Mechanisms for Analysis and Detection of Ransomware in Desktop Operating Systems

Vinicius Belloli Dos Santos

Thesis presented to the School of Technology and Management in the scope of the Master in Informatics.

Supervisors:
Prof. Tiago Miguel Ferreira Guimarães Pedrosa
Prof. Newton Carlos Will

Bragança

2022-2023

Mechanisms for Analysis and Detection of Ransomware in Desktop Operating Systems

Vinicius Belloli Dos Santos

Thesis presented to the School of Technology and Management in the scope of the Master in Informatics.

Supervisors:

Prof. Tiago Miguel Ferreira Guimarães Pedrosa

Prof. Newton Carlos Will

Bragança

2022-2023

Dedication

To my family.

Acknowledgment

I would like to thank the faculty members of the Polytechnic Institute of Bragança and UTFPR for their unwavering support during my academic journey. Special thanks to my supervisors Prof. Tiago Pedrosa and Prof. Newton Carlos Will, for their invaluable assistance without which my research would not have been possible. Finally, I would like to thank my family for their unwavering support.

Abstract

Ransomware attacks have become a danger to computer systems, leading to data loss, monetary losses, and business interruptions. We propose a machine learning-based method for ransomware detection on Linux to identify these attacks. To detect ransomware activity on the system, our approach combines the file system with a predictive model. To obtain sufficient infection information we use the data from the alteration calls to the files on the file system. This data is then fed into a machine-learning algorithm. Using a dataset we collected from uninfected files and files infected with various types of ransomware and were able to achieve a high detection rate with a low false positive rate. Our methodology can be incorporated into current security programs to improve detection and defense against ransomware attacks in the Linux environment.

Keywords: Ransomware; Malware; Cybersecurity; Threat detection; Linux.

Resumo

Os ataques de ransomware se tornaram um perigo para os sistemas de computador, levando à perda de dados, perdas monetárias e interrupções nos negócios. Propomos um método baseado em aprendizado de máquina para detecção de ransomware no Linux para identificar esses ataques. Para detectar a atividade de ransomware no sistema, nossa abordagem combina o sistema de arquivos com um modelo preditivo. Para obter informações suficientes sobre a infecção, usamos os dados das chamadas de alteração dos arquivos no sistema de arquivos. Esses dados são então inseridos em um algoritmo de aprendizado de máquina. Usando um conjunto de dados que coletamos de arquivos não infectados e arquivos infectados com vários tipos de ransomware, conseguimos atingir uma alta taxa de detecção com uma baixa taxa de falsos positivos. Esta metodologia pode ser incorporada nos programas de segurança atuais para melhorar a detecção e a defesa contra ataques de ransomware no ambiente Linux.

Palavras-chave: Ransomware; Malware; Cibersegurança; Detecção de ameaças; Linux.

Contents

1 Introduction **1**

1.1 Context . . . 1

1.2 Justification . . . 2

1.3 Goals . . . 2

1.4 Thesis Structure . . . 3

2 State-of-the-Art **5**

2.1 The Challenge . . . 5

2.2 Review Protocol . . . 5

2.3 Analysis Of The Selected Studies . . . 10

2.4 Cryptography . . . 20

2.5 Entropy . . . 21

2.6 Problem Statement . . . 22

3 Approach **25**

3.1 Proposed solution . . . 25

3.1.1 Detection system . . . 25

3.1.2 Analysis System . . . 26

3.1.3 System Calls . . . 27

3.1.4 Data set Generation . . . 27

3.1.5 Data set cleaning . . . 27

3.1.6 Creation of model . . . 28

4 **Implementation** 31

4.1 Development Technologies 31

4.1.1 Fanotify 31

4.1.2 Libpq 32

4.1.3 Pandas 33

4.1.4 Sci-kit-learn 34

4.1.5 Pickle 34

4.2 System Implementation 35

4.3 Database 37

4.4 Client 38

4.5 Model 41

5 **Discussion** 55

5.1 Experiments 55

5.1.1 File System Data Collection Algorithm 56

5.1.2 Machine Learning Algorithm 56

5.1.3 Detection Algorithm 57

5.2 Limitations 60

6 **Conclusions and future work** 61

6.1 Conclusions 61

6.2 Future Work 62

A **Original dissertation proposal** A1

B **Codes created for the client side** B1

B.1 Inotify first code test B1

B.2 Fanotify first code test B3

B.3 Adapted code for fanotify B16

B.4 Postgres Example Code B24

B.5 Shannon Entropy Code B28

B.6 Adapted code from fanotify with MODIFY event B31

List of Figures

3.1 Flow of the ransomware detection system 26

4.1 Virtual Machine Structure 37

4.2 The figure represents the table responsible for storing the file system logs. . 38

4.3 The figure represents the example of the stored data. 38

4.4 Client Activity Diagram 40

4.5 Model Activity Diagram 52

5.1 The figure represents the message presented to the user when the algorithm detects an anomaly. 59

List of Algorithms

1 Checks out the best intelligent model 44

2 Checks out the best intelligent model 51

Chapter 1

Introduction

This chapter is dedicated to the introduction of the work, where the scope, motivations, and objectives that justify its execution are addressed. Initially, the context of the work is described. Next, the justifications for the development of the work are presented, and at the end, the structure of the document is presented.

1.1 Context

Cybersecurity is constantly threatened by harmful applications and attacks like malware and ransomware. These can severely damage computer systems, data centers, and web and mobile applications, affecting various industries and businesses.

Ransomware programs typically prevent victims from accessing their data and can use complex encryption that only the attacker can decrypt. If the victim refuses to pay the attacker's demand, they may permanently lose their data and system. Attackers are also using advanced technologies to create new types of ransomware that are harder to detect and recover from, making it challenging for experts to combat these threats effectively. This constant evolution of ransomware also makes it difficult to identify new patterns and varieties [1].

1.2 Justification

In just the first half of 2022, there were 236.1 million ransomware attacks worldwide. Through 2021, there were 623.3 million ransomware attacks globally. This doesn't mean every attack was successful, but it does highlight the prevalence of this cyberthreat [2].

According to data uncovered by Trend Micro, the company prevented 63 billion threats in the first half of 2022. The top three industries attacked by malware were government, industry, and healthcare, with 52% more attacks recorded in the first half of the year compared to the same period in 2021. During this period, there was an increase in ransomware-as-a-service attacks. Trend Micro's data also revealed that 67 active RaaS and extortion groups, along with over 1,200 victim companies, were recorded in the first half of 2022 these attacks were recorded on Linux systems by the company [3].

These numbers support the development and focus of ransomware detection systems for Linux systems that have been a great focus of the attackers in the last year due to the increase of the system use.

1.3 Goals

The current work has the objective of developing a detection system that can identify possible files being encrypted on Linux systems. The objectives of the work can be summarized as follows:

- Develop an intelligent model capable of classifying whether a file may or may not have been encrypted.
- Creation of a scenario and methodology for creating datasets.
- Create datasets of infection to provide for future analysis.
- Test the efficiency of the model created and its limitations.

1.4 Thesis Structure

The document has been subdivided into 6 chapters. The current chapter presents the introduction and the intended objectives. The remainder of the document is organized as follows:

- Chapter 2: State-of-the-Art

  - Contains the literature review about the ransomware detection, the simulation in this field, the common characteristics, and deviations between this work and those found in literature.

- Chapter 3: Approach

  - Contains an explanation of the approach that will be used to develop the work, further explaining the division of the proposed model and how we intend to implement it.

- Chapter 4: Implementation

  - Contains an explanation of how the algorithms and intelligent models will be implemented, it also describes the process of creating the database and the tools used for the development of the project as a whole.

- Chapter 5: Experiments and discussion

  - Contains an explanation of how the experiment was executed, how the results were evaluated, and the final results, also pointing out the limitations encountered during development.

- Chapter 6: Conclusions and future work

  - Contains the conclusions that were obtained by developing the work, also pointing out the points that can be better explored in a future work

Chapter 2

State-of-the-Art

This chapter presents the main concepts that were taken into consideration during the planning, analysis, and development stages of this work. Initially, the challenge we seek to solve is presented, followed by the protocol for reviewing the bibliography, and finally an analysis of the selected articles presenting a summary of the article's approach and what it proposes to solve.

We will also present some concepts of cryptography and entropy.

2.1 The Challenge

The proposed challenge is the development of a ransomware detection system that is efficient and fast for platforms with Linux desktop and server systems, aiming to approach an area that does not have a vast approach in the literature which is able to detect when the system is suffering a cryptographic attack and can warn the user about this problem.

2.2 Review Protocol

To define the research questions we needed to define questions that would allow us to find studies that fit our study idea, thus, we decided to determine in what scenario the ransomware technology has been used, in addition to what security premises are applied.

In this context, the research questions reflect that purpose as follows:

- **RQ1:** What are the techniques used for ransomware detection ? Rationale: Here we seek to find current ransomware detection techniques.

- **RQ2:** What are the difficulties of ransomware detection? Rationale: Here we seek to find the difficulties encountered for ransomware detection implementations

- **RQ3:** What types of ransomware exist? Rationale: Here we try to filter all ransomware families that are already addressed in order to find new ones and improve existing ones.

To perform the search for primary studies, we need to define both the search string and the research repositories. The search string combines keywords and their synonyms, presented in Table 2.1, which logical operators later connect, as presented in Table 2.2.

Keyword	**Synonyms**
Detection	“Countermeasure”; “Defense” ; “Mitigation”; “Prevention”
Ransomware	“Crypto-ransomware”; “Cryptoware”; “Cryptoworm”; “Cyber ransom”; “Locker-ransomware”

Table 2.1: Keywords and Synonyms

(“Ransomware” **OR** “Crypto-ransomware” **OR** “Cryptoware” **OR** “Cryptoworm” **OR** “Cyber ransom” **OR** “Locker-ransomware”) **AND** (“Detection” **OR** “Countermeasure” **OR** “Defense” **OR** “Mitigation” **OR** “Prevention”)

Table 2.2: Search String

The search was carried out in February 2022, covering digital repositories considered as the most relevant scientific sources and, therefore, likely to contain important primary studies. Table 2.3 lists the research repositories.

Research Repository	URL
ACM Digital Library	http://portal.acm.org
IEEE Digital Library	http://ieeexplore.ieee.org
Science@Direct	http://www.sciencedirect.com
Scopus	http://www.scopus.com
Springer Link	http://link.springer.com

Table 2.3: Selected Research Repositories

Screening aims to determine which primary studies are relevant to answer research questions. In this stage, we evaluated all the recovered primary studies. To this end, we applied a set of inclusion and exclusion criteria to each work. The inclusion criteria designed and applied are:

- **IC1:** If there are papers presenting more than one study, each study will be evaluated individually
- **IC2:** If there are versions of the same paper, a summary and a complete one, the complete one must be included
- **IC3:** Papers that present mechanisms for ransomware detection or mitigation in desktop operating systems
- **IC4:** When several papers show similar studies, only the most recent is included

Exclusion criteria are important, as they allow greater precision in eliminating studies not relevant to the context of the mapping. For this reason, during the individual analysis of the studies, we discarded all those that met at least one of the following exclusion criteria:

- **EC1:** Non-English papers
- **EC2:** Papers that do not present mechanisms for ransomware detection or mitigation

- **EC3:** Papers that present detection or mitigation mechanisms for other malware than ransomware
- **EC4:** Papers that present mechanisms for ransomware detection or mitigation in servers, cloud, mobile, and other than desktop operating systems
- **EC5:** Positions papers, posters, and talks
- **EC6:** Technical reports, documents that are available in the form of abstracts or presentations and also secondary literature reviews (i.e. systematic literature reviews and mapping)
- **EC7:** There is a more complete paper published by the same authors
- **EC8:** Papers published before 2022

By applying the search string in the digital repositories, the process returned 3005 papers. However, 518 were duplicated, resulting in 2487 unique papers. After applying the inclusion and exclusion criteria, based on the title, abstract, and keywords of each work, we obtained 275 candidate papers. Finally, after reading the candidate papers' introduction and conclusion, 18 papers were selected for the study, as shown in Table 2.4.

Research Repository	**Total Papers**
ACM Digital Library	471
IEEE Digital Library	295
Science@Direct	77
Scopus	802
Springer Link	1360
Total	3005
Duplicated	518
Rejected	2212
Candidate	275
Final Selection	**18**

Table 2.4: Number of papers by each digital repository and summary of the selection process

The 18 selected studies were published in conferences, symposiums, workshops, and journals, with the majority of studies focusing on journals. Table 2.5 lists all the selected studies, which will be analyzed in the following sections.

Title	**Year**	**Cited By**	**Pub.Type**
A few-shot meta-learning based siamese neural network using entropy features for ransomware classification [4]	2022	8	Journal
A novel approach for ransomware detection based on PE header using graph embedding [5]	2022	0	Journal
A Grammar-Based Behavioral Distance Measure Between Ransomware Variants [6]	2022	1	Journal
Behavior-based ransomware classification: A particle swarm optimization wrapper-based approach for feature selection [7]	2022	0	Journal
BigRC-EML: big-data based ransomware classification using ensemble machine learning [8]	2022	1	Journal
Classification of ransomware using different types of neural networks [9]	2022	0	Journal
Comparative analysis of various machine learning algorithms for ransomware detection [10]	2022	0	Journal
Dual Generative Adversarial Networks Based Unknown Encryption Ransomware Attack Detection [11]	2022	0	Journal
FeSA: Feature selection architecture for ransomware detection under concept drift [12]	2022	2	Journal
Inhibiting crypto-ransomware on windows platforms through a honeyfile-based approach with R-Locker [13]	2022	1	Journal
On Ransomware Family Attribution Using Pre-Attack Paranoia Activities [14]	2022	3	Journal
Pre-Encryption and Identification (PEI): An Anti-crypto Ransomware Technique [15]	2022	1	Journal
Process based volatile memory forensics for ransomware detection [16]	2022	1	Journal
Ransomware Classification and Detection With Machine Learning Algorithms [1]	2022	4	Journal
Ransomware Detection Using Open-source Tools [17]	2022	0	Conference
Rcryptect: Real-time detection of cryptographic function in the user-space filesystem [18]	2022	3	Journal
Zero-day Ransomware Attack Detection using Deep Contractive Autoencoder and Voting based Ensemble Classifier [19]	2022	1	Journal
Detecting Ransomware Attacks Distribution Through Phishing URLs Using Machine Learning [20]	2022	0	Conference

Table 2.5: Selected Studies

2.3 Analysis Of The Selected Studies

Zhu, Jang-Jaccard, Singh, *et al.* [4] proposes a novel approach for detecting and classifying ransomware using a few-shot meta-learning based Siamese neural network. The authors suggest a solution to tackle the issue by employing a Siamese neural network, which uses entropy features to classify ransomware samples. This few-shot meta-learning-based network can classify new ransomware samples with minimal training data. The entropy features utilized in the network are obtained from the ransomware sample's various data segments, measuring the complexity and randomness of the data. This information can be used to differentiate between different ransomware types. The Siamese neural network comprises two identical sub-networks that simultaneously process two input samples. The output of these sub-networks is combined and passed through fully connected layers to generate a classification result. The network is trained using a loss function that ensures similar samples have similar embeddings, while dissimilar samples have different embeddings. The experimental results demonstrate that the proposed approach effectively detects and classifies different types of ransomware samples, even with limited training data. Based on this, the authors conclude that the few-shot meta-learning-based Siamese neural network using entropy features is a promising approach for detecting and classifying ransomware.

Manavi and Hamzeh [5] proposes a new approach for detecting ransomware using graph embedding techniques applied to Portable Executable (PE) headers. According to the authors, current methods of detecting ransomware using signature matching or behavior analysis are insufficient because they can be easily circumvented by malware creators. To tackle this issue, they suggest a new approach that utilizes graph embedding to depict the structural properties of PE headers as graphs, followed by machine learning algorithms to classify them as either benign or malicious. To implement this method, the PE header is transformed into a graph representation, where each node corresponds to a particular header attribute, and edges represent the relationships between them. Graph embedding techniques, such as node2vec or GraphSAGE, are then employed to embed

the graph into a low-dimensional space that captures its structural information. Finally, a machine learning algorithm is trained on the embedded graph to categorize the PE header as ransomware or benign. The authors assessed their approach on a dataset comprising 10,000 PE files, encompassing both benign software and ransomware. Their results demonstrate that the proposed technique exhibits high precision, recall, and accuracy in identifying ransomware, surpassing traditional signature-based and behavior-based methods. Overall, the article proposes a promising strategy for identifying ransomware that harnesses the potential of graph embedding techniques to capture the structural characteristics of PE headers. This technique has the potential to enhance the efficacy of ransomware detection and improve the security of computer systems.

Parunak [6] proposes a new method for measuring the similarity between different variants of ransomware. A new technique for measuring the similarity between various ransomware variants based on their behavior is proposed by the authors of the article. The technique, called "grammar-based distance", involves using formal grammar to represent the behavior of each variant as a set of rules, and then calculating the number of different rules between two variants to determine their distance. To validate their approach, the authors tested their method on a dataset containing 48 different ransomware variants, comparing the results to other techniques such as static and dynamic analysis. The authors found that their grammar-based distance method could accurately distinguish between ransomware variants, even when the variants were very similar in behavior. The authors believe that their approach could assist in the detection and analysis of new ransomware variants. Comparing the behavior of new variants to known variants using this method could enable security researchers to quickly identify new threats and develop strategies to defend against them. The technique could also aid in tracking the evolution of ransomware over time and identifying patterns in the behavior of various variants.

Abbasi, Al-Sahaf, Mansoori, *et al.* [7] presents a novel approach to classifying ransomware based on its behavior using a particle swarm optimization (PSO) wrapper-based feature selection method. The article proposes utilizing PSO, a swarm intelligence optimization technique, to select the most relevant behavioral features from a large set of

ransomware samples. These selected features are then utilized to train a machine learning classifier for distinguishing between legitimate software and ransomware. To test the effectiveness of this approach, the authors conducted experiments using a dataset of 1,128 ransomware and benign samples each. The results demonstrate that the proposed approach achieves high accuracy, precision, recall, and F1-score when compared to other state-of-the-art ransomware classification methods. Moreover, this approach can efficiently classify ransomware samples in real time with minimal computational requirements. In summary, the article offers a promising solution for detecting ransomware based on behavior and validates the efficacy of swarm intelligence optimization techniques for feature selection in machine learning.

Aurangzeb, Anwar, Naeem, *et al.* [8] presents a method for classifying ransomware using ensemble machine learning algorithms. The goal of the method is to improve the detection accuracy of ransomware by utilizing big data and advanced machine learning techniques. The authors have proposed a solution to the problem at hand, which they refer to as BigRC-EML, or Big Ransomware Classification using Ensemble Machine Learning. Their method involves gathering a large dataset comprising ransomware and non-ransomware samples, which is then used to train several machine learning algorithms. To implement their ensemble approach, the authors use five different machine learning algorithms: Random Forest, Logistic Regression, Naive Bayes, Decision Tree, and Support Vector Machine. Each of these algorithms is trained on a different subset of the dataset, and their results are combined using a weighted voting mechanism. The performance of the method is evaluated using a dataset of ransomware and non-ransomware samples, which results in an accuracy of 97.5%. This is significantly higher than the accuracy of individual machine-learning algorithms. Overall, the article proposes a promising solution to the problem of ransomware detection, leveraging the power of big data and ensemble machine learning. It indicates that by combining the strengths of multiple machine learning algorithms, it is possible to achieve high accuracy in identifying ransomware.

Madani, Ouerdi, Boumesaoud, *et al.* [9] discusses how neural networks can be used

to classify different types of ransomware. The authors explain how ransomware classification can be accomplished using neural networks that analyze features such as file size, entropy, and opcode frequency. They compare the performance of different types of neural networks, such as MLPs, CNNs, and LSTM networks. The authors test their method using a dataset containing more than 9,000 ransomware samples and achieve classification accuracies of up to 98% for some neural network models. They also demonstrate the ability of their method to detect new and previously unseen ransomware variants. In summary, the article presents a comprehensive overview of the challenges involved in classifying ransomware and the potential of neural networks to tackle these challenges. It emphasizes the need for more advanced techniques to detect and prevent ransomware attacks and suggests that neural networks could be an important tool in this endeavor.

Khammas [10] is a research paper that explores the effectiveness of different machine learning algorithms in detecting ransomware attacks. The authors explain their approach to comparing various machine learning algorithms' performance. They utilized a dataset consisting of real-world ransomware examples and trained and tested multiple algorithms, including decision tree, k-nearest neighbors, support vector machines, and random forest. They assessed each algorithm's effectiveness by analyzing metrics such as precision, recall, accuracy, and F1 score. According to the study's outcomes, the random forest algorithm achieved the highest accuracy rate of 99.71% for detecting ransomware. Additionally, the k-nearest neighbors and decision tree algorithms displayed commendable results, with accuracies of 98.87% and 97.14%, respectively. The authors suggest that these findings demonstrate the vast potential of machine learning algorithms in improving ransomware detection and recommend further investigation to enhance these algorithms' performance. Overall, this article presents valuable insights into the efficacy of different machine-learning algorithms in identifying ransomware attacks. The research's conclusions can aid in developing more robust and efficient ransomware detection tools, safeguarding individuals and organizations from this escalating threat.

Zhang, Wang, and Zhu [11] proposes a method for detecting ransomware attacks on

computer systems. A new method for detecting ransomware using dual generative adversarial networks (GANs) is proposed in the article. GANs are machine learning models that can create realistic synthetic data by training two neural networks against each other. The GANs in this case produce synthetic samples of encrypted and unencrypted files, which are then used to train a binary classification model. The goal of the classification model is to differentiate between encrypted and unencrypted files and to identify unknown encryption algorithms, a critical task because custom encryption algorithms are often used in ransomware attacks and are not known to security researchers. The article provides experimental results showing that the proposed approach is highly effective. The method achieved high accuracy in detecting both known and unknown ransomware attacks, outperforming several baseline methods. In summary, the article introduces a novel approach to ransomware detection utilizing dual GANs and binary classification. The method aims to identify unknown encryption algorithms, which is a significant challenge in ransomware detection. The experimental results demonstrate that the proposed method is effective and has the potential to enhance the state of the art in ransomware detection.

Fernando and Komninos [12] proposes a novel approach for detecting ransomware attacks in real-time. The proposed method for detecting ransomware attacks involves using machine learning algorithms to analyze network traffic data. However, a key obstacle in this approach is that the characteristics of ransomware attacks can change over time, leading to reduced detection accuracy, known as concept drift. To tackle this problem, the authors suggest using a feature selection architecture (FeSA) that can dynamically identify relevant features for detecting ransomware attacks in real time. FeSA comprises a feature selection module and a classification module, which employ a genetic algorithm and an ensemble of machine learning classifiers, respectively. The authors evaluated FeSA using real-world ransomware datasets and compared it to other state-of-the-art techniques. The results indicate that FeSA can achieve high detection accuracy while maintaining low false positives, even under concept drift. This innovative approach has practical applications in real-world cybersecurity systems for detecting ransomware attacks. In summary, the article presents a novel and effective method for detecting ransomware attacks that address

the challenge of concept drift, which is crucial in developing reliable cybersecurity systems.

Gómez-Hernández, Sánchez-Fernández, and García-Teodoro [13] proposes a new approach for preventing and mitigating the impact of crypto-ransomware attacks on Windows systems. The proposed strategy employs honey files, which are dummy files that masquerade as authentic but are specifically designed to activate an alert when an unauthorized user or program accesses or alters them. To combat crypto-ransomware attacks, the authors created R-Locker, a tool that utilizes honey files. R-Locker generates numerous honey files throughout the system and keeps an eye on them for any changes. If a honey file is altered or encrypted, R-Locker will instantly recognize the action and shut down the system to avoid further harm. The tool can also issue notifications to alert administrators of the attack and furnish information on the impacted files and the kind of ransomware utilized. The authors tested R-Locker using various actual ransomware samples, such as WannaCry and Petya and discovered that the tool effectively detected and prevented all attacks. Additionally, they compared R-Locker's performance to other popular anti-ransomware tools and found that it outperformed them in terms of both detection and false positive rates. Overall, the paper demonstrates the potency of honey file-based approaches in preventing and mitigating the effects of crypto-ransomware attacks on Windows systems. Honeyfiles offer a proactive defense method that can identify and prevent attacks before they result in significant harm, and R-Locker is a practical tool that organizations can utilize to enhance their ransomware defense capabilities.

Molina, Torabi, Sarieddine, *et al.* [14] proposes a new approach for attributing ransomware attacks to specific malware families by analyzing their pre-attack activities, specifically the steps they take to identify and evade detection by security systems. The authors point out that current methods for tracing ransomware generally rely on analyzing the attack after it has occurred, which can make it difficult to accurately determine the perpetrator. In contrast, their approach focuses on detecting patterns in the pre-attack behaviors of different ransomware families, such as vulnerability scanning, network connectivity testing, and avoiding detection. To demonstrate the effectiveness of their method, the authors analyze three ransomware families - Locky, Cerber, and Jigsaw - and

identify specific patterns in their pre-attack activities. These patterns are then used to create a set of attribution rules. According to the authors, their method can be helpful to incident responders and security professionals in quickly identifying the specific ransomware family behind an attack, thereby allowing them to develop effective mitigation strategies. They also suggest that this approach could be expanded to other types of malware and used alongside other attribution techniques to enhance the accuracy of tracing cyber-attacks.

Mantri, Singh, Kumar, *et al.* [15] proposes a new approach to combating ransomware attacks. Ransomware attacks are a type of malware that encrypts a victim's files, rendering them inaccessible until a ransom is paid. The proposed technique aims to prevent ransomware from encrypting files by encrypting them before the ransomware can. The proposed method by the authors is called Pre-Encryption and Identification (PEI), which entails encrypting files on a victim's computer before any ransomware can access them. A program operates in the background of the computer to encrypt files as they are created or modified, storing them in a secure location that is inaccessible to ransomware. The PEI technique also includes an identification component that detects when ransomware tries to encrypt a file. This component compares the encrypted files stored in the secure location with the files accessed by the ransomware. If the ransomware tries to encrypt a file already encrypted by the PEI program, the identification component prevents the ransomware from doing so. According to the authors, the PEI technique offers several advantages over other ransomware protection methods. For instance, it does not rely on signature-based detection, which enables it to detect new strains of ransomware that have not been seen before. Moreover, since the PEI program encrypts files before any ransomware can access them, the technique is effective even against ransomware that uses zero-day exploits. In summary, "Pre-Encryption and Identification (PEI): An Anticrypto Ransomware Technique" presents a novel approach to fighting ransomware by encrypting files before they can be accessed by ransomware. Although the technique is not yet widely adopted, it has the potential to be an efficient tool in the battle against ransomware.

Arfeen, Khan, Zafar, *et al.* [16] proposes a technique for detecting ransomware attacks

using memory forensics. The proposed method by the authors detects ransomware by examining the behavior of processes in volatile memory, which is the type of memory that disappears when a computer shuts down. This approach can identify ransomware activity by observing the creation of new processes, file modifications, or data encryption. To implement this technique, the authors utilized the Volatility Framework, a memory forensics tool that can extract process information from memory dumps and analyze running processes. The authors identified ransomware-specific behaviors, such as encryption algorithms and file extension modifications, and successfully tested their technique on various ransomware samples. Furthermore, the authors emphasized the potential of their method to detect other types of malware, not only ransomware. In conclusion, the article highlights an innovative approach to detecting ransomware using memory forensics and provides security professionals and researchers with useful tools to combat cyber threats.

Masum, Faruk, Shahriar, *et al.* [1] focuses on the development of a system for the classification and detection of ransomware attacks using machine learning algorithms. The article details a system that combines network traffic and system call features to identify and classify ransomware attacks. The system was tested using actual ransomware attack data and achieved a high detection rate while minimizing false positives. Additionally, the article examines several machine learning algorithms used in the system, such as Decision Trees, Random Forest, Naive Bayes, and Support Vector Machines (SVMs). After comparing their performance, the authors conclude that SVMs are the most effective in detecting ransomware attacks. Overall, the article offers valuable insights into the creation of a machine learning-based ransomware detection and classification system, which has the potential to enhance the security of computer networks and systems.

Lee, Shim, Lee, *et al.* [17] discusses the use of open-source tools for detecting ransomware attacks. In the article, open-source tools for detecting ransomware are discussed, including Sysmon, YARA, Snort, and OSSEC, which were previously mentioned in response. Each tool is described in detail, with an explanation of how it can be utilized to detect ransomware activity. The article then introduces a proposed ransomware detection system that integrates these open-source tools. The authors provide a comprehensive

guide on how to set up the system and describe how it operates. Lastly, the article reports on the results of experiments conducted to assess the efficacy of the proposed ransomware detection system. The authors demonstrate that the system successfully detected various types of ransomware attacks with a high degree of accuracy. Overall, the article serves as a valuable resource for organizations interested in implementing an open-source ransomware detection system. It emphasizes the importance of using multiple tools in combination and showcases the effectiveness of such an approach in detecting ransomware attacks.

Lee, Jho, Chung, *et al.* [18] present a novel method, Rcryptect (Real time + Crypto + Detect) to detect potentially malicious cryptographic functions at run-time in the filesystem. The article introduces Rcryptect, a novel tool designed to detect and monitor the real-time usage of cryptographic functions in a user-space filesystem. The authors assert that detecting such functions is a critical aspect of identifying and mitigating attacks on sensitive information like financial data or passwords. Rcryptect operates as a kernel module and utilizes a combination of heuristics and pattern-matching algorithms to pinpoint cryptographic functions used by user-level processes. The authors evaluated Rcryptect's performance by testing it on various real-world applications, including password managers and file encryption tools. The results of the evaluation indicate that Rcryptect has a high detection rate and can accurately identify cryptographic functions in multiple applications. Additionally, the authors compared Rcryptect's performance with other similar tools and found that it surpasses them in terms of detection accuracy and speed. In summary, the authors concluded that Rcryptect is a valuable tool for the real-time detection and monitoring of cryptographic functions. It can play a crucial role in safeguarding sensitive information and identifying potential security threats.

Zahoora, Rajarajan, Pan, *et al.* [19] proposes a novel approach for detecting ransomware attacks using a combination of deep learning techniques and ensemble classifiers. The suggested method involves using a deep contractive autoencoder (DCA) to learn a compressed representation of input data, enabling the identification of subtle changes in file behavior induced by ransomware. To learn the underlying structure of each file type,

the DCA is trained on a large set of benign and malicious files. Following DCA training, the compressed representation is inputted into an ensemble classifier, which combines the outputs of several classifiers to produce a final prediction. The ensemble classifier utilizes voting to aggregate the outputs of multiple classifiers that employ different machine learning algorithms, thereby mitigating the risk of false positives or negatives. The proposed approach was tested on a real-world dataset of ransomware attacks and achieved high accuracy, precision, and recall rates in identifying zero-day ransomware attacks. The authors suggest that this approach could be combined with antivirus software and other security measures to provide an extra layer of protection against ransomware attacks. In summary, the article presents a promising strategy for detecting zero-day ransomware attacks using deep learning and ensemble classifiers, which could enhance computer system security.

Chaithanya and Brahmananda [20] is a research paper that explores the use of machine learning algorithms to identify and prevent ransomware attacks that are distributed through phishing URLs. The authors then introduce their proposed solution, which involves using machine learning algorithms to analyze the URLs contained in phishing emails and determine whether they are likely to lead to a ransomware attack. The algorithm is trained using a dataset of known phishing URLs and their associated ransomware payloads. The paper describes the steps involved in the machine learning process, including feature extraction, model selection, and performance evaluation. The authors also discuss the various metrics used to evaluate the effectiveness of the model, such as precision, recall, and F1 score. To test the effectiveness of their approach, the authors conducted experiments using real-world phishing emails and URLs. The results showed that their machine learning model was able to accurately identify phishing URLs that were likely to lead to a ransomware attack, with an accuracy of over 98%.The authors conclude that their approach shows promise in detecting and preventing ransomware attacks that are distributed through phishing emails. They suggest that their model could be integrated into existing security systems to provide an additional layer of protection against ransomware attacks.

2.4 Cryptography

When we talk about ransomware it is important to understand how this type of attack works, so understanding the concept of encryption is extremely important for a better analysis of the attacks.

Encryption is the process of converting original messages, also known as plaintext, into an unreadable format called ciphertext by using an encryption algorithm and a key. This makes the data incomprehensible to unauthorized parties, thereby safeguarding the confidentiality and integrity of the information that is being stored or transmitted. Encryption can be employed for different types of data, including text, images, videos, and files, and can be implemented in diverse applications like email, secure messaging platforms, and online transactions. Symmetric encryption, also called secret-key encryption, employs the same key for both encryption and decryption. The sender and receiver need to have identical keys to decrypt the message. Advanced Encryption Standard (AES) and Data Encryption Standard (DES) are examples of symmetric encryption algorithms. Asymmetric encryption, also known as public-key encryption, employs two distinct keys for encryption and decryption. The public key is utilized for encryption, whereas the private key is used for decryption. The sender encrypts the message using the receiver's public key, and the receiver decrypts the message using their private key. Examples of asymmetric encryption algorithms are RSA and Elliptic Curve Cryptography (ECC). Encryption is a crucial aspect of modern-day cybersecurity that aids in safeguarding sensitive data from unauthorized access, theft, and cyber attacks. However, it is not entirely foolproof, and attackers can still exploit vulnerabilities like weak encryption algorithms and key management issues. Therefore, encryption should be utilized in conjunction with other cybersecurity measures to ensure data security [21].

Cryptography is a common technique used by malware authors to protect their code from detection and enable communication with command and control servers. There are several cryptographic techniques used by malware, including encryption, steganography, hashing, digital signatures, keylogging, and rootkits. Encryption involves encrypting the

malicious code or communication data to evade detection by security software. This encrypted data is decrypted at runtime using a decryption key. Popular encryption algorithms used by malware include Advanced Encryption Standard (AES), Blowfish, and RSA. Steganography is a technique used by malware authors to hide malicious code within seemingly innocent files or images. The code is extracted at runtime using a decryption key. Hashing is used by malware authors to generate unique values for the malicious code, which can be used to verify the integrity of the code and to check for updates. Hashing algorithms such as MD5 or SHA-1 are commonly used for this purpose. Digital signatures are used to make malware appear legitimate and evade detection. The digital signature is created using a private key and can be verified using a public key. Keylogging involves capturing sensitive information such as usernames and passwords. The keylogger records keystrokes and sends them to a remote server using encryption. Rootkits are a type of malware that hides its presence on the system by modifying the operating system or kernel. Rootkits can intercept system calls and modify system files to evade detection by security software [22].

2.5 Entropy

Entropy is a measure of uncertainty or randomness in a message or data source in information theory. Claude Shannon first introduced it in 1948 to quantify the information content of messages and assess communication limits over noisy channels. Entropy is computed as the average amount of information required to encode a message in bits. The greater the uncertainty or unpredictability of the message, the higher its entropy. A message with one possible outcome, like a coin flip, has lower entropy than a message with many possible outcomes, like rolling a die. Entropy also measures the compression potential of a data source. High entropy indicates the presence of redundancy or patterns that can be exploited to compress data without losing information. Conversely, low entropy means the data is already highly compressed, and further compression may lead to

information loss. Entropy has numerous applications in information theory, computer science, and cryptography. It is employed in lossless data compression algorithms, including Huffman coding and arithmetic coding, and in encryption algorithms like the Advanced Encryption Standard (AES) [23].

In the encryption process of ransomware attacks, entropy plays a critical role. The encryption process typically involves the creation of a unique key to encrypt the victim's files. To enhance the security of the encryption, ransomware attackers often utilize high-entropy keys. These keys are generated using a random number generator with a high level of entropy, making it extremely challenging for the victim to recover their files without paying the ransom. The key is virtually impossible to guess or brute force. In summary, entropy plays a vital role in the effectiveness of ransomware attacks. Using high-entropy keys can improve the security of the encryption and make it more challenging to crack, but it can also make it more challenging for the attacker to decrypt the files [24].

2.6 Problem Statement

Ransomware attacks have become more prevalent in recent years in linux, posing a significant threat to both individuals and organizations. This type of malicious software encrypts files or systems and demands payment for a decryption key. Such attacks can cause financial losses, data breaches, and operational disruptions. Unfortunately, traditional antivirus and anti-malware software are not always effective in detecting and preventing ransomware attacks. Attackers employ sophisticated techniques to evade detection. Therefore, there is an urgent need for an advanced ransomware detection system that can identify and mitigate these threats in real time. This project aims to design and develop a ransomware detection system that employs advanced algorithms and machine learning techniques to detect ransomware attacks in real time. The system will be able to monitor the file system, identify suspicious activity, and alert system administrators of potential threats. By successfully implementing a ransomware detection system, individuals and organizations will be able to safeguard their valuable data and assets from

malicious cyber-attacks.

Chapter 3

Approach

This chapter will manifest the approach adopted to solve the original problem, presenting the system architecture and explaining how it will operate.

3.1 Proposed solution

The proposed system aims to develop an efficient ransomware detection system for servers and desktops with a Linux system, using entropy and file size to check if a file may have suffered a malicious change. The focus of the proposal is the development of an intelligent model that will be trained to identify these changes and will warn the user when a possible attack is occurring. An additional focus of the work is the availability of the collected data to provide a robust infection dataset for new researchers who wish to implement something in the area.

3.1.1 Detection system

The proposed system will implement ransomware detection at the system level using file entropy as a comparison. In the proposed work we will analyze the file system calls to check which files are trying to be accessed so that we can later compare the entropy level of the previous file with the current one and if it is a higher value than expected we will

warn the user about a possible encryption of his data. We will look at the best approach to work with system calls in a dedicated section, to complement the file system monitoring model, an algorithm will be developed in python that will use models that will be trained to check file values to determine if a file may have been encrypted or not, this model will be trained using 8 types of ransomware and its training will be better explained in a dedicated section. The detection system will be implemented in python while the file system data collection system will be implemented using C.

As can be seen in Figure 3.1 the detection system is divided into two parts, one responsible for performing the data collection from the file system of the files that are being changed in real-time and storing the results in the database, the second part is a detection model implemented in python that will use the data collected in real-time to determine whether or not a file may have been encrypted and if the algorithm considers that the file may have been encrypted an alert will be presented to the user warning this.

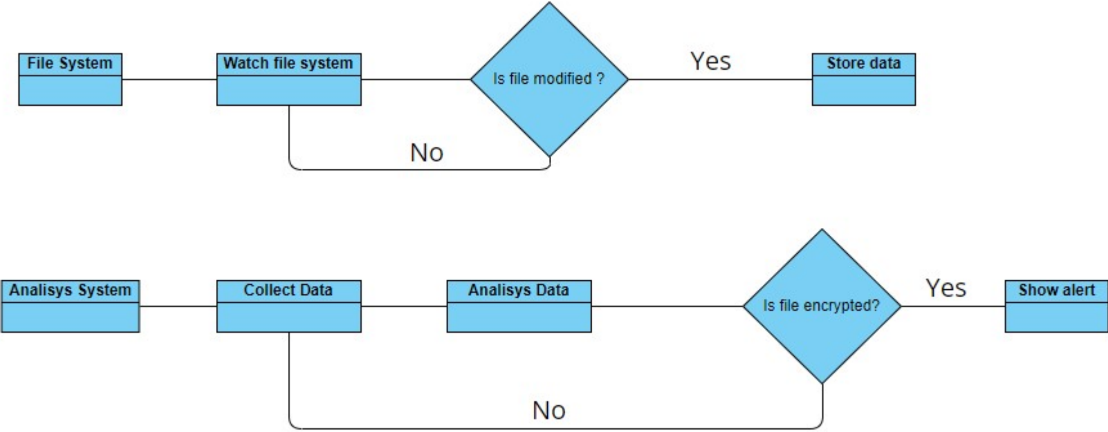

Figure 3.1: Flow of the ransomware detection system

3.1.2 Analysis System

After verifying that we would collect the data and save it in a database, it was important to have a way to verify this data and be able to correlate them to create an intelligent

detection model that would be able to verify if a file could be encrypted or not based on some values that we will collect in the system.

The analysis system will be implemented using python language, and test models will be created based on the available ransomware from which we collected infection data based on a balanced model, we will check some machine learning algorithms that best fit our data and choose the best one for the creation of the models, we will also explain the feature selection method for training.

3.1.3 System Calls

After defining how the detection problem was going to be approached, we needed method that could collect file access logs so that we could analyze the logs and define the best approach to use in evaluating the entropy of the file. We then decided to analyze two libraries better known in the Linux community which are inotify and fanotify aiming also to get more support and more examples for our work.

3.1.4 Data set Generation

With a focus on providing data from experiments to the scientific community, we also want to provide our datasets of data that we will collect for everyone to access, filling in some of the gaps in obtaining test data that are always difficult to obtain and in an acceptable quantity.

To have an initial base of files, instead of creating a new base, we took advantage of a well-known and robust base in the scientific environment, NapierOne[25], using the NapierOne-small base that contains several files, we used 9384 files for our tests.

3.1.5 Data set cleaning

According to Kirsten Barkved [26] dataset cleaning is a critical stage in creating dependable and precise AI models. The following are some reasons why dataset cleaning is crucial:

- Enhances model accuracy: The precision of an AI model heavily relies on the quality of the data it learns from. By cleaning the dataset, you can eliminate inaccuracies, errors, and noise that could negatively affect the model's performance. This results in a more precise and dependable model.

- Reduces bias: Data bias can result in partial AI model predictions. Dataset cleaning aids in detecting and eliminating biased data, minimizing the possibility of biased results.

- Saves time and resources: Cleaning the dataset can help you identify and eliminate irrelevant or duplicated data, decreasing the time and computational resources needed to train the model. This can hasten the development process and reduce expenses.

- Improves data comprehension: Dataset cleaning frequently involves in-depth exploration and analysis of the data. This can assist you in better understanding the data, recognizing patterns, and gaining insights that can be used to develop more efficient AI models.

In summary, dataset cleaning is a critical step in creating dependable, accurate, and unbiased AI models. It aids in enhancing model accuracy, decreasing bias, conserving time and resources, and improving data comprehension.

3.1.6 Creation of model

According to Goodfellow, Bengio, and Courville [27] AI-based intelligent models are developed through the training of machine learning algorithms, which enable the recognition of patterns within data, facilitating accurate predictions and decisions. These models are widely utilized in diverse applications, such as image and speech recognition, natural language processing, predictive analytics, and autonomous systems. Supervised learning and unsupervised learning are the two primary approaches employed in creating intelligent models. Supervised learning entails training the algorithm with labeled data, where

input and corresponding output data are presented to the algorithm for mapping. Unsupervised learning involves training the algorithm with unlabeled data, where it has to identify patterns and structures independently. Deep learning, a subset of machine learning, is an especially powerful technique for creating intelligent models. It employs artificial neural networks to learn from data and make predictions or decisions. These networks are composed of numerous layers of interconnected nodes, designed to recognize progressively complex features in the input data. There are numerous tools and frameworks accessible for creating intelligent models using AI, such as TensorFlow [28], PyTorch [29], and Keras [30]. These tools provide developers with a broad selection of pre-built models and components that can be customized for specific applications. When creating intelligent models, ethical considerations such as bias and privacy must be taken into account. AI models are only as good as the data they are trained on, and if the data is biased or incomplete, the model will reflect those biases. Additionally, AI models may collect and process sensitive information, so it's essential to ensure that privacy and security measures are in place.

In this work the machine learning algorithm is used in the development of a classification model that will be trained using collected data and will evaluate based on some values of entropy and file size if a file may or may not have been encrypted, the algorithm will be trained using different data sets so that there are several models to be used in the final evaluation of the result.

Chapter 4

Implementation

In this chapter the technologies used for the development of the ransomware detection system are presented, also containing the details of the developed system and the processes related to the development of the solution

4.1 Development Technologies

In this section, the technologies used to realize the implementation of the project are discussed. It is discussed the Fanotify library used to collect logs from the file system, the libpq library used for connections with the Postgres database, it is also discussed the technologies used for creating the models, pandas for loading and processing the data, sci-kit-learn for using data evaluation such as Adaboost, Random Forest and others and Pickle for generating the models and using the generated models in the prediction of new results

4.1.1 Fanotify

The fanotify library [31] is a Linux kernel feature enabling applications to track changes in the filesystem. It provides notifications to applications when files or directories are accessed, modified, or deleted. This feature is accessible from the Linux kernel version

2.6.37 onwards. Through fanotify, applications can register for events related to a specific set of files or directories, including file access, modification, opening, closing, and deletion. The fanotify library allows for customized filtering of events based on criteria such as the operation's type or the process that is performing it. Fanotify operates using a specialized kernel module that intercepts filesystem events and notifies registered applications. When an event occurs, the kernel module sends a notification to the fanotify library, which then calls a user-defined callback function in the application. One of the main advantages of fanotify is its ability to monitor filesystem events without frequent polling. This results in better performance and resource utilization. The fanotify feature can be utilized for various purposes, such as intrusion detection, file system auditing, and malware detection. It should be noted, however, that fanotify requires special permissions to access filesystem events and can negatively affect system performance if not used prudently. Furthermore, fanotify is specific to Linux and may not be available on other operating systems.

The fanotify library is used in our implementation to monitor and collect file changes on the Linux file system.

4.1.2 Libpq

Libpq [32] is a C library that facilitates connection to and communication with a PostgreSQL database. It features a user-friendly API for executing SQL queries and managing database connections. The library is bundled with the PostgreSQL database distribution and is compatible with multiple operating systems, including Linux, macOS, and Windows. Libpq is versatile and supports both synchronous and asynchronous communication with the PostgreSQL server. It can be employed in applications written in an array of programming languages, such as C, C++, Python, Ruby, and Perl, among others. Noteworthy aspects of the libpq library include SSL encryption for secure communication with the database, prepared statements for query optimization, connection pooling for scalability and reduced overhead, and batch processing of queries to enhance performance. Additionally, libpq supports asynchronous communication for improved responsiveness

and scalability, making it a popular choice for a variety of applications, from small-scale command-line utilities to large web-based applications.

The libpq library is used in our implementation to connect to the Postgresql database and with this connection, it is possible to insert the records that are collected in real-time by the fanotify library.

4.1.3 Pandas

The popular open-source Python library called pandas [33] is frequently used for analyzing and manipulating data. Its capabilities include high-performance data structures and tools designed for working with various types of structured data such as tabular, time series, and heterogeneous data. The pandas library features several important components, such as the DataFrame, which is a two-dimensional table that can store different types of data in columns. DataFrames are versatile and can perform numerous operations on data. Additionally, the Series is a one-dimensional labeled array that can store various types of data and functions as a single column in a DataFrame. Pandas provide a wide range of functions to manipulate data, such as merging, joining, filtering, aggregating, and transforming data. It also offers robust tools for working with time series data, including the ability to resample data, handle missing values, and perform rolling window calculations. Finally, the pandas library offers input/output tools, including functions that allow for reading and writing data in different formats, such as CSV, Excel, SQL databases, and HDF5. In summary, pandas is a flexible and potent tool for structured data analysis, widely used in data science, and machine learning, and an essential component for any Python programmer dealing with data.

The pandas library is used in our implementation to help load the data that is used for training and also the execution of the learning process of our model.

4.1.4 Sci-kit-learn

Scikit-learn [34] is a Python-based open-source library for machine learning that offers efficient and user-friendly tools for data mining and analysis. It is constructed on top of NumPy, SciPy, and matplotlib, and is compatible with various other Python libraries. The library provides an extensive range of both supervised and unsupervised learning algorithms, such as logistic regression, decision trees, random forests, support vector machines, linear regression, polynomial regression, K-means clustering, hierarchical clustering, DBSCAN, principal component analysis, and manifold learning. It also has data preprocessing utilities, including feature extraction, feature selection, and data normalization. Scikit-learn's key strength is its straightforward API, enabling users to train and evaluate machine learning models with minimal coding. Additionally, the library offers comprehensive documentation and examples to assist users in quickly getting started. Overall, sci-kit-learn is a flexible and potent machine learning library widely employed in both academia and industry.

The Scikit-learn library is used in our implementation to load the machine learning algorithms that are used for training, in our study we use the library's Adaboost algorithm.

4.1.5 Pickle

Pickle [35] is a Python module that enables object serialization and deserialization. It converts a Python object hierarchy into a byte stream that can be saved, transferred, or stored in a database. This process is called pickling, while the opposite process of converting a pickled byte stream into a Python object hierarchy is called unpickling. Being part of the Python standard library, Pickle comes pre-installed and doesn't require any additional installation or configuration. One of its primary advantages is that it can handle complex Python objects, such as lists, tuples, dictionaries, classes, and instances, as well as object references and object graphs. Thus, it can maintain the relationships between objects even after pickling and unpickling. However, Pickle is not secure against maliciously constructed data and should not be used to deserialize data from untrusted

sources or protocols. Furthermore, the Pickle protocol may change across different Python versions, resulting in compatibility issues when using pickled data across various Python versions. Overall, Pickle is a useful library for saving and transmitting complex Python objects, but it should be used with caution and only in secure environments.

The Pickle library is used in our implementation to export the models that are trained in our algorithm and then import them and make predictions based on these models created.

4.2 System Implementation

After the literature review, it was observed a consensus among authors that the most common method to identify ransomware is to survey its actions by static analysis, dynamic analysis, or a combination of both.

By analogy, given the specificity of this study is based on the use of the file system, it was considered important to verify the normal behavior of the system accessing files and also its base values such as entropy, number of bytes of the file, and file magic byte, we also evaluated the accesses of the files that were being infected by collecting their entropy, number of bytes of the file, all this, in the end, provided us with a large database for analysis and training of the intelligent model.

For this purpose, it was suggested by the supervisor to use file system notification interfaces, in order to detect activity. In parallel, a Linux Virtual Machine (VM) was created and some code examples of the inotify https://man7.org/linux/man-pages/man7/inotify.7.html and fanotify https://man7.org/linux/man-pages/man7/fanotify.7.html activity detectors were tested.

Next, an excerpt of the code was assembled that would allow us to: extract as much information as possible about the events that occur in the filesystem and that this collected information could be stored in a small database, and thus be able to answer part of the problem that was intended to be solved.

During the analysis we raised some possibilities for storing the access logs to the file

system, in the end, we opted for storing them securely in a Postgresql database.

For tests two VM were used, one with the filesystem checking algorithm and the second VM only with the database instance so that it would be safe during the infection experiments.

Tables 4.1, 4.2 and 4.3 show the configurations of the original machine and the VM settings.

Processor	Intel(R) Core(TM) i7-11800H @ 2.30GHz, 2400 Mhz, 8 Cores, 16 Logical Processors
RAM	32GB
Storage device	1TB SSD M2
Virtualization software	VMware Workstation Pro 17.0.0

Table 4.1: Personal PC Settings

Processor	2 processores with 2 cores each
RAM	4GB
Storage device	150 GB (preallocated)
Network adapter	NAT
Operating system	Ubuntu 22.04.1 LTS.

Table 4.2: Client VM Settings

Processor	1 processores with 1 cores each
RAM	2GB
Storage device	30 GB (preallocated)
Network adapter	NAT
Operating system	Debian Server 11 LTS.

Table 4.3: Server VM Settings

Figure 4.1 represents the Virtual Machine Structure.

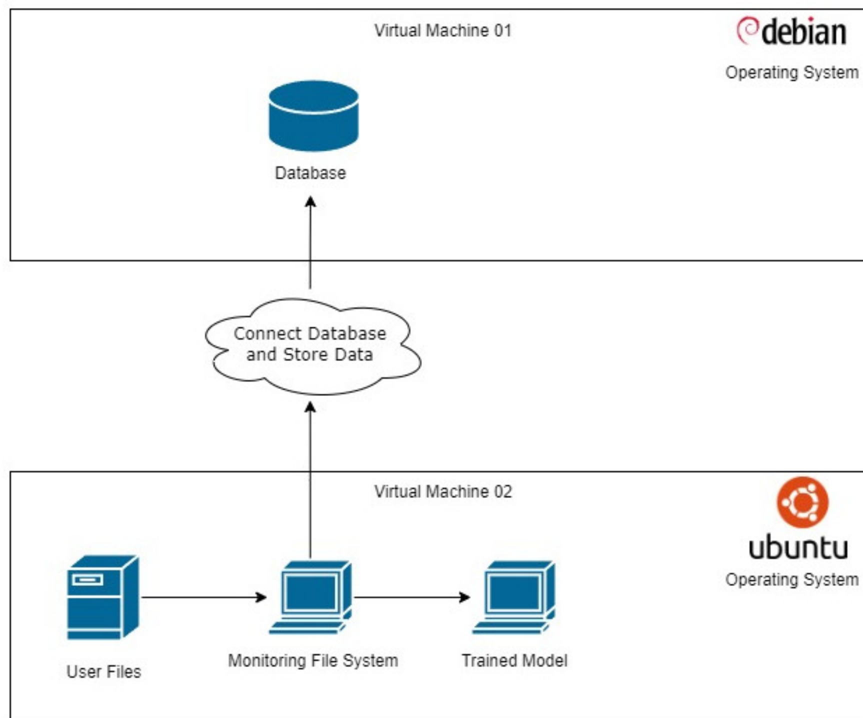

Figure 4.1: Virtual Machine Structure

4.3 Database

The database has the function of registering and storing information about the processes that occurred at the clients that will serve for future analysis and statistics.

Figure 4.2 shows the information on all the processes that occurred in the system registered by the database that was created referring to the code used.

As can be seen in the image of Figure 4.2, the database created for the code used contains some information about the processes that occurred, such as file descriptor (fd), the event number (mask), the event name (maskName), the process number (pid), the process name (pidName), the file path (file), and the identification of the hour, minutes, seconds, and the date, day of the week, day, month, and year in which the event occurred (time). This information refers to the processes that occur with the client part.

The following Figure 4.3 shows, for better clarification, a printout for a specific process about how this information is registered.

public
logs
id uuid
eventsize bigint
version bigint
reserved bigint
metadatasize bigint
fd bigint
mask bigint
maskname character varyi ng(250)
pid bigint
pidname character varying (250)
file character varying(1000)
timerec timestamp with tim e zone
entropy double precision
filebytes double precision
magicbytes bytea

Figure 4.2: The figure represents the table responsible for storing the file system logs.

	id uuid	eventsize bigint	version bigint	reserved bigint	metadatasize bigint	fd bigint	mask bigint	maskname character varying	pid bigint	pidname character varying	file character varying (1000)	timerec timestamp with time zone	entropy double precision	filebytes double precision	magicbytes bytea
1	bbee9101...	24	3	0	24	5	32	OPEN	8275	example7.exe	/home/ubuntu/Desktop/teste/files/files/0085-doc-nomagic.doc	2023-01-30 11:16:08.947493+00	4.794089	65536	[binary data]
2	d5623ac8...	24	3	0	24	5	32	OPEN	8275	example7.exe	/home/ubuntu/Desktop/teste/files/files/0017-gif.gif	2023-01-30 11:16:10.947578+00	4.423542	65536	[binary data]
3	0b43eaab...	24	3	0	24	5	32	OPEN	8275	example7.exe	/home/ubuntu/Desktop/teste/files/files/0051-png-c7.png	2023-01-30 11:16:12.947856+00	4.003166	65536	[binary data]
4	4874ebfc...	24	3	0	24	5	32	OPEN	8275	example7.exe	/home/ubuntu/Desktop/teste/files/files/0025-svg-from-web.svg	2023-01-30 11:16:14.948061+00	4.144383	65536	[binary data]
5	0cf1027b...	24	3	0	24	5	32	OPEN	8275	example7.exe	/home/ubuntu/Desktop/teste/files/files/0033-xml.xml	2023-01-30 11:16:16.948084+00	5.075891	1004	[binary data]

Figure 4.3: The figure represents the example of the stored data.

4.4 Client

For the data collection part of the file system, two libraries were verified the inotify and fanotify libraries, to better understand the operation of the libraries were executed tests and evaluated the performance and functionalities of each one, as a conclusion of these

tests we decided to proceed with the use of the fanotify library because it had more data that could be used and easier integration with functionalities that we wanted to develop, the test code can be seen in (Appendix B.1) for the inotify library and in (Appendix B.2) for the fanotify library.

After choosing the library we developed a more robust code that could capture several events from the file system to evaluate what would be interesting to collect in our study. This code was important so that we could understand the general functioning of the library and also how the log system behaved during the use of the system, the code that was developed for testing can be seen in Appendix B.3.

To be able to keep the collected data safe during the infection experiments it was necessary to find a way to store this data, so using the PostgreSQL library for the C language it was possible to connect to our database created in section 4.1 and store the logs in real-time in the database during the use of the system, the example of the code used as a basis for the connection can be checked in (Appendix B.4).

It was important for our study that it was possible to collect the entropy of the file so that we could have a basis for comparison with the original file and the infected file so that there would be a way to evaluate whether or not the file could have been encrypted by ransomware, for this we chose the Shannon entropy calculation a method already known and widely used in research in the scientific community as in [4], and you can check the code developed for testing in (Appendix B.5).

At the end of these checks, all these codes were adapted into a final data collection model that checks only the file change logs that we were most interested in analyzing, this model saves the data in the database and also checks at each file change whether or not a file may have been compromised using the intelligent model created that is called through a C algorithm that uses a python file created just to check the data against the models trained later in section 4.3, the final code used for testing in the experiments can be seen in (Appendix B.6).

An activity diagram follows in Figure 4.4 showing how the client module works to clarify the reader's understanding, which will also be followed by a brief explanation.

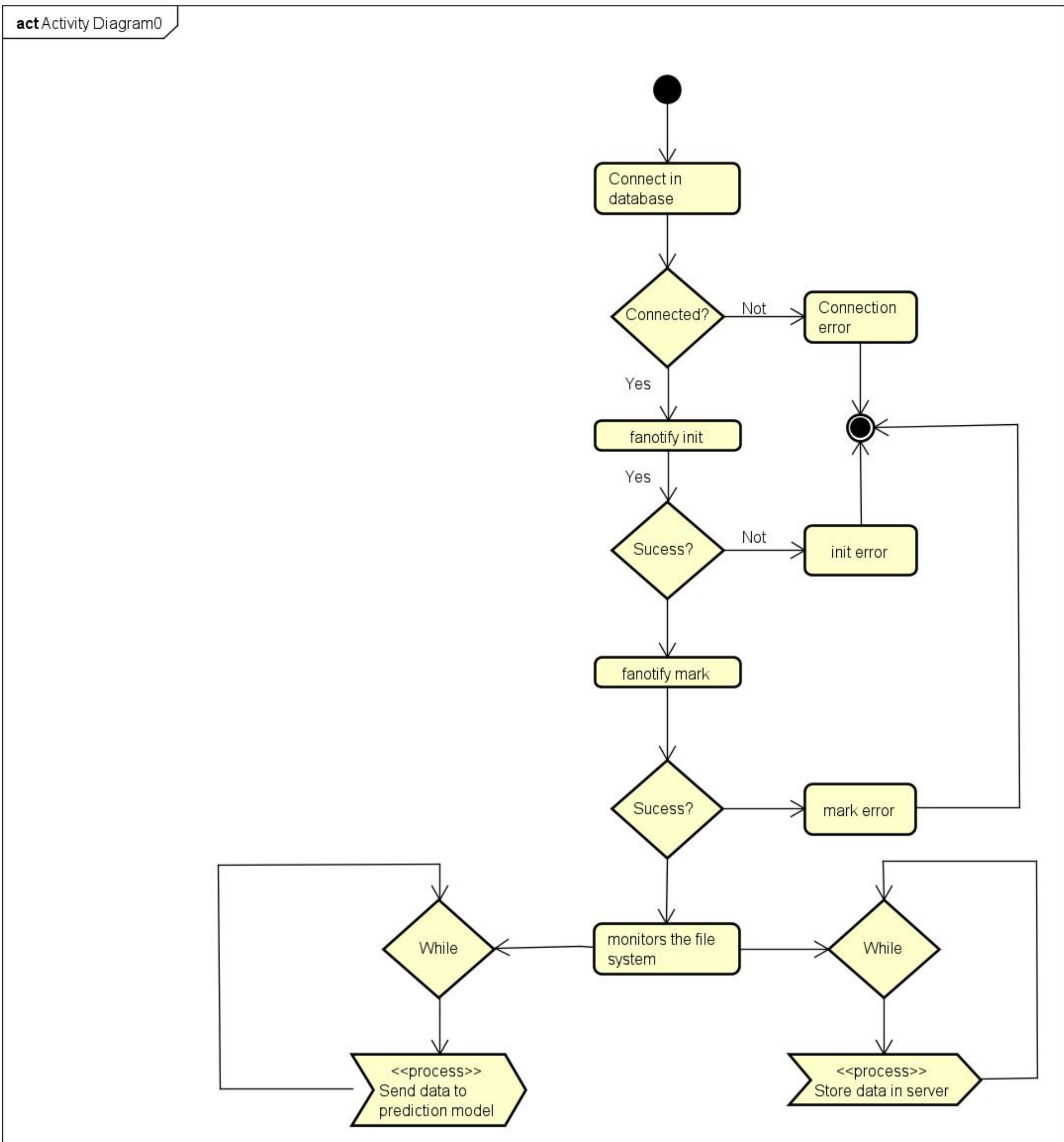

Figure 4.4: Client Activity Diagram

The flow can be summarized as follows: The process starts with trying to connect to the database where the logs of access to the filesystem will be stored, if it succeeds it proceeds to the initialization of the fanotify library, and if it succeeds it tries to set up the directory that is received by a parameter in the program call if all goes well it starts

the program and starts monitoring the changes in the filesystem.

At this point two processes will be executed, one responsible for saving the data of file changes in the database that only captures the data and saves it using an existing library, and the other responsible for calling the intelligent model by sending the entropy and file size information so that the model can check whether the file may or may not have been encrypted.

4.5 Model

In this section, we will discuss the process of building the intelligent model in general, explain how to select the best fields and also the best algorithms, arrive at the final trained models, and also at a new program that uses these models as a basis to tell whether a file may or may not have been encrypted.

To have sufficient infection data and to be able to create robust models, tests were run using 8 types of existing ransomware for Linux, the system was infected with these viruses, and the data from infected files were collected and stored in the database for later analysis, below is the list of ransomware used in the experiment.

- Revil(tutorialjinni) [36]
- TellYouThePass(tutorialjinni) [37]
- HelloKitty(tutorialjinni) [38]
- AvosLocker(tutorialjinni) [39]
- Conti(MALWARE bazaar) [40]
- Sodinoki(MALWARE bazaar) [41]
- Monti(MALWARE bazaar) [42]
- Hive(MALWARE bazaar) [43]

After the execution of the experiment and data collection, it was possible to assemble the intelligent models Table 4.4 shows the number of files that were used to build each model, the models were balanced to always have the same number of malicious files and legitimate files, this was necessary due to a problem that happens in some cases of collecting information from files.

Ransomware	**Files**	**Legitimate files**	**Total Files**
AvosLocker	9376	9376	18752
Conti	9381	9381	18762
HelloKitty	9381	9381	18762
Hive	9381	9381	18762
Monti	9374	9374	18748
Revil	9376	9376	18752
Sodinoki	9381	9381	18762
TellYouThePass	9377	9377	18754

Table 4.4: Total number of files for each model

To create the intelligent model some steps were performed so that the model could be more reliable, one of these steps was the selection of features that were relevant to the creation of the model as described in section 3.1.5. Even with an interesting amount of columns as seen in table 4.1, some of these data would not be very relevant because they are frequently repeated data, For instance, the field Process Identifier (PID), associated with a ransomware infection, remains constant throughout the entire execution. If we use these data, the algorithm would have a negative impact because in the prediction we would have a variable with 100% of the value and this would make the algorithm biased and always classify a PID change as a malicious process. After a deeper analysis, we verified that the most interesting fields for the evaluation would be the entropy, filebytes, and magicbytes columns. However, the magicbytes field was discarded because, being a text field, it would be difficult to create a good analysis variable with it and, consequently,

this would have a negative impact on the algorithm since it would be difficult to have an exact translation of that field for the machine learning algorithm.

After defining which would be the best variables we followed the procedure for creating the model devised in section 3.1.6, two data sets were joined, the data set of the original files and the data set of the files infected with the *conti* ransomware, for model training purposes a flag called "infected" was created that would have a value of 0 for uninfected files and a value of 1 for infected files, after this creation, the correlation tests of the variables were run, the result can be seen in Table 4.5.

	entropy	**filebytes**	**infected**
entropy	1.000000	-0.072705	-0.063014
filebytes	-0.072705	1.000000	-0.003277
infected	-0.063014	-0.003277	1.000000

Table 4.5: The table represents the result of the correlation of the variables.

To exemplify the functioning of the algorithm a pseudocode was developed to make understanding easier, and a short explanation of the model will be made.

Algorithm 1 Checks out the best intelligent model

Input: $dataset \leftarrow conti.csv$

Output: $dataframe \leftarrow dataRest$

$N \leftarrow 0$

$y \leftarrow dataset['infected']$

$x \leftarrow dataset[list(filter(lambda\ x : x\ in\ ['entropy', 'filebytes'], dataset.columns))]$

$models \leftarrow [AdaBoostClassifier]$

$names \leftarrow ['AdaBoost']$

for $N \leq 100$ **do**

$X_train, X_test, y_train, y_test \leftarrow train_test_split(x, y, test_size \leftarrow 0.33)$

forall $name,\ model\ in\ names, models$ **do**

$model \leftarrow model()$

$model.fit(X_train, y_train)$

$frame \leftarrow DataFrame(frame['real'] \leftarrow y_test, frame['precict'] \leftarrow model.predict(X_test))$

$tn, fp, fn, tp \leftarrow confusion_matrix(frame['real'], frame['predict']).ravel()$

$accuracy \leftarrow (tp + tn)/(tp + fp + tn + fn)$

$recall \leftarrow tp/(tp + fn)$

$precision \leftarrow tp/(tp + fp)$

$fscore \leftarrow 2 * ((precision * recall)/(precision + recall))$

$dataRest['modelo'] \leftarrow name$

$dataRest['accuracy'] \leftarrow accuracy$

$dataRest['recall'] \leftarrow recall$

$dataRest['precision'] \leftarrow precision$

$dataRest['f1 - score'] \leftarrow fscore$

end

end

$print(DataFrame(dataRest).groupby('modelo').mean())$

As we can see in Algorithm 1 the program uses a dataset to train the models, it selects

the predefined columns and puts the target to be calculated that in our case is the column "infected", then it starts running the 100 tests for the models it captures every new run the training data and test data with a dynamic seed to provide greater randomness in data, Subsequent to the selection process, tests were conducted for each model and the results were stored within a matrix for later use, at the end of the program he groups all the data obtained by model and makes an average of the values and presents this data on the screen.

To create the model some tests were performed to verify the best algorithm that would fit our solution using algorithm 1, for the tests we used the 8 balanced datasets presented in the Table 4.4 for the verification of which machine learning algorithm would be the best for our approach, for testing we developed a program in python that checked several algorithms testing all 100 times and making an average of successes, as a result AdaBoost had the best performance, the test result can be seen in the Tables 4.6 to 4.13.

The Accuracy A given the number of true positives TP, the false positives FP, the false negative FN and the true negative TN can be calculated by the Equation 4.1:

$$A = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.1}$$

The precision P given the number of true positives TP and the false positives FP can be calculated by the Equation 4.2:

$$P = \frac{TP}{TP + FP} \tag{4.2}$$

The recall R given the number of true positive TP and the false negative FN can be calculated by the Equation 4.3:

$$R = \frac{TP}{TP + FN} \tag{4.3}$$

The F1-Score F given the number of precision P and the recall R can be calculated by the Equation 4.4:

$$F = \frac{2(P * R)}{P + R} \tag{4.4}$$

Algorithm	Accuracy	Recall	Precision	F1-Score
AdaBoost	0.838100	0.929329	0.787211	0.852387
Decision Tree	0.786395	0.774173	0.795642	0.784761
Gaussian Process	0.643238	0.984902	0.586570	0.735252
Linear SVM	0.627242	0.993897	0.574879	0.728428
Naive Bayes	0.647116	0.950530	0.593105	0.730437
Nearest Neighbors	0.825497	0.894957	0.787228	0.837643
Neural Net	0.781548	0.989721	0.700068	0.820069
QDA	0.683309	0.966270	0.618548	0.754263
RBF SVM	0.627242	0.993897	0.574879	0.728428
Random Forest	0.801745	0.813685	0.796541	0.805021

Table 4.6: Average accuracy of each algorithm for Avos after normalization

Algorithm	Accuracy	Recall	Precision	F1-Score
AdaBoost	0.833656	0.945771	0.772679	0.850508
Decision Tree	0.765504	0.758231	0.769659	0.763902
Gaussian Process	0.650840	1.000000	0.588973	0.741326
Linear SVM	0.634690	1.000000	0.577985	0.732561
Naive Bayes	0.758075	0.966430	0.682315	0.799893
Nearest Neighbors	0.819929	0.896708	0.777498	0.832859
Neural Net	0.781977	0.999354	0.696670	0.821002
QDA	0.767603	0.970303	0.690558	0.806872
RBF SVM	0.634690	1.000000	0.577985	0.732561
Random Forest	0.785691	0.806004	0.774744	0.790065

Table 4.7: Average accuracy of each algorithm for Conti after normalization

Algorithm	Accuracy	Recall	Precision	F1-Score
AdaBoost	0.825097	0.913361	0.780369	0.841644
Decision Tree	0.754683	0.749921	0.763736	0.756765
Gaussian Process	0.659399	0.918756	0.609730	0.733004
Linear SVM	0.630329	0.989844	0.580171	0.731559
Naive Bayes	0.658430	0.988258	0.599769	0.746494
Nearest Neighbors	0.805717	0.867026	0.777019	0.819559
Neural Net	0.770672	0.975246	0.696036	0.812318
QDA	0.657946	0.986671	0.599614	0.745921
RBF SVM	0.630329	0.989844	0.580171	0.731559
Random Forest	0.773740	0.790860	0.770563	0.780579

Table 4.8: Average accuracy of each algorithm for Hello Kitty after normalization

Algorithm	Accuracy	Recall	Precision	F1-Score
AdaBoost	0.796512	0.869716	0.764983	0.813995
Decision Tree	0.723837	0.708833	0.740606	0.724371
Gaussian Process	0.626453	0.839748	0.595926	0.697132
Linear SVM	0.614018	0.967508	0.572843	0.719615
Naive Bayes	0.609496	0.951420	0.571212	0.713846
Nearest Neighbors	0.775678	0.852997	0.745520	0.795645
Neural Net	0.755168	0.961199	0.686261	0.800788
QDA	0.609981	0.939748	0.572554	0.711573
RBF SVM	0.614018	0.967508	0.572843	0.719615
Random Forest	0.732720	0.744479	0.736349	0.740392

Table 4.9: Average accuracy of each algorithm for Hive after normalization

Algorithm	Accuracy	Recall	Precision	F1-Score
AdaBoost	0.836270	0.942202	0.777719	0.852095
Decision Tree	0.775497	0.761059	0.784098	0.772407
Gaussian Process	0.650719	1.000000	0.589007	0.741352
Linear SVM	0.633102	1.000000	0.577045	0.731805
Naive Bayes	0.758364	0.967388	0.682460	0.800321
Nearest Neighbors	0.820268	0.896351	0.778245	0.833133
Neural Net	0.800226	0.996771	0.715743	0.833198
QDA	0.769193	0.970617	0.692148	0.808065
RBF SVM	0.633102	1.000000	0.577045	0.731805
Random Forest	0.797155	0.811108	0.789441	0.800127

Table 4.10: Average accuracy of each algorithm for Monti after normalization

Algorithm	Accuracy	Recall	Precision	F1-Score
AdaBoost	0.822588	0.928058	0.763724	0.837910
Decision Tree	0.748425	0.744604	0.745824	0.745214
Gaussian Process	0.645500	0.999019	0.582348	0.735790
Linear SVM	0.613023	0.985939	0.561766	0.715727
Naive Bayes	0.624818	0.977436	0.570202	0.720241
Nearest Neighbors	0.799968	0.870831	0.759555	0.811395
Neural Net	0.769591	0.972204	0.689152	0.806565
QDA	0.627242	0.979398	0.571674	0.721948
RBF SVM	0.613023	0.985939	0.561766	0.715727
Random Forest	0.762643	0.779267	0.750079	0.764395

Table 4.11: Average accuracy of each algorithm for Revil after normalization

Algorithm	Accuracy	Recall	Precision	F1-Score
AdaBoost	0.820252	0.909931	0.766537	0.832101
Decision Tree	0.750646	0.739030	0.748413	0.743692
Gaussian Process	0.618217	0.999340	0.561862	0.719307
Linear SVM	0.607881	0.986473	0.556072	0.711227
Naive Bayes	0.623708	0.975916	0.567210	0.717439
Nearest Neighbors	0.802487	0.881887	0.755512	0.813822
Neural Net	0.739987	0.975256	0.658205	0.785961
QDA	0.623224	0.977895	0.566730	0.717589
RBF SVM	0.607881	0.986473	0.556072	0.711227
Random Forest	0.771318	0.786539	0.756105	0.771022

Table 4.12: Average accuracy of each algorithm for Sodinoki after normalization

Algorithm	Accuracy	Recall	Precision	F1-Score
AdaBoost	0.782841	0.874429	0.736538	0.799582
Decision Tree	0.703991	0.697652	0.702694	0.700164
Gaussian Process	0.599127	0.980104	0.553917	0.707808
Linear SVM	0.608499	0.979778	0.559925	0.712608
Naive Bayes	0.629019	0.962166	0.575049	0.719863
Nearest Neighbors	0.755372	0.804305	0.729586	0.765126
Neural Net	0.719502	0.944553	0.649036	0.769394
QDA	0.629342	0.963470	0.575156	0.720312
RBF SVM	0.608499	0.979778	0.559925	0.712608
Random Forest	0.720472	0.733855	0.711125	0.722311

Table 4.13: Average accuracy of each algorithm for Tell the pass after normalization

After analyzing the best algorithm to work with the data we had, we decided to

proceed using the Adaboost algorithm to create intelligent models that would be used later to analyze new values.

Individual models were created for each type of ransomware tested to maintain a balanced and efficient data set, the data was divided into 70% for training and 30% for testing and after that 100 tests were run with dynamic seed for each model, and only after the process was finished the model was exported.

For comparison purposes, the average accuracy of each model was collected and can be seen in Table 4.14.

Model	**Accuracy**	**Recall**	**Precision**	**F1-Score**
AvosLocker	0.842786	0.937902	0.788904	0.856975
Conti	0.833656	0.933205	0.779506	0.849459
HelloKitty	0.829619	0.922853	0.777959	0.844234
Hive	0.799903	0.872051	0.754457	0.809003
Monti	0.833845	0.933611	0.780011	0.849927
Revil	0.821619	0.904915	0.777103	0.836153
Sodinoki	0.819121	0.904238	0.772313	0.833085
TellYouThePass	0.784779	0.87323	0.743154	0.802959

Table 4.14: Average accuracy of each model

To clarify the functioning of the algorithm a pseudo-code has been made, and a brief explanation of its functioning will be presented.

Algorithm 2 Checks out the best intelligent model

Input: $argv1 \leftarrow sys.argv[1]$ $argv2 \leftarrow sys.argv[2]$

Output: $model \leftarrow dump_model$

$N \leftarrow 0$

$dataset \leftarrow read_csv(argv1)$

$y \leftarrow dataset['infected']$

$x \leftarrow dataset[list(filter(lambda\ x : x\ in\ ['entropy','filebytes'], dataset.columns))]$

$models \leftarrow [AdaBoostClassifier]$

$names \leftarrow ['AdaBoost']$

for $N \leq 100$ **do**

  $X_train, X_test, y_train, y_test \leftarrow train_test_split(x, y, test_size \leftarrow 0.33)$

  forall *name, model in names, models* **do**

    $model \leftarrow model()$

    $model.fit(X_train, y_train)$

    $frame \leftarrow DataFrame(frame['real'] \leftarrow y_test, frame['precict'] \leftarrow model.predict(X_test))$

  end

end

$with\ open(argv2,'wb')\ as\ model_file$

$dump(model, model_file)$

As we can see in Algorithm 2, the program receives two information parameters, one is the dataset that should be loaded and the other is the name of the model to be exported.

After loading the dataset, the data is trained and tested during 100 runs and then exported to the model, which is saved in a directory for later use.

In the end, the intelligent model created uses the training models as a basis to be able to say whether or not a file may have been encrypted, this model receives parameters that it uses for comparison, which are the entropy values and the number of bytes in the file, with these values, the algorithm tries to verify the possibility that the file has been encrypted and if the result is positive, it presents a message to the user of the machine.

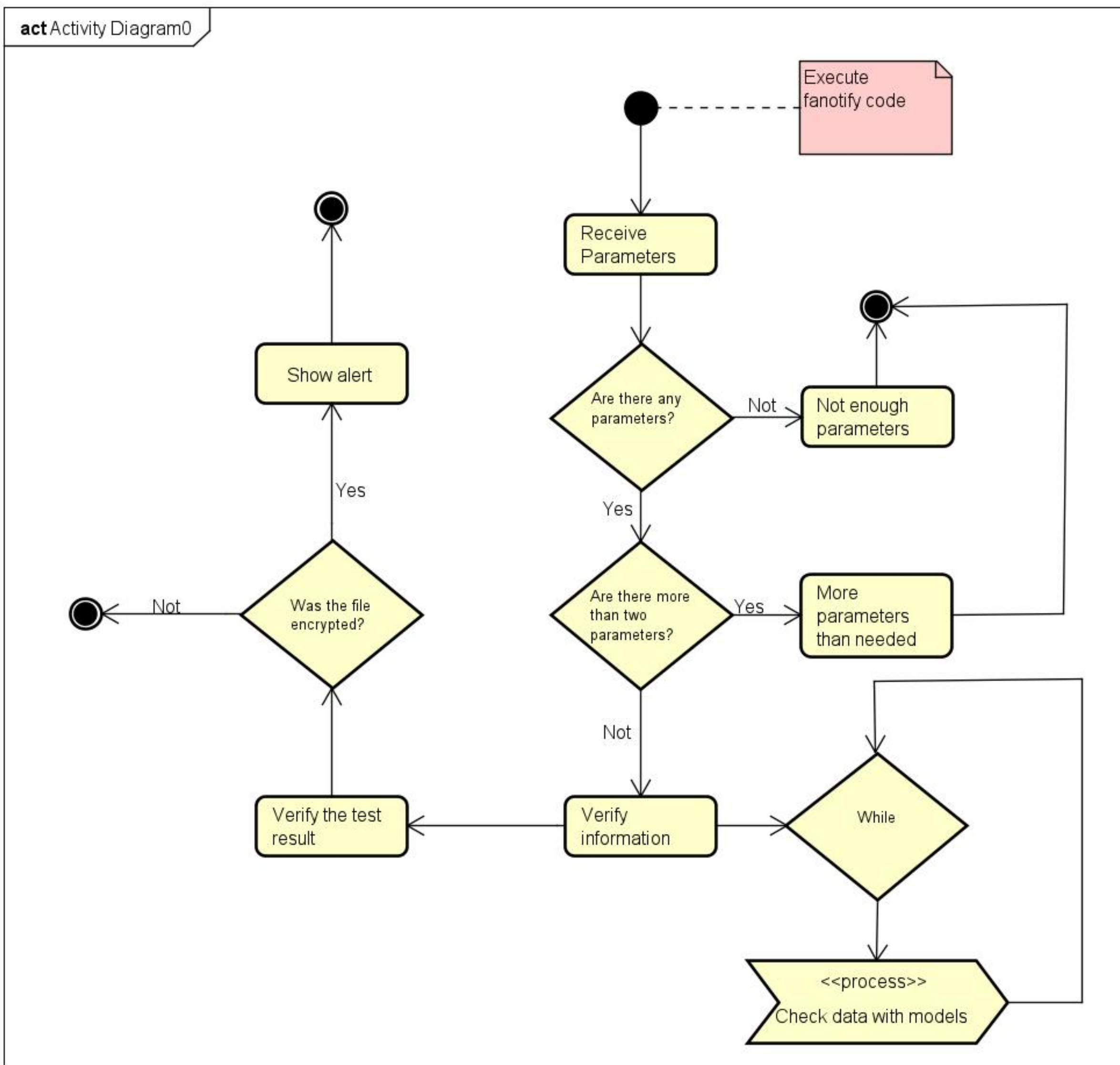

Figure 4.5: Model Activity Diagram

As we proceeded for the client, there follows an activity diagram in Figure 4.5 showing how the prediction module works, intending to clarify its understanding by the reader, which will also be followed by a brief explanation.

During the execution of the fanotify client, the intelligent model is called at each file change and receives two parameters, which are the entropy and the file size.

After ensuring that we have the necessary parameters for testing the models are loaded and the information is tested in each model, if any model classifies the file as suspicious

a message will be presented to the user stating that the file may have been encrypted and the process is terminated if no suspicion is found in any of the models the cycle ends waiting for the next program call.

In this chapter, the structure implemented during the development of this work was presented, contextualizing what was necessary for the creation of the log collection system and also the intelligent model.

The tools used in the development and their relationship with the project were addressed, it also presented an explanation of how the construction of the machine learning models was evaluated and what the results obtained during the construction.

Chapter 5

Discussion

In this chapter, we describe the experiments and discussions conducted on a model designed to address a specific research question. The model was created to capture the fundamental features of the phenomenon being investigated and to provide insights into its behavior. To validate the model, we carried out a series of experiments that involved altering its parameters and observing the resulting output. These experiments allowed us to obtain a deeper understanding of the model's strengths and weaknesses, and to identify areas that required further refinement. Furthermore, we engaged in extensive discussions about the model's results, analyzing the underlying assumptions and implications of our findings. These discussions enabled us to identify potential avenues for future research and to develop a more nuanced understanding of the phenomenon we were examining. Taken together, the experiments and discussions presented in this chapter yield valuable insights into the model's behavior and the phenomenon it represents. They demonstrate the effectiveness of computational modeling as a tool for exploring complex systems and underscore the importance of meticulous experimentation and analysis in this domain.

5.1 Experiments

In this section the tests performed for each of the 3 experiments will be presented, showing the process performed for testing and the results obtained from each of them.

5.1.1 File System Data Collection Algorithm

In order to conduct the tests, a program was developed to examine the virtual machine files to assess whether the system could collect this process that was being executed, the process was run for about 7 hours to assess the functioning of the algorithm which proved satisfactory. The data collected was comprehensive and largely accurate, and the algorithm was suitable to handle a large volume of data in a short time, with only few problems encountered in high-volume cases.

One of the algorithm's crucial strengths was its capability to identify and collect file metadata, including creation dates, file size, and entropy. This metadata was essential for the accurate analysis of file systems and the detection of any problems or anomalies.

The algorithm's speed and effectiveness were also interesting, enabling it to collect data from large file systems in a fairly short time. This point is essential for the accuracy of the ransomware detection algorithm.

Taken together, the test results indicate that the file system data collection algorithm is a largely effective and dependable tool. Its accurate data collection and analysis, speed and effectiveness, and capability to handle large volumes of data make it an inestimable asset for our study.

We used the file data collection system created to collect information from the files in their original state, it was also used to collect the information from the files during the 8 infections executed and thus we were able to obtain the necessary data for the creation of the datasets that were used in the training of our intelligent model.

5.1.2 Machine Learning Algorithm

After finishing the data collection process, we analyzed the data obtained and selected the best fields for our algorithm. We performed some tests using the remaining fields and the results were satisfactory, so we decided to proceed with the analysis.

For the data analysis, we decided to test 10 machine learning algorithms to see which one best adapted to the data we had available, for this 8 datasets were assembled with the

infection data we had collected, joining the information from the original files with the infected files and training the algorithm to classify new files as malicious or not, in this process the Adaboost algorithm was the one that we got the best results so we continued with it for the model creation process, the process for creating the model can be found in Section 4.5.

With the algorithm defined, the Pikle library was used to export the models that we trained so it would be possible to reuse them in the next step where a program was created that receives the entropy values and file size and tries to classify if the file may or may not have been encrypted, it performs the test on all models created that total 8 checks, but if a failure is found before finishing the complete verification it already returns the error so that the process is more efficient.

After completing this process, an alert message was added to warn the user that the file may have been encrypted, and then was added a call of this program to the main program that monitors the file system, so that every time a file is changed it is checked by the smart model to ensure that if any malicious process is identified the user can be alerted.

5.1.3 Detection Algorithm

In this section, the steps for replication of the executed experiment will be explained, and the data collected and how it was tested will also be presented.

Replication

For replication tests, it is important to have a Linux environment configured and divided into server and client, after having the environment ready the following steps can be performed.

To evaluate the behavior of a family of ransomware it is important to run the procedure in two parts to get a sense of the real behavior of ransomware because when the program that makes the notification of encryption is used the number of logs collected is reduced,

so during the first run should be run the program without checking the models, In the second run, the data should not be saved, only the data should be sent to the model, and as soon as the encryption message is presented, the execution of the program should be interrupted. To collect the time, the program can be run with the Linux time command.

After having this information it can be executed a query in the database that evaluates how many executions have been performed in the space of time of the execution until the message presentation, obtaining the number of events that were executed by the ransomware until it was detected, this is probably not the most efficient way to do it but due to this flaw in the logs collection and the safest way to compare data.

With this process, all experiments were finished and replicated to the remaining families presenting the data in a table.

Tests and Results

To verify the effectiveness of the model created in section 4.5 tests were performed using the ransomware described in that section and also to perform a more robust test we used two ransomware that were not used in the creation of the models to test the accuracy of the algorithm, the new ransomware will be presented below.

- Buhti(MALWARE bazaar) [44]
- Conti - ArkbirdDevil(MALWARE bazaar) [45]

To evaluate the performance of the algorithm some data that we will collect from the execution of each ransomware will be important to note the real effectiveness of the model, the data that will be collected are the execution time until the presentation of the alert about the possible encryption of data and also the number of events that occurred until the message was presented, this is important so that we know how quickly the algorithm can detect an anomaly.

The data from the experiments can be seen in Table 5.1 as well as an image that shows the message that is presented to the user of the system when the algorithm detects an anomaly that can be seen in Figure 5.1.

Ransomware	Runtime in seconds	Number of events
AvosLocker	3.319	1475
Conti	4.225	1735
HelloKitty	10.015	2180
Hive	13.567	1253
Monti	4.235	1688
Revil	3.738	1570
Sodinoki	6.149	1005
TellYouThePass	10.291	2500
Buhti	7.357	2710
Conti - ArkbirdDevil	4.524	1236

Table 5.1: Result of experiments

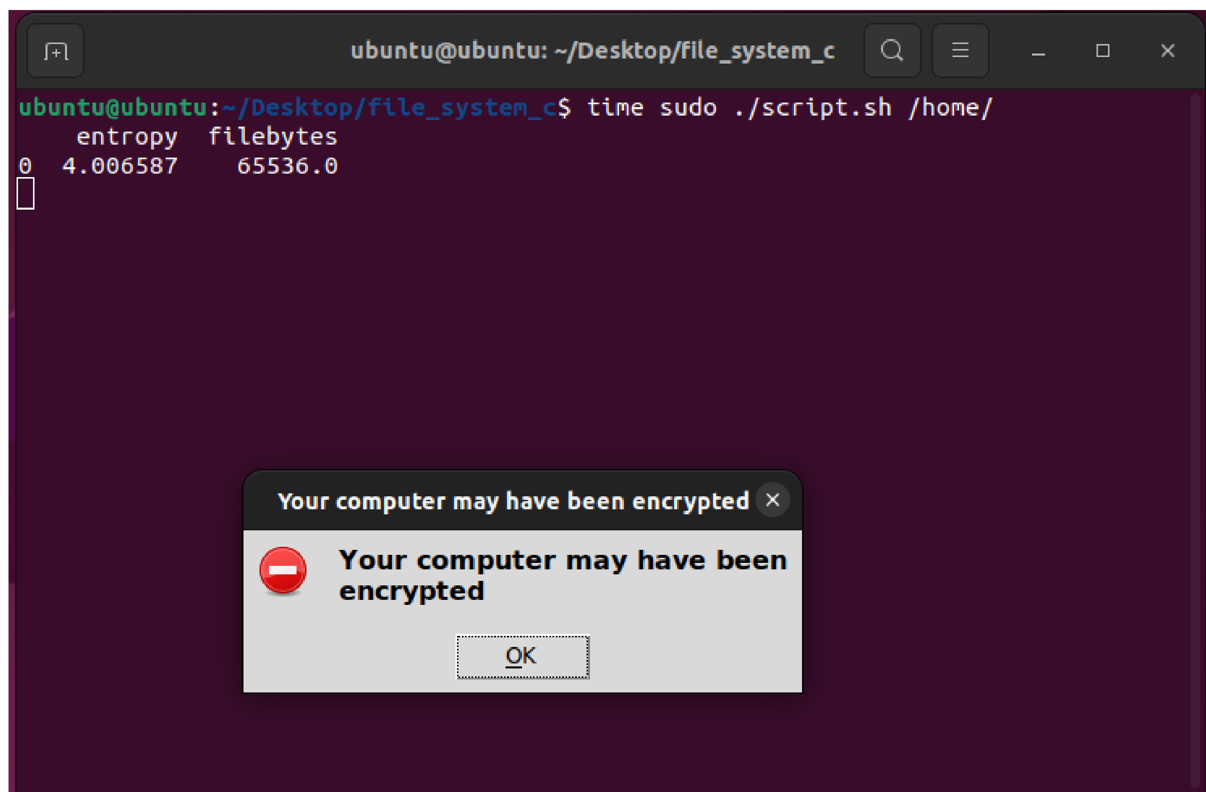

Figure 5.1: The figure represents the message presented to the user when the algorithm detects an anomaly.

5.2 Limitations

Some limitations were encountered during the development of this project, which negatively impacted the overall result of the algorithm, one of these limitations was finding ransomware that could satisfactorily run on a Linux environment, this impacted the number of families analyzed during the work and also the quality of the model, another problem was some limitations of the fanotify library to collect some important data in our study as the number of bytes written that made it more laborious to collect this information making it only possible in the end to have the total number of bytes of the whole file, another limitation that impacted the model was the number of features available for analysis, with a smaller number the success of the model was somewhat compromised having to be increased the number of training data to have a greater success of the algorithm.

Chapter 6

Conclusions and future work

The current chapter discusses the conclusions drawn from the analysis and development stages of the algorithm. Firstly, the conclusions are presented, followed by a description of future work.

6.1 Conclusions

With the proliferation and diffusion of cyberattacks, it becomes imperative to promptly identify and respond to such incidents in order to minimize the damage to the system. Consequently, there is a requirement for tools and systems for gathering intelligence that can detect attacks in real-time.

With the significant increase in ransomware attacks Linux systems have become a major focus of large groups, aiming to obtain data and attack servers that were previously considered secure.

Based on building and testing our ransomware detection model for Linux systems, it can be concluded that this model can effectively detect ransomware attacks. The model uses a combination of machine learning algorithms and system call analysis to detect and classify suspicious behavior and has demonstrated satisfactory accuracy during testing.

This model provides a valuable tool for organizations and individuals looking to protect their Linux systems from the growing threat of ransomware attacks. By monitoring

system calls and analyzing behavior patterns, the model can detect and warn the user of ransomware attacks. We also contribute to the comunity by sharing our datasets used for the creation of the model [46].

Overall, this ransomware detection model demonstrates the potential of machine learning and system call analysis in developing effective cybersecurity solutions. Continued research and development in this area can help improve the detection and prevention of ransomware attacks and other cyber threats, ultimately improving the security of Linux systems and the protection of sensitive data.

6.2 Future Work

There are several potential scenarios for future work arising from this project. One idea that was not executed but could be important is the implementation of a file backup system using a library such as FUSE. This could be useful in recovering original files in the event of incidents involving ransomware.

Another important approach could involve a hybrid system that incorporates network data to improve the algorithm's performance. Additionally, it would be beneficial to expand the range of variables analyzed to increase the algorithm's reliability and robustness, ultimately improving the quality of detections, also it would be important to have a dataset of the normal use of the system to note the evolution of the variables during normal use of system.

It is also planned to be implemented as an executable application to enable installation of the detection system on any Linux-based machine via a package or other approach.

Bibliography

[1] M. Masum, M. J. H. Faruk, H. Shahriar, K. Qian, D. Lo, and M. I. Adnan, "Ransomware classification and detection with machine learning algorithms," Institute of Electrical and Electronics Engineers Inc., 2022, pp. 316–322, ISBN: 9781665483032. DOI: 10.1109/CCWC54503.2022.9720869.

[2] *Annual number of ransomware attacks worldwide from 2016 to first half 2022*, https://www.statista.com/statistics/494947/ransomware-attacks-per-year-worldwide/, Accessed: 2022-11-26, Aug. 2022.

[3] *Trend micro warns of 75% surge in ransomware attacks on linux as systems adoptions soared*, https://www.trendmicro.com/en_hk/about/newsroom/press-releases/2022/09-01-2022.html, [Online; accessed 8-March-2023], Sep. 2022.

[4] J. Zhu, J. Jang-Jaccard, A. Singh, I. Welch, H. AI-Sahaf, and S. Camtepe, "A few-shot meta-learning based siamese neural network using entropy features for ransomware classification," *Computers and Security*, vol. 117, Jun. 2022, ISSN: 01674048. DOI: 10.1016/j.cose.2022.102691.

[5] F. Manavi and A. Hamzeh, "A novel approach for ransomware detection based on pe header using graph embedding," *Journal of Computer Virology and Hacking Techniques*, 2022, ISSN: 22638733. DOI: 10.1007/s11416-021-00414-x.

[6] H. V. D. Parunak, "A grammar-based behavioral distance measure between ransomware variants," *IEEE Transactions on Computational Social Systems*, vol. 9, pp. 8–17, 1 Feb. 2022, ISSN: 2329924X. DOI: 10.1109/TCSS.2021.3060972.

[7] M. S. Abbasi, H. Al-Sahaf, M. Mansoori, and I. Welch, "Behavior-based ransomware classification: A particle swarm optimization wrapper-based approach for feature selection," *Applied Soft Computing*, vol. 121, May 2022, ISSN: 15684946. DOI: 10.1016/j.asoc.2022.108744.

[8] S. Aurangzeb, H. Anwar, M. A. Naeem, and M. Aleem, "Bigrc-eml: Big-data based ransomware classification using ensemble machine learning," *Cluster Computing*, vol. 25, pp. 3405–3422, 5 Oct. 2022, ISSN: 15737543. DOI: 10.1007/s10586-022-03569-4.

[9] H. Madani, N. Ouerdi, A. Boumesaoud, and A. Azizi, "Classification of ransomware using different types of neural networks," *Scientific Reports*, vol. 12, 1 Dec. 2022, ISSN: 20452322. DOI: 10.1038/s41598-022-08504-6.

[10] B. M. Khammas, "Comparative analysis of various machine learning algorithms for ransomware detection," *Telkomnika (Telecommunication Computing Electronics and Control)*, vol. 20, pp. 43–51, 1 Feb. 2022, ISSN: 23029293. DOI: 10.12928/TELKOMNIKA.v20i1.18812.

[11] X. Zhang, J. Wang, and S. Zhu, "Dual generative adversarial networks based unknown encryption ransomware attack detection," *IEEE Access*, vol. 10, pp. 900–913, 2022, ISSN: 21693536. DOI: 10.1109/ACCESS.2021.3128024.

[12] D. W. Fernando and N. Komninos, "Fesa: Feature selection architecture for ransomware detection under concept drift," *Computers and Security*, vol. 116, May 2022, ISSN: 01674048. DOI: 10.1016/j.cose.2022.102659.

[13] J. A. Gómez-Hernández, R. Sánchez-Fernández, and P. García-Teodoro, "Inhibiting crypto-ransomware on windows platforms through a honeyfile-based approach with r-locker," *IET Information Security*, vol. 16, pp. 64–74, 1 Jan. 2022, ISSN: 17518717. DOI: 10.1049/ise2.12042.

[14] R. M. A. Molina, S. Torabi, K. Sarieddine, E. Bou-Harb, N. Bouguila, and C. Assi, "On ransomware family attribution using pre-attack paranoia activities," *IEEE*

Transactions on Network and Service Management, vol. 19, pp. 19–36, 1 Mar. 2022, ISSN: 19324537. DOI: 10.1109/TNSM.2021.3112056.

[15] A. Mantri, N. Singh, K. Kumar, and S. Dahiya, “Pre-encryption and identification (pei): An anti-crypto ransomware technique,” *IETE Journal of Research*, 2022, ISSN: 0974780X. DOI: 10.1080/03772063.2022.2048706.

[16] A. Arfeen, M. A. Khan, O. Zafar, and U. Ahsan, “Process based volatile memory forensics for ransomware detection,” *Concurrency and Computation: Practice and Experience*, vol. 34, 4 Feb. 2022, ISSN: 15320634. DOI: 10.1002/cpe.6672.

[17] S. J. Lee, H. Y. Shim, Y. R. Lee, T. R. Park, and I. G. Lee, “Ransomware detection using open-source tools,” vol. 2022-February, Institute of Electrical and Electronics Engineers Inc., 2022, pp. 1385–1391, ISBN: 9791188428090. DOI: 10.23919/ICACT53585.2022.9728873.

[18] S. Lee, N. su Jho, D. Chung, Y. Kang, and M. Kim, “Rcryptect: Real-time detection of cryptographic function in the user-space filesystem,” *Computers and Security*, vol. 112, Jan. 2022, ISSN: 01674048. DOI: 10.1016/j.cose.2021.102512.

[19] U. Zahoora, M. Rajarajan, Z. Pan, and A. Khan, “Zero-day ransomware attack detection using deep contractive autoencoder and voting based ensemble classifier,” *Applied Intelligence*, Sep. 2022, ISSN: 15737497. DOI: 10.1007/s10489-022-03244-6.

[20] B. N. Chaithanya and S. H. Brahmananda, “Detecting ransomware attacks distribution through phishing urls using machine learning,” in *Computer Networks and Inventive Communication Technologies*, S. Smys, R. Bestak, R. Palanisamy, and I. Kotuliak, Eds., Singapore: Springer Singapore, 2022, pp. 821–832, ISBN: 978-981-16-3728-5.

[21] M. Al-Shabi, “A survey on symmetric and asymmetric cryptography algorithms in information security,” *International Journal of Scientific and Research Publications (IJSRP)*, vol. 9, no. 3, pp. 576–589, 2019.

[22] E. Gandotra, D. Bansal, and S. Sofat, “Malware analysis and classification: A survey,” *Journal of Information Security*, vol. 2014, 2014.

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

[24] S. Jung and Y. Won, “Ransomware detection method based on context-aware entropy analysis,” *Soft Computing*, vol. 22, pp. 6731–6740, 2018.

[25] S. R. Davies, R. Macfarlane, and W. J. Buchanan, “Napierone: A modern mixed file data set alternative to govdocs1,” *Forensic Science International: Digital Investigation*, vol. 40, p. 301 330, 2022, ISSN: 2666-2817. DOI: `https://doi.org/10.1016/j.fsidi.2021.301330`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S2666281721002560`.

[26] Kirsten Barkved, *Data cleaning: The most important step in machine learning*, `https://www.obviously.ai/post/data-cleaning-in-machine-learning`, [Online; accessed 8-March-2023], Jan. 2022.

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

[28] Martín Abadi, Ashish Agarwal, Paul Barham, *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: `https://www.tensorflow.org/`.

[29] A. Paszke, S. Gross, F. Massa, *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems 32*, Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[30] F. Chollet *et al.* “Keras.” (2015), [Online]. Available: `https://github.com/fchollet/keras`.

[31] U. authors, *fanotify(7) - Linux manual page*, https://manpages.ubuntu.com/manpages/focal/en/man7/fanotify.7.html, Accessed: March 8, 2023, 2021.

[32] P. G. D. Group. "Libpq - c library for postgresql." (current), [Online]. Available: https://www.postgresql.org/docs/current/libpq.html.

[33] pandas Development Team, *Pandas documentation*, https://pandas.pydata.org/docs/, Accessed on March 8, 2023, 2021.

[34] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, *Scikit-learn: Machine learning in Python*, https://scikit-learn.org/stable/index.html, Accessed: March 8, 2023, 2021.

[35] Python Software Foundation, *pickle – Python object serialization*, https://docs.python.org/3/library/pickle.html, [Online; accessed 8-March-2023], 2021.

[36] *Revil ransomware linux download*, https://www.tutorialjinni.com/revil-ransomware-linux-download.html, [Online; accessed 8-February-2023], Jul. 2021.

[37] *Tellyouthepass ransomware download*, https://www.tutorialjinni.com/tellyouthepass-ransomware-download.html, [Online; accessed 8-February-2023], Dec. 2021.

[38] *Hellokitty ransomware linux variant download*, https://www.tutorialjinni.com/hellokitty-ransomware-linux-variant-download.html, [Online; accessed 8-February-2023], Aug. 2021.

[39] *Avoslocker esxi linux ransomware*, https://www.tutorialjinni.com/avoslocker-esxi-linux-ransomware.html, [Online; accessed 8-February-2023], Jan. 2022.

[40] *Malwarebazaar | sha256 95776f31cbcac08eb3f3e9235d07513a6d7a6bf9f1b7f3d400b2cf0afdb088a7 (conti)*, https://bazaar.abuse.ch/sample/95776f31cbcac08eb3f3e9235d07513a6d7a6bf9f1b7f3d400b2cf0afdb088a7/, [Online; accessed 8-February-2023], Apr. 2022.

[41] *Malwarebazaar | sha256 f864922f947a6bb7d894245b53795b54b9378c0f7633c521240488e86f60c2c5 (sodinokibi)*, https://bazaar.abuse.ch/sample/f864922f947a6bb7d894245b53795b54b9378c0f7633c521240488e86f60c2c5/, [Online; accessed 8-February-2023], Oct. 2021.

[42] *Malwarebazaar | sha256 edfe81babf50c2506853fd8375f1be0b7bebbefb2e5e9a33eff95ec23e867de1 (monti)*, https://bazaar.abuse.ch/sample/edfe81babf50c2506853fd8375f1be0b7bebbefb2e5e9a33eff95ec23e867de1/, [Online; accessed 8-February-2023], Dec. 2022.

[43] *Malwarebazaar | sha256 6d4cb4dcae0ad95c2112b2b2c110e296475daf2f64cf64720caec172555ba6f5*, https://bazaar.abuse.ch/sample/6d4cb4dcae0ad95c2112b2b2c110e296475daf2f64cf64720caec172555ba6f5/, [Online; accessed 8-February-2023], Sep. 2022.

[44] *Malwarebazaar | sha256 01b09b554c30675cc83d4b087b31f980ba14e9143d387954df484894115f82d4 (buhti)*, https://bazaar.abuse.ch/sample/01b09b554c30675cc83d4b087b31f980ba14e9143d387954df484894115f82d4/, [Online; accessed 18-February-2023], Feb. 2023.

[45] *Malwarebazaar | sha256 35ea625eb99697efdeb016192b25c5323ec10b0b33642cd9b2641e058e5e8dc6 (conti)*, https://bazaar.abuse.ch/sample/35ea625eb99697efdeb016192b25c5323ec10b0b33642cd9b2641e058e5e8dc6/, [Online; accessed 18-February-2023], Jan. 2023.

[46] T. Pedrosa, V. Belloli, and N. C. Will, *Replication Data for: Mechanisms for Analysis and Detection of Ransomware in Desktop Operating Systems*, version V1, 2023. DOI: 10.34620/dadosipb/J25YYO. [Online]. Available: https://doi.org/10.34620/dadosipb/J25YYO.

Appendix A

Original dissertation proposal

INSTITUTO POLITÉCNICO DE BRAGANÇA

UTFPR UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

Proposta de tema para
Dissertação/Estágio/Projeto - Trabalho de Conclusão de Curso

Orientador da Instituição onde se realiza o trabalho:

Tiago Miguel Ferreira Guimarães Pedrosa	pedrosa@ipb.pt

Instituição do orientador:

IPB	ESTiG

Co-orientador da Instituição parceira:

Newton Carlos Will	will@utfpr.edu.br

Instituição do co-orientador:

UTFPR	Dois Vizinhos

Curso ou cursos da Instituição do orientador onde se propõe que o trabalho seja realizado:

Mestrado em Informática - IPB

Título do trabalho:

Mecanismos para Análise e Detecção de Ransomwares em Sistemas Operacionais Desktop

Palavras chave:

Ransomware; malware; cibersegurança; detecção de ameaças

Objetivos:

O trabalho proposto tem como objetivo construir um modelo de algoritmo detector de ransomwares, que possa ser aplicado de maneira nativa em sistemas Linux. Propõe-se que o algoritmo seja executado após as chamadas de escrita do sistema de arquivos, e assim verifique a mudança de estado de cada arquivo que é escrito, detectando padrões de comportamento compatíveis com ataques de ransomware.

Descrição adicional:

Um ransomware é um tipo de software malicioso (malware) cujo objetivo é a cifragem dos arquivos de um usuário, para então pedir uma quantia de dinheiro em troca do "resgate" para decifrar esses arquivos. Como ransomwares têm a característica de ação rápida e altamente destrutiva, os softwares utilizados para detecção de anomalias de softwares maliciosos em um sistema, como o antivírus, não conseguem identificar a invasão a tempo de conseguir impedir a ação do ransomware. Isso ocorre porque, para conseguir detectar um ataque real, é necessário que o antivírus ou softwares utilizados para proteção identifiquem que os arquivos do usuário já foram efetivamente cifrados pelo malware.
Algumas soluções para esse problema começaram a ser desenvolvidas para sistemas operacionais Windows, mas ainda existe uma grande defasagem em sistemas Linux contra esse tipo de malware. Assim, destaca-se a importância deste trabalho, visto que, apesar de sistemas Linux serem pouco utilizados por usuários domésticos, quando comparado com outros sistemas operacionais, ele é largamente utilizado por muitas empresas e organizações, as quais são alvos potenciais para esse tipo de ataque.

Metodologia/Plano de trabalhos:

As seguintes atividades serão desenvolvidas ao longo deste trabalho:
1 - Revisão da literatura acerca das metologias para detecção de ransomwares em sistemas operacionais (M1-M3);
2 - Análise crítica do estado da arte (M4-M5);
3 - Definição dos algoritmos e modelos a serem implementados (M6);
4 - Construção de um protótipo como prova de conceito (M7-M9);
5 - Testes de validação e coleta de resultados (M10-M12).
A escrita da dissertação irá decorrer ao longo de todo o período de trabalho.

Recursos necessários:

Ferramentas de desenvolvimento e ambiente Linux para testes.

Appendix B

Codes created for the client side

B.1 Inotify first code test

```
/*This is the sample program to notify us for the file creation and file
   deletion takes place in /tmp directory
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <linux/inotify.h>

#define EVENT_SIZE ( sizeof (struct inotify_event) )
#define EVENT_BUF_LEN  ( 1024 * ( EVENT_SIZE + 16 ) )

int main( )
{
  int length, i = 0;
  int fd;
  int wd;
  char buffer[EVENT_BUF_LEN];
```

```
/*creating the INOTIFY instance*/
fd = inotify_init();

/*checking for error*/
if ( fd < 0 ) {
  perror( "inotify_init" );
}

/*adding the  / tmp  directory into watch list. Here, the suggestion is to
    validate the existence of the directory before adding into monitoring
    list.*/
wd = inotify_add_watch( fd, "/home/debian-tese/Desktop/teste", IN_CREATE |
    IN_DELETE );

/*read to determine the event change happens on /tmp directory. Actually
    this read blocks until the change event occurs*/

length = read( fd, buffer, EVENT_BUF_LEN );

/*checking for error*/
if ( length < 0 ) {
  perror( "read" );
}

/*actually read return the list of change events happens. Here, read the
    change event one by one and process it accordingly.*/
while ( i < length ) {  struct inotify_event *event = ( struct inotify_event
    * ) &buffer[ i ]; if ( event->len ) {
    if ( event->mask & IN_CREATE ) {
```

```
        if ( event->mask & IN_ISDIR ) {
          printf( "New directory %s created.\n", event->name );
        }
        else {
          printf( "New file %s created.\n", event->name );
        }
      }
      else if ( event->mask & IN_DELETE ) {
        if ( event->mask & IN_ISDIR ) {
          printf( "Directory %s deleted.\n", event->name );
        }
        else {
          printf( "File %s deleted.\n", event->name );
        }
      }
    }
    i += EVENT_SIZE + event->len;
  }
  /*removing the  / tmp  directory from the watch list.*/
   inotify_rm_watch( fd, wd );

  /*closing the INOTIFY instance*/
   close( fd );

}
```

B.2 Fanotify first code test

```
/*
```

```
 *   File:  fanotify-example-access-control.c
 *   Date:  Thu Nov 14 13:47:37 2013
 *   Author: Aleksander Morgado <aleksander@lanedo.com>
 *
 *   A simple tester of fanotify in the Linux kernel.
 *
 *   This program is released in the Public Domain.
 *
 *   Compile with:
 *     $> gcc -o fanotify-example-access-control
    fanotify-example-access-control.c
 *
 *   Run as:
 *     $> ./fanotify-example-access-control /mount/path /another/mount/path ...
 */

/* Define _GNU_SOURCE, Otherwise we don't get O_LARGEFILE */
#define _GNU_SOURCE

#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <poll.h>
#include <errno.h>
#include <limits.h>
#include <sys/stat.h>
#include <sys/signalfd.h>
#include <fcntl.h>
```

```c
#include <linux/fanotify.h>

/* Structure to keep track of monitored directories */
typedef struct
{
    /* Path of the directory */
    char *path;
} monitored_t;

/* Size of buffer to use when reading fanotify events */
#define FANOTIFY_BUFFER_SIZE 8192

/* Enumerate list of FDs to poll */
enum
{
    FD_POLL_SIGNAL = 0,
    FD_POLL_FANOTIFY,
    FD_POLL_MAX
};

/* Setup fanotify notifications (FAN) mask. All these defined in fanotify.h. */
static uint64_t event_mask =
    (FAN_OPEN_PERM); /* Open permission control */

/* Array of directories being monitored */
static monitored_t *monitors;
static int n_monitors;

static char *
```

```
get_program_name_from_pid(int pid,
                          char *buffer,
                          size_t buffer_size)
{
    int fd;
    ssize_t len;
    char *aux;

    /* Try to get program name by PID */
    sprintf(buffer, "/proc/%d/cmdline", pid);
    if ((fd = open(buffer, O_RDONLY)) < 0)
        return NULL;

    /* Read file contents into buffer */
    if ((len = read(fd, buffer, buffer_size - 1)) <= 0)
    {
        close(fd);
        return NULL;
    }
    close(fd);

    buffer[len] = '\0';
    aux = strstr(buffer, "^@");
    if (aux)
        *aux = '\0';

    return buffer;
}

static char *
```

```
get_file_path_from_fd(int fd,
                      char *buffer,
                      size_t buffer_size)
{
    ssize_t len;

    if (fd <= 0)
        return NULL;

    sprintf(buffer, "/proc/self/fd/%d", fd);
    if ((len = readlink(buffer, buffer, buffer_size - 1)) < 0)
        return NULL;

    buffer[len] = '\0';
    return buffer;
}

static void
event_process(struct fanotify_event_metadata *event,
              int fanotify_fd)
{
    char file_path[PATH_MAX];
    char program_path[PATH_MAX];

    printf("Received event in path '%s'",
           (get_file_path_from_fd(event->fd, file_path, PATH_MAX) ? file_path :
              "unknown"));
    printf(" pid=%d (%s): \n",
           event->pid,
```

```
        (get_program_name_from_pid(event->pid, program_path, PATH_MAX) ?
          program_path : "unknown"));

    if (event->mask & FAN_OPEN_PERM)
    {
        struct fanotify_response access;

        access.fd = event->fd;

        /* We all know that files and paths containing '666' are evil,
         * so never allow them */
        if (strstr(file_path, "666") != NULL)
        {
            printf("\tFAN_OPEN_PERM: denying\n");
            access.response = FAN_DENY;
        }
        else
        {
            printf("\tFAN_OPEN_PERM: allowing\n");
            access.response = FAN_ALLOW;
        }

        /* Write response in the fanotify fd! */
        write(fanotify_fd, &access, sizeof(access));
    }

    fflush(stdout);
    close(event->fd);
}
```

```c
static void
shutdown_fanotify(int fanotify_fd)
{
    int i;

    for (i = 0; i < n_monitors; ++i)
    {
        /* Remove the mark, using same event mask as when creating it */
        fanotify_mark(fanotify_fd,
                      FAN_MARK_REMOVE,
                      event_mask,
                      AT_FDCWD,
                      monitors[i].path);
        free(monitors[i].path);
    }
    free(monitors);
    close(fanotify_fd);
}

static int
initialize_fanotify(int argc,
                    const char **argv)
{
    int i;
    int fanotify_fd;

    /* Create new fanotify device */
    if ((fanotify_fd = fanotify_init(FAN_CLOEXEC | FAN_CLASS_CONTENT,
                                     O_RDONLY | O_CLOEXEC | O_LARGEFILE |
                                       O_NOATIME)) < 0)
```

```
{
    fprintf(stderr,
            "Couldn't setup new fanotify device: %s\n",
            strerror(errno));
    return -1;
}

/* Allocate array of monitor setups */
n_monitors = argc - 1;
monitors = malloc(n_monitors * sizeof(monitored_t));

/* Loop all input directories, setting up marks */
for (i = 0; i < n_monitors; ++i)
{
    monitors[i].path = strdup(argv[i + 1]);
    /* Add new fanotify mark */
    if (fanotify_mark(fanotify_fd,
                      FAN_MARK_ADD | FAN_MARK_MOUNT,
                      event_mask,
                      AT_FDCWD,
                      monitors[i].path) < 0)
    {
        fprintf(stderr,
                "Couldn't add monitor in mount '%s': '%s'\n",
                monitors[i].path,
                strerror(errno));
        return -1;
    }

    printf("Started monitoring mount '%s'...\n",
```

```
            monitors[i].path);
    }

    return fanotify_fd;
}

static void
shutdown_signals(int signal_fd)
{
    close(signal_fd);
}

static int
initialize_signals(void)
{
    int signal_fd;
    sigset_t sigmask;

    /* We want to handle SIGINT and SIGTERM in the signal_fd, so we block
       them. */
    sigemptyset(&sigmask);
    sigaddset(&sigmask, SIGINT);
    sigaddset(&sigmask, SIGTERM);

    if (sigprocmask(SIG_BLOCK, &sigmask, NULL) < 0)
    {
        fprintf(stderr,
                "Couldn't block signals: '%s'\n",
                strerror(errno));
        return -1;
```

```
    }

    /* Get new FD to read signals from it */
    if ((signal_fd = signalfd(-1, &sigmask, 0)) < 0)
    {
        fprintf(stderr,
                "Couldn't setup signal FD: '%s'\n",
                strerror(errno));
        return -1;
    }

    return signal_fd;
}

int main(int argc,
         const char **argv)
{
    int signal_fd;
    int fanotify_fd;
    struct pollfd fds[FD_POLL_MAX];

    /* Input arguments... */
    if (argc < 2)
    {
        fprintf(stderr, "Usage: %s directory1 [directory2 ...]\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* Initialize signals FD */
    if ((signal_fd = initialize_signals()) < 0)
```

```
{
    fprintf(stderr, "Couldn't initialize signals\n");
    exit(EXIT_FAILURE);
}

/* Initialize fanotify FD and the marks */
if ((fanotify_fd = initialize_fanotify(argc, argv[0])) < 0)
{
    fprintf(stderr, "Couldn't initialize fanotify\n");
    exit(EXIT_FAILURE);
}

/* Setup polling */
fds[FD_POLL_SIGNAL].fd = signal_fd;
fds[FD_POLL_SIGNAL].events = POLLIN;
fds[FD_POLL_FANOTIFY].fd = fanotify_fd;
fds[FD_POLL_FANOTIFY].events = POLLIN;

/* Now loop */
for (;;)
{
    /* Block until there is something to be read */
    if (poll(fds, FD_POLL_MAX, -1) < 0)
    {
        fprintf(stderr,
                "Couldn't poll(): '%s'\n",
                strerror(errno));
        exit(EXIT_FAILURE);
    }
```

```
/* Signal received? */
if (fds[FD_POLL_SIGNAL].revents & POLLIN)
{
    struct signalfd_siginfo fdsi;

    if (read(fds[FD_POLL_SIGNAL].fd,
            &fdsi,
            sizeof(fdsi)) != sizeof(fdsi))
    {
        fprintf(stderr,
                "Couldn't read signal, wrong size read\n");
        exit(EXIT_FAILURE);
    }

    /* Break loop if we got the expected signal */
    if (fdsi.ssi_signo == SIGINT ||
        fdsi.ssi_signo == SIGTERM)
    {
        break;
    }

    fprintf(stderr,
            "Received unexpected signal\n");
}

/* fanotify event received? */
if (fds[FD_POLL_FANOTIFY].revents & POLLIN)
{
    char buffer[FANOTIFY_BUFFER_SIZE];
    ssize_t length;
```

```
            /* Read from the FD. It will read all events available up to
             * the given buffer size. */
            if ((length = read(fds[FD_POLL_FANOTIFY].fd,
                               buffer,
                               FANOTIFY_BUFFER_SIZE)) > 0)
            {
                struct fanotify_event_metadata *metadata;

                metadata = (struct fanotify_event_metadata *)buffer;
                while (FAN_EVENT_OK(metadata, length))
                {
                    event_process(metadata, fanotify_fd);
                    if (metadata->fd > 0)
                        close(metadata->fd);
                    metadata = FAN_EVENT_NEXT(metadata, length);
                }
            }
        }
    }

    /* Clean exit */
    shutdown_fanotify(fanotify_fd);
    shutdown_signals(signal_fd);

    printf("Exiting fanotify example...\n");

    return EXIT_SUCCESS;
}
```

B.3 Adapted code for fanotify

```
#define _GNU_SOURCE /* Needed to get O_LARGEFILE definition */
#include <errno.h>
#include <fcntl.h>
#include <limits.h>
#include <poll.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/fanotify.h>
#include <unistd.h>

/* Read all available fanotify events from the file descriptor 'fd' */

static void
handle_events(int fd)
{
    const struct fanotify_event_metadata *metadata;
    struct fanotify_event_metadata buf[200];
    ssize_t len;
    char path[PATH_MAX];
    ssize_t path_len;
    char procfd_path[PATH_MAX];
    struct fanotify_response response;
    char *str;

    /* Loop while events can be read from fanotify file descriptor */

    for (;;)
    {
```

```
/* Read some events */

len = read(fd, buf, sizeof(buf));
if (len == -1 && errno != EAGAIN)
{
    perror("read");
    exit(EXIT_FAILURE);
}

/* Check if end of available data reached */

if (len <= 0)
    break;

/* Point to the first event in the buffer */

metadata = buf;

/* Loop over all events in the buffer */

while (FAN_EVENT_OK(metadata, len))
{
    /* Check that run-time and compile-time structures match */

    if (metadata->vers != FANOTIFY_METADATA_VERSION)
    {
        fprintf(stderr,
                "Mismatch of fanotify metadata version.\n");
        exit(EXIT_FAILURE);
    }
```

```c
switch (metadata->mask)
{
case FAN_ATTRIB:
    str = "ATTRIB";
    break;
case FAN_CREATE:
    str = "CREATE";
    break;
case FAN_DELETE:
    str = "DELETE";
    break;
case FAN_DELETE_SELF:
    str = "DELETE SELF";
    break;
case FAN_MOVED_FROM:
    str = "MOVED FROM";
    break;
case FAN_MOVED_TO:
    str = "MOVED TO";
    break;
case FAN_MOVE_SELF:
    str = "MOVE SELF";
    break;
case FAN_MODIFY:
    str = "MODIFY";
    break;
default:
    /* Fallback : use question mark for unrecognized bit */
    str = "?";
```

```c
        break;
    }

    printf("%s\n", str);

    /* metadata->fd contains either FAN_NOFD, indicating a
       queue overflow, or a file descriptor (a nonnegative
       integer). Here, we simply ignore queue overflow. */

    if (metadata->fd >= 0)
    {

        /* Handle open permission event */

        if (metadata->mask & FAN_OPEN_PERM)
        {
            printf("FAN_OPEN_PERM: ");

            /* Allow file to be opened */

            response.fd = metadata->fd;
            response.response = FAN_ALLOW;
            write(fd, &response, sizeof(response));
        }

        /* Handle closing of writable file event */

        if (metadata->mask & FAN_CLOSE_WRITE)
            printf("FAN_CLOSE_WRITE: ");
```

```
                /* Retrieve and print pathname of the accessed file */

                snprintf(procfd_path, sizeof(procfd_path),
                        "/proc/self/fd/%d", metadata->fd);
                path_len = readlink(procfd_path, path,
                                  sizeof(path) - 1);
                if (path_len == -1)
                {
                    perror("readlink");
                    exit(EXIT_FAILURE);
                }

                path[path_len] = '\0';
                printf("File %s\n", path);

                /* Close the file descriptor of the event */

                close(metadata->fd);
            }

            /* Advance to next event */

            metadata = FAN_EVENT_NEXT(metadata, len);
        }
    }
}

int main(int argc, char *argv[])
{
    char buf;
```

```
int fd, poll_num;
nfds_t nfds;
struct pollfd fds[2];

/* Check mount point is supplied */

if (argc != 2)
{
    fprintf(stderr, "Usage: %s MOUNT\n", argv[0]);
    exit(EXIT_FAILURE);
}

printf("Press enter key to terminate.\n");

/* Create the file descriptor for accessing the fanotify API */

fd = fanotify_init(FAN_CLASS_NOTIF | FAN_REPORT_FID, O_RDWR);
if (fd == -1)
{
    perror("fanotify_init");
    exit(EXIT_FAILURE);
}

/* Mark the mount for:
   - permission events before opening files
   - notification events after closing a write-enabled
     file descriptor */
printf("%s\n", argv[1]);

if (fanotify_mark(fd, FAN_MARK_ADD,
```

```
                FAN_CREATE | FAN_DELETE | FAN_MODIFY | FAN_ATTRIB |
                    FAN_DELETE_SELF | FAN_MOVED_FROM | FAN_MOVED_TO |
                    FAN_MOVE_SELF | FAN_ONDIR | FAN_EVENT_ON_CHILD,
                AT_FDCWD,
                argv[1]) == -1)
{

    perror("fanotify_mark");
    exit(EXIT_FAILURE);
}

/* Prepare for polling */

nfds = 2;

/* Console input */

fds[0].fd = STDIN_FILENO;
fds[0].events = POLLIN;

/* Fanotify input */

fds[1].fd = fd;
fds[1].events = POLLIN;

/* This is the loop to wait for incoming events */

printf("Listening for events.\n");

while (1)
```

```
{
    poll_num = poll(fds, nfds, -1);
    if (poll_num == -1)
    {
        if (errno == EINTR) /* Interrupted by a signal */
            continue;      /* Restart poll() */

        perror("poll"); /* Unexpected error */
        exit(EXIT_FAILURE);
    }

    if (poll_num > 0)
    {
        if (fds[0].revents & POLLIN)
        {

            /* Console input is available: empty stdin and quit */
            while (read(STDIN_FILENO, &buf, 1) > 0 && buf != '\n')
                continue;
            break;
        }

        if (fds[1].revents & POLLIN)
        {

            /* Fanotify events are available */

            handle_events(fd);
        }
    }
```

```
    }

    printf("Listening for events stopped.\n");
    exit(EXIT_SUCCESS);
}
```

B.4 Postgres Example Code

```
/*
 * src/test/examples/testlibpq.c
 *
 *
 * testlibpq.c
 *
 *      Test the C version of libpq, the PostgreSQL frontend library.
 */
#include <stdio.h>
#include <stdlib.h>
#include <libpq-fe.h>

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int main(int argc, char **argv)
{
```

```c
const char *conninfo;
PGconn *conn;
PGresult *res;
int nFields;
int i,
    j;

/*
 * If the user supplies a parameter on the command line, use it as the
 * conninfo string; otherwise default to setting dbname=postgres and using
 * environment variables or defaults for all other connection parameters.
 */
if (argc > 1)
    conninfo = argv[1];
else
    conninfo = "host=192.168.163.129 port=5432 dbname=tesedb user=postgres
        password=admin connect_timeout=10";

/* Make a connection to the database */
conn = PQconnectdb(conninfo);

/* Check to see that the backend connection was successfully made */
if (PQstatus(conn) != CONNECTION_OK)
{
    fprintf(stderr, "%s", PQerrorMessage(conn));
    exit_nicely(conn);
}

/* Set always-secure search path, so malicious users can't take control. */
res = PQexec(conn,
```

```
            "SELECT pg_catalog.set_config('search_path', '', false)");
if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

/*
 * Should PQclear PGresult whenever it is no longer needed to avoid memory
 * leaks
 */
PQclear(res);

/*
 * Our test case here involves using a cursor, for which we must be inside
 * a transaction block. We could do the whole thing with a single
 * PQexec() of "select * from pg_database", but that's too trivial to make
 * a good example.
 */

/* Start a transaction block */
res = PQexec(conn, "BEGIN");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "BEGIN command failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}
PQclear(res);
```

```
/*
 * Fetch rows from pg_database, the system catalog of databases
 */
res = PQexec(conn, "DECLARE myportal CURSOR FOR select * from
   pg_database");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "DECLARE CURSOR failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}
PQclear(res);

res = PQexec(conn, "FETCH ALL in myportal");
if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "FETCH ALL failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

/* first, print out the attribute names */
nFields = PQnfields(res);
for (i = 0; i < nFields; i++)
    printf("%-15s", PQfname(res, i));
printf("\n\n");

/* next, print out the rows */
for (i = 0; i < PQntuples(res); i++)
```

```
    {
        for (j = 0; j < nFields; j++)
            printf("%-15s", PQgetvalue(res, i, j));
        printf("\n");
    }

    PQclear(res);

    /* close the portal ... we don't bother to check for errors ... */
    res = PQexec(conn, "CLOSE myportal");
    PQclear(res);

    /* end the transaction */
    res = PQexec(conn, "END");
    PQclear(res);

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    return 0;
}
```

B.5 Shannon Entropy Code

```
/*
 Shannon entropy calculation
 $ cc -Wall shent.c -o shent -lm
 $ ./shent shent.c
*/
```

```c
#include <stdio.h>
#include <stdint.h>
#include <math.h>

int main(int argc, char *argv[])
{
    uint64_t map[256];
    size_t i;
    FILE *f;
    long int flen;
    double info = 0.0;

    if (argc < 2)
    {
        fprintf(stderr, "Usage: %s file\n", argv[0]);
        return 1;
    }
    printf("%s\n", argv[1]);

    f = fopen(argv[1], "r");
    if (!f)
    {
        fprintf(stderr, "Can't open %s\n", argv[1]);
        return 1;
    }

    for (i = 0; i < (sizeof(map) / sizeof(map[0])); i++)
    {
        map[i] = 0;
    }
```

```
while (!feof(f))
{
    char buf[1024 * 8];
    size_t r;

    r = fread(buf, 1, sizeof(buf), f);
    if (r == 0)
        break;
    for (i = 0; i < r; i++)
    {
        size_t index = buf[i];
        map[index]++;
    }
}
flen = ftell(f);
fclose(f);

for (i = 0; i < (sizeof(map) / sizeof(map[0])); i++)
{
    double freq;

    if (map[i] == 0)
        continue;
    freq = (double)map[i] / flen;
    info += freq * log2(freq);
}

info = -info;
```

```
    printf("%f\n", info);

    return 0;
}
```

B.6 Adapted code from fanotify with MODIFY event

```
#define _GNU_SOURCE
#include <errno.h>
#include <fcntl.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/fanotify.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>
#include <libpq-fe.h>
#include <math.h>
#define ARRAY_SIZE(a) (sizeof(a) / sizeof(*a))
#define BUF_SIZE (1024 * 64)

typedef struct
{
    int EventSize;
    int Version;
    int Reserved;
    int MetadataSize;
```

```c
    int Mask;
    char MaskName[250];
    long int FD;
    int PID;
    char NameProcess[250];
    char File[1000];
    float Entropy;
    float FileBytes;
    char MagicBytes[04];
} Dados;

static void exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

static void storeLogData(PGconn *conn, Dados dados)
{
    PGresult *res;
    char buf[5000] = {};
    char query_string[] = {"INSERT INTO public.logs (eventsize, version,
        reserved, metadatasize, fd, mask, maskname, pid, pidname, file,
        entropy, filebytes, magicbytes)
        VALUES('%d','%d','%d','%d','%li','%d','%s','%d','%s','%s', '%f', '%f',
        '\\x%02x'::bytea || '\\x%02x'::bytea || '\\x%02x'::bytea ||
        '\\x%02x'::bytea)"};

    sprintf(buf, query_string, dados.EventSize, dados.Version,
            dados.Reserved, dados.MetadataSize, dados.FD, dados.Mask,
```

```
            dados.MaskName, dados.PID, dados.NameProcess, dados.File,
            dados.Entropy, dados.FileBytes, dados.MagicBytes[0],
               dados.MagicBytes[1], dados.MagicBytes[2], dados.MagicBytes[3]);

    // PGRES_COMMAND_OK;
    res = PQexec(conn, buf);

    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        printf("INSERT command failed\n");
        printf("%s\n", buf);
        PQclear(res);
        exit_nicely(conn);
        exit(EXIT_FAILURE);
    }

    PQclear(res);
}

double calculate_entropy(char *buf, size_t size)
{
    uint64_t map[256];
    size_t i;
    double info = 0.0;

    for (i = 0; i < (sizeof(map) / sizeof(map[0])); i++)
    {
        map[i] = 0;
    }
```

```c
    for (int i = 0; i < size; i++)
    {
        size_t index = buf[i];
        map[index]++;
    }

    for (i = 0; i < (sizeof(map) / sizeof(map[0])); i++)
    {
        double freq;

        if (map[i] == 0)
            continue;
        freq = (double)map[i] / size;
        info += freq * log2(freq);
    }

    info = -info;

    return info;
}

static char *readFirstLine(const char *path)
{
    FILE *fp = fopen(path, "r");
    char *line = NULL;
    size_t sz;
    /* Real buffersizeof returned string, unused,
        cannot be null */
    ssize_t n;
```

```
    if (fp != NULL)
    {
        /* XXX : Empty file causes getline(3) return 1, errno 0,
            the line is an empty string */

        n = getline(&line, &sz, fp);
        if (n > 0 && line[n - 1] == '\n')
        {
            line[--n] = '\0';
        }
        (void)fclose(fp);
    }

    return line;
}

static char *processName(pid_t pid)
{
    char path[ARRAY_SIZE("/proc/4294967295/comm") + 1];
    int n;

    if (pid <= 0)
    {
        perror("read");
        exit(EXIT_FAILURE);
    }

    n = snprintf(path, sizeof(path), "/proc/%d/comm", pid);

    if (n <= 0)
```

```
    {
        perror("n");
        exit(EXIT_FAILURE);
    }

    return readFirstLine(path);
}

static inline char *fanotifyMaskStr(uint64_t mask)
{
    char *str;
    if (mask == 0)
    {
        perror("mask ");
        /* Unexpected error */
        exit(EXIT_FAILURE);
    }

    switch (mask)
    {
    case FAN_ATTRIB:
        str = "ATTRIB";
        break;
    case FAN_CREATE:
        str = "CREATE";
        break;
    case FAN_DELETE:
        str = "DELETE";
        break;
    case FAN_DELETE_SELF:
```

```
        str = "DELETE SELf ";
        break;
    case FAN_MOVED_FROM:
        str = "MOVED FROM";
        break;
    case FAN_MOVED_TO:
        str = "MOVED TO";
        break;
    case FAN_MOVE_SELF:
        str = "MOVE SELF";
        break;
    case FAN_MODIFY:
        str = "MODIFY";
        break;
    case FAN_OPEN:
        str = "OPEN";
        break;
    default:
        /* Fallback : use question mark for unrecognized bit */
        str = "?";
        break;
    }
    return str;
}

int main(int argc, char **argv)
{
    int fan;
    char buf[4096];
    char path[PATH_MAX];
```

```c
ssize_t buflen;
struct fanotify_event_metadata *metadata;
char *pName;
char *events;
const char *conninfo;
PGconn *conn;
ssize_t path_len;
char procfd_path[PATH_MAX];
char bufFile[BUF_SIZE];
char magicByte[4];

if (argc != 2)
{
    fprintf(stderr, "Usage : %s /dir\n", argv[0]);
    exit(EXIT_FAILURE);
}

conninfo = "host=192.168.163.129 port=5432 dbname=tesedb user=postgres
    password=admin connect_timeout=10";
conn = PQconnectdb(conninfo);

/* Check to see that the backend connection was successfully made */
if (PQstatus(conn) != CONNECTION_OK)
{
    fprintf(stderr, "%s", PQerrorMessage(conn));
    exit_nicely(conn);
    exit(EXIT_FAILURE);
}

fan = fanotify_init(FAN_CLASS_CONTENT, O_RDONLY);
```

```
if (fan == -1)
{
    perror(" fanotify_init ");
    exit(EXIT_FAILURE);
}

int ret = fanotify_mark(fan,
                        FAN_MARK_ADD | FAN_MARK_MOUNT,
                        FAN_MODIFY | FAN_ONDIR | FAN_EVENT_ON_CHILD,
                        AT_FDCWD, argv[1]);

if (ret == -1)
{
    perror("fanotify_mark ");
    exit(EXIT_FAILURE);
}

while (1)
{
    buflen = read(fan, buf, sizeof(buf));
    metadata = (struct fanotify_event_metadata *)&buf;
    for (;
         FAN_EVENT_OK(metadata, buflen);
         metadata = FAN_EVENT_NEXT(metadata, buflen))
    {
        snprintf(procfd_path, sizeof(procfd_path), "/proc/self/fd/%d",
            metadata->fd);

        path_len = readlink(procfd_path, path,
                            sizeof(path) - 1);
```

```
if (path_len == -1)
{
    perror("readlink");
    exit(EXIT_FAILURE);
}

path[path_len] = '\0';

if (strchr(path, '.'))
{
    ssize_t size = read(metadata->fd, bufFile, BUF_SIZE);
    if (size == -1)
    {
        perror("read");
        continue;
    }
    else
    {

        events = fanotifyMaskStr(metadata->mask);
        if (!(events == "?"))
        {
            pName = processName(metadata->pid);
            double entropy = calculate_entropy(bufFile, size);
            memcpy(magicByte, bufFile, 4);

            Dados dados;
            dados.EventSize = metadata->event_len;
            dados.Version = metadata->vers;
            dados.Reserved = metadata->reserved;
```

```
                        dados.MetadataSize = metadata->metadata_len;
                        dados.FD = metadata->fd;
                        dados.Mask = metadata->mask;
                        strcpy(dados.MaskName, events);
                        dados.PID = metadata->pid;
                        strcpy(dados.NameProcess, pName);
                        strcpy(dados.File, path);
                        dados.Entropy = entropy;
                        dados.FileBytes = size;
                        strcpy(dados.MagicBytes, magicByte);
                        storeLogData(conn, dados);
                    }
                }
            }

            close(metadata->fd);
            metadata = FAN_EVENT_NEXT(metadata, buflen);
        }
    }
}
```