal window stacking feature (Method1: Sampling) (Contributors Dario Gagulic and Lynn Zumtaugwald)
    * EXP036_readme: Feature Importance for Random Forrest with Impurity-Based and Feature Permutation (Method 1 feature extraction) (Contributor Siddhant Sahu)

- trace_analyse
  - README: Trace Analysis (Contributor Siddhant Sahu)

A.2 Installation Guidelines and Dependencies

This Section provides guidance regarding the prerequisites needed for running the produced code during this masters project. The Python code is compatible with version 3.9.0. The following Libraries need to be installed in order to run the scripts from this repository.

- XlsxWriter 3.0.3
- joblib 1.2.0
- keras 2.11.0
- matplotlib 3.6.2
- numpy 1.23.4
- pandas 1.5.1
- scikit-learn 1.1.2

- scipy 1.9.3
- seaborn 0.12.1
- tensorflow 2.11.0
- xgboost 1.7.1

A.3 Feature Extraction Code

```
class FeatureExtractor:
    """
    A class to extract features from dataframes grouping by specified window sizes and
                                                window offsets and stacking a specified
                                               amount of windows

    The currently implemented feature set is: SMA_transfer_size_rd,
                                               SMA_transfer_size_wr, Entropy_mean_wr,
                                               Variance_LBA_rd, Variance_LBA_wr,
                                               Kurtosis_LBA_rd, Kurtosis_LBA_wr,
                                               Kurtosis_entropy_wr, Rewrite_rate,
                                               Entropy_rewrites, Variance_entropy_wr,
                                               Slope_entropy_wr
    """

    def __init__(self, datasets_reads: list[pandas.DataFrame], datasets_writes: list[
                                               pandas.DataFrame],
                 windows: list[int],
                 offsets: list[int], save_file: str, stacking: int = 1):
        self.windows = windows
        self.datasets_reads = datasets_reads
        self.datasets_writes = datasets_writes
        self.offsets = offsets
        self.save_file = save_file
        self.stacking = stacking
        self.features_with_label = ['SMA_transfer_size_rd', 'SMA_transfer_size_wr', '
                                               Entropy_mean_wr', 'Variance_LBA_rd', '
                                               Variance_LBA_wr', 'Kurtosis_LBA_rd', '
                                               Kurtosis_LBA_wr', 'Kurtosis_entropy_wr', '
                                               Rewrite_rate', 'Entropy_rewrites', '
                                               Variance_entropy_wr', 'Slope_entropy_wr', '
                                               Label']
        self.features = ['SMA_transfer_size_rd', 'SMA_transfer_size_wr', '
                                               Entropy_mean_wr', 'Variance_LBA_rd', '
                                               Variance_LBA_wr','Kurtosis_LBA_rd', '
                                               Kurtosis_LBA_wr', 'Kurtosis_entropy_wr', '
                                               Rewrite_rate', 'Entropy_rewrites', '
                                               Variance_entropy_wr', 'Slope_entropy_wr']

    def extract(self):
        """
        Extracts feature vectors for multiple dataset and multiple window sizes and
                                               offsets and stores them in a csv file
        """
        # create dataframe
        combined_df = pd.DataFrame(columns=self.features_with_label)

        # create files
        for window, offset in zip(self.windows, self.offsets):
            filename = self.save_file.replace('.csv', '_ws{window}_wo{offset}.csv')
            combined_df.to_csv(filename, index=False)

        # iterate over all given dataframes containing reads and dataframes containing
                                            writes
        # and extract feature windows
        for df_rd, df_wr in zip(self.datasets_reads, self.datasets_writes):
            feature_dataframes = self.windowing(df_rd=df_rd, df_wr=df_wr)

            # save all the feature vectors
            for feature_dataframe, window, offset in zip(feature_dataframes, self.
                                            windows, self.offsets):
                feature_dataframe.to_csv(f"{self.save_file.replace('.csv', f'_ws{
                                            window}_wo{offset}.csv')}", mode='a',
                                          header=False, index=False)

    def windowing(self, df_rd: pandas.DataFrame, df_wr: pandas.DataFrame):
        """
        Computes feature vectors for different window sizes and window offsets
        :param df_rd: pandas.DataFrame for reads
        :param df_wr: pandas.DataFrame for writes
        :return: list: [pandas.Dataframe with feature vectors per window size and
                                            window offset]
        """
        feature_vectors = []
        for window_size, window_offset in zip(self.windows, self.offsets):
            if self.stacking == 1:
                X = pd.DataFrame(
                    columns=self.features)
            else:
                X = pd.DataFrame(
                    columns=self.features * self.stacking)

            # compute the number of windows
            max_second = df_rd['Timestamp_ms'].max()
            max_second_w = df_wr['Timestamp_ms'].max()
            max_iter = 1 + (max_second - window_size + window_offset) // window_offset
            max_iter_w = 1 + (max_second_w - window_size + window_offset) //
                                            window_offset
            if max_iter_w > max_iter:
                max_iter = max_iter_w
                max_second = max_second_w

            # create windows
            stacked_windows = []
            stacked = 0
            for i in range(0, max_iter):

                min = i * window_offset
                max = min + window_size - 1
                if max > max_second:
                    break
                df_span_rd = df_rd.loc[(df_rd['Timestamp_ms']).between(min, max)]
                df_span_wr = df_wr.loc[(df_wr['Timestamp_ms']).between(min, max)]

                # create feature row and append it to the dataframe
                row = self.compute_features(df_span_rd=df_span_rd, df_span_wr=
                                            df_span_wr,
                                             window_size=window_size)

                # save the computed feature row
                if self.stacking == 1:
                    X.loc[len(X)] = row
                else:
                    stacked_windows.extend(row)
                    stacked += 1

                if self.stacking != 1 and stacked == self.stacking:
                    X.loc[len(X)] = stacked_windows
                    stacked_windows, stacked = [], 0

            # set the label
            lbl = int(df_rd.head(1)['Label'])
            X['Label'] = lbl
            # save the finished feature vector data frame in list
            feature_vectors.append(X)

        return feature_vectors

    def compute_features(self, df_span_rd: pandas.DataFrame, df_span_wr: pandas.
                                            DataFrame, window_size=1000):
        """
        Calls various functions to compute features and stores the features in a list
        :param df_span_rd: pandas.Dataframe for reads
        :param df_span_wr: pandas.Dataframe for writes
        :param window_size: int: the current window size in ms
        :return: list: feature vector
        """
        row = []

        # simple moving average transfer size
        row.extend(self.sma_transfer_size(df_span_rd=df_span_rd, df_span_wr=df_span_wr
                                            , window_size=window_size))

        # mean Entropy
        row.append(self.mean_entropy(df_span_wr=df_span_wr))

        # Variance of LBA
        row.extend(self.variance_lba(df_span_rd=df_span_rd, df_span_wr=df_span_wr))

        # Difference Variance LBA writes and Variance of LBA reads
        # removed feature (because of high correlation with variance of LBA writes
                                            feature)
        # row.append(self.difference_variance_lba_writes_and_reads(variance_lba_reads=
                                            row[3], variance_lba_writes=row[4]))

        # Kurtosis of LBA
        row.extend(self.kurtosis_lba(df_span_rd=df_span_rd, df_span_wr=df_span_wr))

        # Kurtosis of Entropy
        row.append(self.kurtosis_entropy(df_span_wr=df_span_wr))

        # Rewrite Rate and entropy of rewrites
        row.extend(self.rewrite_rate_and_entropy(df_span_rd=df_span_rd, df_span_wr=
                                                 df_span_wr))

        # Variance of entropy
        row.append(self.variance_entropy(df_span_wr=df_span_wr))

        # Slope of entropy
        row.append(self.slope_entropy(df_span_wr=df_span_wr))

        return row

    @staticmethod
    def sma_transfer_size(df_span_rd: pandas.DataFrame, df_span_wr: pandas.DataFrame,
                                                 window_size: int):
        """
         Calculates the simple moving average transfer size in bits for reads and
                                                 writes
         :param df_span_rd: pandas.Dataframe: the dataframe containing reads
         :param df_span_wr: pandas.Dataframe: the dataframe containing writes
         :param window_size: int the window size in milliseconds
         :return: list: [float: sma transfer size for reads, float: sma transfer size
                                                 for writes]
        """
        if len(df_span_rd.index) > 1:
            read = df_span_rd['Block_size_Bytes'].sum() / (df_span_rd.shape[0] *
                                                 window_size * 0.001)
        else:
            read = 0

        if len(df_span_wr.index) > 1:
            write = df_span_wr['Block_size_Bytes'].sum() / (df_span_wr.shape[0] *
                                                 window_size * 0.001)
        else:
            write = 0
        return [read, write]

    @staticmethod
    def mean_entropy(df_span_wr: pandas.DataFrame):
        """
        Calculated the mean entropy for writes
        :param df_span_wr: pandas.Dataframe: the dataframe with the writes
        :return: float: entropy
        """
        return df_span_wr['Entropy'].mean()

    @staticmethod
    def variance_lba(df_span_rd: pandas.DataFrame, df_span_wr: pandas.DataFrame):
        """
        Computes the variance of LBA
        :param df_span_rd: pandas.Dataframe for reads
        :param df_span_wr: pandas.Dataframe for writes
        :return: list: [varince of LBA reads, Variance of LBA writes]
        """
        return [df_span_rd['LBA'].var(), df_span_wr['LBA'].var()]

    @staticmethod
    def difference_variance_lba_writes_and_reads(variance_lba_reads: float,
                                       variance_lba_writes: float):
        """
        Computes the difference of the variance of LBA writes and the variance of LBA
                                       reads
        :param variance_lba_reads: float: variance of LBA reads
        :param variance_lba_writes: float: variance of LBA writes
        :return: float: the difference of the variance of LBA writes and the variance
                                       of LBA reads
        """
        return variance_lba_writes - variance_lba_reads

    @staticmethod
    def kurtosis_lba(df_span_rd: pandas.DataFrame, df_span_wr: pandas.DataFrame):
        """
        Computes the kurtosis of LBA reads and writes
        :param df_span_rd: pandas.Dataframe for reads
        :param df_span_wr: pandas.Dataframe for writes
        :return: list: [kurtosis of LBA reads, kurtosis of LBA writes]
        """
        return [df_span_rd['LBA'].kurtosis(skipna=True), df_span_wr['LBA'].kurtosis(
                                       skipna=True)]

    @staticmethod
    def kurtosis_entropy(df_span_wr: pandas.DataFrame):
        """
        Computes the kurtosis of entropy for writes
        :param df_span_wr: pandas.Dataframe for writes
        :return: float: kurtosis of entropy writes
        """
        return df_span_wr['Entropy'].kurtosis(skipna=True)

    @staticmethod
    def rewrite_rate_and_entropy(df_span_rd: pandas.DataFrame, df_span_wr: pandas.
                                       DataFrame):
        """
        Computes the number of reads where there is also a write in place (=to the
                                       same LBA) and their mean entropy
        :param df_span_rd: pandas.Dataframe for reads
        :param df_span_wr: pandas.Dataframe for writes
        :return: list: [int: the number of rewrites, float: mean entropy of rewrites]
        """
        number_of_rewrites = 0
        entropy_of_rewrites = []
        merged = pd.merge(df_span_rd, df_span_wr, on='LBA', how='inner')
        merged.drop_duplicates(['Timestamp_ms_y', 'Block_size_Bytes_y', 'Entropy'],
                                       inplace=True)
        for row in merged.itertuples():
            if row.Timestamp_ms_x <= row.Timestamp_ms_y:
                number_of_rewrites += 1
                entropy_of_rewrites.append(row.Entropy)

        if entropy_of_rewrites:
            mean_entropy_rewrites = sum(entropy_of_rewrites) / len(entropy_of_rewrites
                                                                )
        else:
            mean_entropy_rewrites = 0
        return [number_of_rewrites, mean_entropy_rewrites]

    @staticmethod
    def variance_entropy(df_span_wr: pandas.DataFrame):
        """
        Computes the variance of entropy writes
        :param df_span_wr: pandas.Dataframe for writes
        :return: float: variance of entropy writes
        """
        return df_span_wr['Entropy'].var()

    @staticmethod
    def slope_entropy(df_span_wr: pandas.DataFrame):
        """
        Computes the slope of entropy writes by fitting a linear least squares model
        :param df_span_wr: pandas.Dataframe for writes
        :return: float: slope of entropy writes (linear least squares fit)
        """

        if df_span_wr.shape[0] > 0 and df_span_wr.Timestamp_ms.unique().size > 1:
            x = df_span_wr['Timestamp_ms']
            y = df_span_wr['Entropy']
            res = stats.linregress(x.astype(float), y.astype(float))
            slope = res.slope
        else:
            slope = np.NAN

        return slope
```

Appendix B

Contributions of Students

This chapter states in a first section the contributions of students to this report (Dario Gagulic 41.78 %, Lynn Zumtaugwald 41.78 % and Siddhant Sahu 16.43 %) and in a second section the contributions of students to the task list of this master project (Dario Gagulic 39.75 %, Lynn Zumtaugwald 39.75 % and Siddhant Sahu 20.5 %).

B.1 Writing of the Report

Table B.1 shows the contributions from each student to the report. Dario Gagulic has written 41.78 %, Lynn Zumtaugwald 41.78 % and Siddhant Sahu 16.43 % of the report according to the number of chapters, subchapters and subsubchapters. This aligns with the number of pages.

Table B.1: List of individual contributions to the report.

Abstract	Collaboration Lynn Zumtaugwald and Dario Gagulic
Acknowledgements	Collaboration Lynn Zumtaugwald and Dario Gagulic
1.1 Motivation	Collaboration Lynn Zumtaugwald and Dario Gagulic
1.2 Description of work	Collaboration Lynn Zumtaugwald and Dario Gagulic
1.3 Thesis Outline	Collaboration Lynn Zumtaugwald and Dario Gagulic
2. Background	Collaboration Lynn Zumtaugwald and Dario Gagulic
3.1 Ransomware Overview	Collaboration Lynn Zumtaugwald and Dario Gagulic
3.2 Static and Dynamic Detection Approaches	All
3.2.1 Static Detection	Siddhant Sahu
3.2.2 Dynamic Detection	All
3.3 Placement of Ransomware Detection in Computer Systems	Siddhant Sahu
3.3.1 Hardware Level	Siddhant Sahu
3.3.2 Kernel Level	Siddhant Sahu

3.4 Findings from Related Work	Collaboration Lynn Zumtaugwald and Dario Gagulic
4. Design and Implementation	Collaboration Lynn Zumtaugwald and Dario Gagulic
4.1 Reproduction of Related Work	Collaboration Lynn Zumtaugwald and Dario Gagulic
4.1.1 In Depth Analysis of Latest Existing Work in Ransomware Detection in Block Storage Systems	Collaboration Lynn Zumtaugwald and Dario Gagulic
4.1.2 Reproduction of Latext Existing Work in Ransomware Detection in Block Storage Systems	Collaboration Lynn Zumtaugwald and Dario Gagulic
4.2 Design	Collaboration Lynn Zumtaugwald and Dario Gagulic
4.2.1 Ransomware Selection	Collaboration Lynn Zumtaugwald and Dario Gagulic
4.2.1.1 Sodinokibi	Lynn Zumtaugwald
4.2.1.2 BlackBasta	Dario Gagulic
4.2.1.3. Lockbit	Dario Gagulic
4.2.1.4 LockFile	Lynn Zumtaugwald
4.2.1.5 WannaCry	Siddhant Sahu
4.2.1.6 Conti	Collaboration Lynn Zumtaugwald and Dario Gagulic
4.2.2 Feature Selection	Collaboration Lynn Zumtaugwald and Dario Gagulic
4.2.2.1 Feature A: Variance of Entropy of Writes	Collaboration Lynn Zumtaugwald and Dario Gagulic
4.2.2.2 Feature B: Rewrite Rate	Collaboration Lynn Zumtaugwald and Dario Gagulic
4.2.2.3 Feature C: Mean Entropy of Rewrites	Collaboration Lynn Zumtaugwald and Dario Gagulic
4.2.2.4 Feature D: Difference of Variance of LBA Writes and Variance of LBA Reads	Collaboration Lynn Zumtaugwald and Dario Gagulic
4.2.2.5 Feature E: The Slope of Entropy	Collaboration Lynn Zumtaugwald and Dario Gagulic
4.2.2.6 Feature F: Kurtosis of Entropy	Collaboration Lynn Zumtaugwald and Dario Gagulic
4.2.2.7 Feature G and H. Kurtosis of LBA Reads and Kurtosis of LBA Writes	Collaboration Lynn Zumtaugwald and Dario Gagulic
4.2.2.8 Feature I: Stacking of Multiple Windows	Collaboration Lynn Zumtaugwald and Dario Gagulic

4.2.3 Model Selection and Parametrization	Collaboration Lynn Zumtaugwald and Dario Gagulic
4.2.3.1 Random Forest	Collaboration Lynn Zumtaugwald and Dario Gagulic
4.2.3.2 XGBoost	Collaboration Lynn Zumtaugwald and Dario Gagulic
4.2.3.3 DNN	Collaboration Lynn Zumtaugwald and Dario Gagulic
4.3 Implementation	Collaboration Lynn Zumtaugwald and Dario Gagulic
4.3.1 Test Environment for Feature Extraction on IO Operations	Siddhant Sahu
4.3.1.1 Preparation of SSDs for the VMs	Siddhant Sahu
4.3.1.2 Entropy Extraction in the Device Mapper Kernel Module	Siddhant Sahu
4.3.1.3 Collection of IO Traces using System Tap	Siddhant Sahu
4.3.1.4 Trace Collection Steps	Siddhant Sahu
4.3.2 Modification of the ENT tool	Siddhant Sahu
4.3.2 Benign Workload Simulator	Collaboration Lynn Zumtaugwald and Dario Gagulic
4.3.3 Collected Storage Access Patterns	Collaboration Lynn Zumtaugwald and Dario Gagulic
4.3.4 Feature Extraction, Training and Evaluation of Models	Collaboration Lynn Zumtaugwald and Dario Gagulic
4.3.5 Code Implementation	Collaboration Lynn Zumtaugwald and Dario Gagulic
5. Experiments, Results and Discussion	Collaboration Lynn Zumtaugwald and Dario Gagulic
5.1 Trace Validation	Collaboration Lynn Zumtaugwald and Dario Gagulic
5.1.1 Entropy Analysis	Collaboration Lynn Zumtaugwald and Dario Gagulic
5.1.2 IO Time Series Analysis	Collaboration Lynn Zumtaugwald and Dario Gagulic
5.1.3 Active Parts	Collaboration Lynn Zumtaugwald and Dario Gagulic
5.2 Feature Analysis	Collaboration Lynn Zumtaugwald and Dario Gagulic
5.2.1 Performance Evaluation of New Features A to H	Collaboration Lynn Zumtaugwald and Dario Gagulic
5.2.2 Feature Correlation	Collaboration Lynn Zumtaugwald and Dario Gagulic
5.2.3 Analysis of Stacking Feature I	Collaboration Lynn Zumtaugwald and Dario Gagulic
5.2.4 Feature Importance	Siddhant Sahu
5.2.5 Theoretical Overhead Analysis	Collaboration Lynn Zumtaugwald and Dario Gagulic

5.3 Impact of Trace Data Selection on the Model Performance	Collaboration Lynn Zumtaugwald and Dario Gagulic
5.4 Generalizability	Collaboration Lynn Zumtaugwald and Dario Gagulic
5.4.1 Storage System Setups	Collaboration Lynn Zumtaugwald and Dario Gagulic
5.4.2 Mixed Workloads	Collaboration Lynn Zumtaugwald and Dario Gagulic
5.4.3 Unseen Ransomware	Collaboration Lynn Zumtaugwald and Dario Gagulic
5.5 Comparative Discussion of All Experiments and Results	Collaboration Lynn Zumtaugwald and Dario Gagulic
6. Summary and Conclusion	Collaboration Lynn Zumtaugwald and Dario Gagulic
Appendix A Code and Installation Guidelines	Collaboration Lynn Zumtaugwald and Dario Gagulic
A.1 Structure of the Code Repository	Collaboration Lynn Zumtaugwald and Dario Gagulic
A.2 Installation Guidelines and Dependencies	Collaboration Lynn Zumtaugwald and Dario Gagulic
A.3 Feature Extraction Code	Collaboration Lynn Zumtaugwald and Dario Gagulic
Appendix B Contributions of Students	Collaboration Lynn Zumtaugwald and Dario Gagulic
B.1 Writing of the Report	Collaboration Lynn Zumtaugwald and Dario Gagulic
B.2 Master Project Tasks	Collaboration Lynn Zumtaugwald and Dario Gagulic

B.2 Master Project Tasks

The following task list B.2 shows the tasks that have been completed and which students contributed to it. It has been extracted from the excel lists from the bi-daily scrum meetings. Based on this list and the number of tasks, Dario Gagulic has completed 39.75 % of the tasks, Lynn Zumtaugwald has completed 39.75 % of the tasks and Siddhant Sahu has completed 20.5 % of the tasks. Dario Gagulic and Lynn Zumtaugwald often worked in collaboration on tasks. Every student had smaller and bigger tasks to complete. The planning and realization of presentation slides (weekly meeting with IBM and biweekly with UZH) have mostly been created by Dario Gagulic and Lynn Zumtaugwald and are not on this list. However, Siddhant Sahu has also added his results. The midterm presentation slides have been created by Dario Gagulic and Lynn Zumtaugwald. Everyone participated in the trace collection. From the final valid traces used in the experiments of this project, 5 have been collected by Siddhant Sahu and 12 have been collected by Dario Gagulic and Lynn Zumtaugwald. However, many more traces have been collected by all parties, but they were not considered as valid.

Table B.2: List of tasks completed.

Task	student(s)
Add ideas for additional features in Boxnote	Collaboration Dario Gagulic, Lynn Zumtaugwald
add machine and trace time to time series ppp and to trace analysis ppp	Collaboration Dario Gagulic, Lynn Zumtaugwald
Adjust Code of feature extraction to work with ms	Collaboration Dario Gagulic, Lynn Zumtaugwald
Adjust midterm presentation slides according to Albertos feedback	Collaboration Dario Gagulic, Lynn Zumtaugwald
Adjust midterm presentation slides according to ZRL-Team feedback	Collaboration Dario Gagulic, Lynn Zumtaugwald
Analysis of mixed workload (LockBit + Bening and BlackBasta + Benign on trained model, waiting for 3rd trace)	Collaboration Dario Gagulic, Lynn Zumtaugwald
Apply naming convention to folders and files of traces	Collaboration Dario Gagulic, Lynn Zumtaugwald
Apply new improvement suggestion from Roman In report (diverse chapters) and placement of related work	Collaboration Dario Gagulic, Lynn Zumtaugwald
Apply romans improvement suggestions on report	Collaboration Dario Gagulic, Lynn Zumtaugwald
Benign Workload file conversion PowerShell Script *	Collaboration Dario Gagulic, Lynn Zumtaugwald
Changed Method of Feature Extraction in Code *	Collaboration Dario Gagulic, Lynn Zumtaugwald
Collect benign workload trace (setup 1 OS&Data seperate) * (experiment entry containing link to trace, histogram and deltas in box)	Collaboration Dario Gagulic, Lynn Zumtaugwald
Collect benign workload trace (setup 2 OS&Data single) * (experiment entry containing link to trace, histogram and deltas in box & difference for the two setups)	Collaboration Dario Gagulic, Lynn Zumtaugwald
Evaluation of Stacking Feature with three models and two different possible setups (ws5000,stack2, WS2000, stack5) *	Collaboration Dario Gagulic, Lynn Zumtaugwald
Execute Lockfile with new sample from Malware Bazzaar	Collaboration Dario Gagulic, Lynn Zumtaugwald
EXP001 - Recreate Hirano2021 confusion matrix *	Collaboration Dario Gagulic, Lynn Zumtaugwald
EXP002 - Windowing Information leak analysis *	Collaboration Dario Gagulic, Lynn Zumtaugwald
EXP003 - Train and Evaluate Hirano2019 models with own traces *	Collaboration Dario Gagulic, Lynn Zumtaugwald

EXP004 - Plot 5 Features with own traces *	Collaboration Dario Gagulic, Lynn Zumtaugwald
Feature correlation analysis *	Collaboration Dario Gagulic, Lynn Zumtaugwald
Fix Introdacqua disk partitions	Collaboration Dario Gagulic, Lynn Zumtaugwald
Functions to extract/compute new features *	Collaboration Dario Gagulic, Lynn Zumtaugwald
Generalizability to s1 and s2 (train with s1 evaluate with s2 and vice versa) *	Collaboration Dario Gagulic, Lynn Zumtaugwald
Generalizability to unseen ransomware (train with 5, evaluate on 6th) *	Collaboration Dario Gagulic, Lynn Zumtaugwald
Implement and Integrate DNN model	Collaboration Dario Gagulic, Lynn Zumtaugwald
Last iteration for Report: Read through, ensure reading flow, correct if necessary	Collaboration Dario Gagulic, Lynn Zumtaugwald
Latency experiment (for loop window 1, window1+2, window 1+2+3)	Collaboration Dario Gagulic, Lynn Zumtaugwald
Planning Phase (Trello, Gantt Chart)	Collaboration Dario Gagulic, Lynn Zumtaugwald
Preparation of midterm presentation slides	Collaboration Dario Gagulic, Lynn Zumtaugwald
Re-Executed all Experiments *	Collaboration Dario Gagulic, Lynn Zumtaugwald
Renaming ent files to adhere to naming convention	Collaboration Dario Gagulic, Lynn Zumtaugwald
Renaming trace files to adhere to naming convention (date_workload_s1/s2/s3_wr/rd.csv) and create split dat file into rd and wr csv files	Collaboration Dario Gagulic, Lynn Zumtaugwald
Re-run Blackbasta (incl. ent extraction and trace analysis)	Collaboration Dario Gagulic, Lynn Zumtaugwald
Re-run Lockbit (incl. ent extraction and trace analysis) S1	Collaboration Dario Gagulic, Lynn Zumtaugwald
Re-run Lockbit (incl. ent extraction and trace analysis) S2	Collaboration Dario Gagulic, Lynn Zumtaugwald
Retake WannaCry with .NET installed S1 (as discussed with roman)	Collaboration Dario Gagulic, Lynn Zumtaugwald
Retrain models (redo experiments) with new, valid traces (leave out GandCrab)	Collaboration Dario Gagulic, Lynn Zumtaugwald
Retry different samples of GandCrab	Collaboration Dario Gagulic, Lynn Zumtaugwald
Retry GandCrab S2	Collaboration Dario Gagulic, Lynn Zumtaugwald
Started taking traces of Conti (to replace GandCrab)	Collaboration Dario Gagulic, Lynn Zumtaugwald

Tracing of Mixed Workload Traces for WannaCry + Benign *	Collaboration Dario Gagulic, Lynn Zumtaugwald
Train Hiranos models with original+new features on our Traces 12min each and Random Forest *	Collaboration Dario Gagulic, Lynn Zumtaugwald
Train Hiranos models with original+new features on our Traces 12min each and XGBoost (balanced for multi&binary each)	Collaboration Dario Gagulic, Lynn Zumtaugwald
Unittest for feature extraction code	Collaboration Dario Gagulic, Lynn Zumtaugwald
collect Traces for Lockfile S2	Collaboration Dario Gagulic, Lynn Zumtaugwald
Entropy analysis (WannaCry S1)	Collaboration Dario Gagulic, Lynn Zumtaugwald
mixed workloads (also for training/ and without training) 1/2, 1/3, 1/5	Collaboration Dario Gagulic, Lynn Zumtaugwald
Train Hiranos models with new features and Hiranos Traces *	Collaboration Dario Gagulic, Lynn Zumtaugwald
Collect traces for BlackBasta (S1 & S2) * (collected traces for S2 with no ransomware note, change in disk recognizable)	Collaboration Dario Gagulic, Lynn Zumtaugwald
Retake WannaCry with .NET installed S1 (as discussed with roman)	Collaboration Dario Gagulic, Lynn Zumtaugwald
Experiment which LBA's contain govdocs data on S2 (sequential read) *	Dario Gagulic
Literature Review of assigned papers	Dario Gagulic
Select two ransomwares	Dario Gagulic
WannaCry on setup2 (not sure if its valid, no encryption)	Dario Gagulic
Literature Review of assigned papers	Lynn Zumtaugwald
Meeting with Janik Proof of concept ransomware	Lynn Zumtaugwald
Select two ransomwares	Lynn Zumtaugwald
Separate code to sector csv files and save it into new csv files	Lynn Zumtaugwald
add machine and trace time to time series ppp and to trace analysis ppp	Siddhant Sahu
modification of the ENT entropy extractor	Siddhant Sahu
Apply naming convention to folders and files of traces	Siddhant Sahu
Apply romans improvement suggestions on report	Siddhant Sahu
Collect traces	Siddhant Sahu
Comparison of block devices (incl. modifying C code and plotting) *	Siddhant Sahu
LBA Time series plots	Siddhant Sahu
Literature Review of assigned papers	Siddhant Sahu
Mixed workload traces collection and analysis (lockbit + benign) S1	Siddhant Sahu

Recreation of Hirano2022 tSNE plots *	Siddhant Sahu
Renaming trace files to adhere to naming convention (date_workload_s1/s2/s3_wr/rd.csv) and create split dat file into rd and wr csv files	Siddhant Sahu
Run PCA and tSNE on our proposed features set using RanSAP; Focus on feature importance	Siddhant Sahu
Select two ransomwares	Siddhant Sahu
Setting up test environment	Siddhant Sahu
Trace Analysis script(5 metrices - number of reads IO and writes IO, entropy min and max, read rate, write rate	Siddhant Sahu
Modifying the file conversion script to prevent race conditions between the ransomware process and file conversion process trying to open the same file, for mixed workloads	Siddhant Sahu