University of Zurich[UZH]

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

Intelligent Framework to Detect Ransomware Affecting Linux-based and Resource-constrained Devices

Dennis Shushack
Zurich, Switzerland
Student ID: 15-703-341

Supervisor: Dr. Alberto Huertas Celdrán, Jan von der Assen
Date of Submission: August 12, 2022

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

ifi

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

Abstract Deutsch

Das Internet of Things (IoT), ein expandierendes Netzwerk bestehend aus miteinander verbundenen Geräten, gewinnt in verschiedenen Branchen immer mehr an Bedeutung.

Diese Technologie kann unser Leben positiv beeinflussen und erhebliche wirtschaftliche Vorteile mit sich bringen. Beispielsweise Crowdsensing-Plattformen wie ElectroSense, welche IoT-Geräte mit Sensoren zur Frequenzüberwachung verwenden, haben sich als effizient, kostengünstig und skalierbar erwiesen. Obwohl diese ressourcenbeschränkten IoT-Geräte mit zahlreichen Vorteilen verbunden sind, sind sie oft anfällig für Cyberangriffe. Demnach könnte Ransomware eine ernsthafte Bedrohung für das IoT-Ökosystem darstellen. Die ElectroSense Plattform, welche IoT-Sensoren verwendet, um Funkfrequenzdaten zu sammeln und auszutauschen, könnte einem solchen Angriff zum Opfer fallen. Dies würde zu schwerwiegenden Sensorausfällen führen, welche den Zugang zu Sensordaten erheblich beeinträchtigen könnten.

Algorithmen für maschinelles und tiefes Lernen, welche Geräteverhaltensdaten nutzen, wurden als vielversprechende Ransomware-Erkennungs- und Klassifizierungstechniken identifiziert. Die meisten Erkennungsframeworks, die diese Technologien nutzen, wurden jedoch für Windows-basierte Systeme entwickelt, die im Allgemeinen über mehr Ressourcen verfügen als IoT-Geräte. Diese Lösungen sind oftmals nicht geeignet für Crowdsensing-Plattformen, welche ressourcenbeschränkte Systeme verwenden. Neben maschinellem Lernen haben sich Ransomware Richtlinien (Policies) als eine ressourceneffiziente und effektive Methode zur Erkennung und Klassifizierung von Ransomware erwiesen, bringen jedoch gewisse Einschränkungen mit sich.

Daher wird in dieser Arbeit ein Ransomware Erkennungs- und Klassifikationsframework basierend auf maschinellen und tiefen Lernen entwickelt und getestet. Dieses Framework nutzt drei unterschiedliche Geräteverhaltens-Quellen zur Erkennung bzw. Klassifikation von Ransomware, welche ressourcenbeschränkte ElectoSense-Sensoren angreift. Das entwickelte Tool präsentiert eine effiziente, skalierbare und datenschutzfreundliche Lösung zur Identifizierung von Zero-Day-Ransomware Angriffen und zur Klassifizierung verschiedener Ransomwaretypen. Darüber hinaus werden reale Ransomware-Angriffsszenarien verwendet, um die Effektivität des Systems zu testen.

Abstract English

The Internet of Things (IoT), a network of interconnected devices, has been growing and gaining traction in various industries. This technology can impact our lives while also providing significant economic benefits. For example, crowdsensing platforms such as ElectroSense that use sensor-equipped IoT devices to collect and share spectrum monitoring data have proven efficient, cost-effective, and scalable. However, although these resource-constrained IoT devices provide numerous benefits, they are also vulnerable to cyberattacks. As a result, ransomware could severely threaten the IoT ecosystem. ElectroSense, which employs IoT device sensors, may fall victim to such an attack, resulting in operational problems and sensor data unavailability.

Machine and deep learning algorithms using behavioral data have been identified as promising ransomware detection and classification techniques. However, most detection frameworks that utilize these technologies have been developed for Windows-based systems, which generally have more resources than IoT devices. As a result, these solutions may not be well-suited for crowdsensing platforms which utilize resource-constrained components. In addition, while ransomware policies are effective and resource efficient in detecting and classifying ransomware, they do have some limitations.

This thesis, therefore, proposes to develop and test a machine and deep learning-based framework that utilizes three different behavioral sources to detect and classify ransomware impacting resource-constrained ElectoSense sensors. This framework will employ an efficient, scalable, and data-protective approach to identify zero-day ransomware attacks and classify various ransomware strains. In addition, real-world ransomware attack scenarios are utilized to test the platform's effectiveness.

Acknowledgments

First and foremost, I would like to thank my supervisor, Dr. Alberto Huertas Celdrán, for his guidance and support throughout this thesis. His knowledge of cybersecurity contributed tremendously to the successful completion of the project. Secondly, I would like to thank Prof. Dr. Burkhard Stiller for allowing me to complete my thesis in the Communication Systems Group. In addition, my thanks go out to Jan von der Assen for his insightful feedback.

Last but not least, this work would not have been possible without the support of my family and friends, who were always there for me.

Contents

Chapter 1

Introduction

1.1 Motivation

The Internet of Things (IoT) has received a great deal of attention in recent years. This expanding network of resource-constrained devices will significantly affect our daily lives and positively impact the economy and society. By 2022, over 18 billion IoT devices are predicted to be in use, bringing benefits to transportation, manufacturing, health care, retail services, and other industries [60, 55, 43]. With the rapid adoption of IoT, crowdsensing platforms have gained traction by allowing users to share data acquired through sensor-equipped machines [16].

Spectrum sensing, the process of monitoring the electromagnetic environment with sensors, has grown in prominence due to its broad applicability in consumer, regulatory, and military applications. An example of a crowdsensing activity that uses IoT sensors is the ElectroSense platform, which offers an inexpensive and reliable way of measuring and analyzing spectrum data. To contribute to this project requires the purchase of a low-cost, resource-constraint radio sensor. Spectrum data collected from the sensors are sent to a central backend server and displayed to users on the ElectroSense website for further analysis [24, 46].

The use of IoT devices, however, is not without its drawbacks. Poor security, lax device management procedures, and software-level vulnerabilities are common and can be exploited by cybercriminals. As such, ransomware, a type of malware that encrypts files to extort money from victims, poses a severe threat to the IoT ecosystem [33, 28].

Two emerging ransomware families that target different systems, including Linux servers, are Hive and LockBit. Leveraging a ransomware-as-a-service model, they have caused havoc in various industries, including healthcare, technology, and education. An example of such an attack occurred on August 15, 2021. The Hive ransomware disrupted the daily operations of three hospitals by encrypting their IT infrastructure, forcing them to cancel essential surgical procedures and examinations [49, 50].

The ElectroSense platform developed around IoT devices running a distribution of Linux could fall victim to such a ransomware attack. Infected sensors would have their data encrypted and not function properly, significantly impacting the platform's overall operation and accessibility to spectrum data.

Only a few options are available today to effectively identify a ransomware infection on these systems. Using manually defined state-of-the-art policies for their detection is one example. However, while offering a lightweight solution for detecting ransomware attacks on resource-constrained devices, specific limitations exist with this approach. For instance, the process of creating policies is time-consuming and requires specialized expertise [17]. In addition, the following challenges and obstacles remain when evaluating related ransomware detection research papers:

- Only a limited amount of research focuses on detecting and classifying ransomware on resource constraint devices running Linux. Furthermore, most papers only examine detection performance and ignore the issue of resource utilization.
- Few research studies assess the effectiveness of detecting and classifying ransomware using different device behavioral sources.
- There appears to be insufficient research comparing the performance of a policy-based method versus a Machine Learning anomaly-based strategy in ransomware detection/classification and resource utilization.
- Although various ransomware detection and classification frameworks have been proposed, the majority of these do not address issues such as data protection, resource utilization, and distribution. Thus, there is the need to invest additional resources to design, develop and implement the ransomware detection and classification framework.

With the ever-increasing use of Machine Learning (ML) and Deep Learning (DL) in cybersecurity, there is an opportunity to utilize these tools in the IoT technology space. Furthermore, device behavioral fingerprinting, a technique for analyzing the behavior of a device, has been identified as a valuable way of recognizing cyberattacks [54]. This thesis will therefore focus on detecting and classifying ransomware on the ElectroSense platform with ML and DL using three different behavioral sources.

1.2 Description of Work

To address the previously mentioned challenges, an ML/DL-based framework to detect and classify ransomware was designed and implemented as a proof-of-concept. This framework monitors three different behavioral dimensions of an ElectroSense sensor for detecting and classifying heterogeneous ransomware. In addition, this solution is suitable for IoT spectrum sensors, integrates into the ElectroSense architecture, and does not impact the sensor's normal spectrum data monitoring process.

As part of this work, several experiments were carried out to evaluate the framework's ransomware detection and classification capabilities, as well as its resource utilization. These experiments were performed by utilizing an ElectroSense sensor (Raspberry Pi 3 Model B) infected with three different ransomware samples: DarkRadiation, RAASNet, and Ransomware-PoC.

DarkRadiation targets Linux-based systems using the Telegram messaging application as its command-and-control server. RAASNet is a cross-platform ransomware-as-a-service allowing users to create customized ransomware payloads easily. Finally, Ransomware-PoC is a Python based proof-of-concept ransomware that can be deployed on different operating systems.

The frameworks' effectiveness and feasibility were determined by comparing the obtained results to a policy-based ransomware detection approach. Furthermore, a detailed analysis of the detection/classification performance using different behavioral sources and ML/DL algorithms is provided in this thesis.

1.3 Thesis Outline

This thesis is structured into topic-specific chapters. Chapter 2 introduces the reader to the concept of ransomware, ML, and anomaly detection. This background knowledge will assist the reader in comprehending the current status of research in ransomware detection, as provided in Chapter 3. Chapter 4 will then introduce the reader to the Electrosense platform and its configuration as used in this thesis. In addition, a description of the three ransomware samples is provided. Chapters 5 and 6 outline the ransomware detection and classification framework and give insight into its implementation. Chapter 7, Experiment and Analysis, will present the thesis findings, which are summarized and concluded in the last chapter of this document.

Chapter 2

Background

Ransomware has evolved greatly since its first appearance in 1989. Disguised as a questionnaire about the Acquired Immunodeficiency Syndrome Virus (AIDS) and distributed by mail via floppy disks, the Trojan horse AIDS, attempted to extort money from unwitting computer users [11]. With the advent of the Internet, ransomware became much more common, giving hackers a tool for wider distribution and leading to higher profit margins [53]. Recent trends such as cryptocurrencies, the use of IoT devices, the increasing interconnectivity of systems, and the growing use of social media platforms that expose sensitive personal data are leading to new types of ransomware attacks as they provide hackers with new attack opportunities [66].

This chapter will briefly introduce malware and ransomware. Then, the different methods used to detect ransomware and the concept of behavioral fingerprinting are presented. Finally, an introduction to Machine Learning (ML) and Deep Learning (DL) and their application in cybersecurity is given.

2.1 Malware

Malware is software that has a malicious purpose. It generally refers to any piece of code that has been intentionally modified, added, or removed. Its primary goal is to cause harm by altering the functionality of a system [37]. Cybercriminals usually develop this malicious software to steal user data or significantly damage or destroy the target system. There are many different types of malware with a variety of nefarious purposes. These include computer viruses, spyware, worms, adware, Trojan viruses and ransomware, among others [65].

2.1.1 Classification of Malware

As malware becomes more sophisticated, it becomes more difficult to classify. Traditional malicious software typically had a specific function (i.e., encrypting files). Current malware, however, can have multiple negative characteristics at the same time and employs

strategies to conceal itself. This fact makes detecting modern malware far more complex [6]. A simple approach suggested by [57] is to categorize malware based on two different features: Propagation and Payload.

Propagation refers to the mechanism malware uses to spread in order to achieve its goal. After gaining access to the system, the malicious software will deploy its payload. The payload is the part of the malware that causes the intended malicious behavior on the system. The different ways malware spreads include social engineering attacks, exploiting vulnerabilities in software, and infection by viruses that spread to other systems. Stealing information, corrupting files, hiding its presence, or hijacking the server to join a botnet are examples of payload actions performed.

2.2 Ransomware

Ransomware, as the name suggests, combines two words: ransom and malware. It is a type of malicious software demanding a ransom in exchange for some system functionality that has either been removed or disabled maliciously. Most ransomware use file encryption on the victim's device as a form of extortion [28].

2.2.1 Types of Ransomware

Yaqoob et al. [66] suggest a classification of three different ransomware types:

- Crypto ransomware
- Locker ransomware
- Hybrid ransomware

Crypto ransomware scans the infected system for relevant files and encrypts them using symmetric, asymmetric, or hybrid key cryptography algorithms such as AES, RSA, or a combination of both [9, 10]. *Locker ransomware*, on the other hand, employs a different attack methodology, restricting access to or altering system functionalities. Finally, *Hybrid ransomware* combines the two methods mentioned above, using encryption and a locking mechanism for a more devastating attack [66].

In all cases, the victim must pay a ransom to return their system to its 'pre-attack' configuration.

2.2.2 Stages of a ransomware attack

All three ransomware types can affect the system differently. However, as noted earlier, file encryption is modern ransomware's most prevalent attack behavior. A typical attack procedure found during a Crypto ransomware infection is discussed in a threat report by Exabeam [61].

The report divides the attack into six different stages, namely

1. Campaign phase
2. Infection phase
3. Staging phase
4. Scanning phase
5. Encryption phase
6. Payday phase

Figure 2.1: Main Stages of the Ransomware Kill Chain [61]

The goal of the *(1) Campaign phase* is to spread and distribute a malicious dropper. This code can be initiated in various ways, including exploiting system- and application-level vulnerabilities, a drive-by download, or unknowingly downloading from an email attachment.

In the *(2) Infection phase*, after gaining access to the system, the dropper communicates home to a command-and-control server (C2) operated by the cybercriminals. From there, it downloads a hidden malicious executable. The server then sends a command to remove the dropper from the target system and executes the payload.

In the *(3) Staging phase*, the ransomware prepares itself before encrypting any files. Verifying user privileges, configurations, proxy settings, moving to a different folder, etc., are some actions the ransomware performs to ensure a seamless operation in the later stages of the attack. Finally, the ransomware communicates with the C2 server to evaluate whether the infected system is suitable for encryption and to negotiate the public key.

In the *(4) Scanning phase*, files from system directories, network file shares, and cloud storage repositories are scanned and mapped based on a list of file extensions. Depending on the volume of data available, the scanning and mapping can take several minutes or even hours to complete.

The *(5) Encryption phase* is a three-step process for each targeted file. First, the ransomware has to obtain the file from the system. It then encrypts its content and uploads it to the original location. Only the encrypted version remains on the system, as the original file is deleted in the third step.

Each encrypted directory also contains a ransom note with payment instructions. Depending on the number of files to be encrypted, this process can take a significant amount of time.

In the last phase, also known as the *(6) Payday phase* the victim is presented with a pop-up window containing the payment instructions. Some ransomware variants include a timer or a countdown to create a sense of urgency. A decryption program or a key is sent to the victim if he chooses to pay the ransom.

2.2.3 Linux ransomware

This thesis deals with devices running a distribution of Linux. As such, the following properties relevant to most Linux ransomware attacks will be reviewed [21]:

- Exploitation of vulnerabilities
- Privilege escalation
- Open-source nature of the operating system

Linux ransomware, in general, exploits vulnerabilities and flaws in the operating system and application software to obtain access to the device and its files. These exposures include vulnerabilities in outdated versions of the Linux operating system, application software (i.e., Exim), SQL injections, and deficiencies in middleware software such as WordPress or Drupal CMS systems. Linux ransomware often can escalate system privileges (i.e., obtain root privileges) on the infected device. Escalated privileges can lead to a more impactful attack, as the ransomware can access more parts of the system. Additionally, since Linux is open-source, anyone can contribute to the project and view its source code. Fortunately, the user community quickly finds and patches most security flaws before cybercriminals can exploit them. However, systems remain vulnerable if security patches are not applied regularly.

2.2.4 Ransomware in IoT

The Internet of Things (IoT) describes a network of devices/things that collect and exchange data with other devices. They are usually embedded with other technology, such as sensors or communication hardware [64]. The devices that make up this network are usually resource-constrained in terms of their processing capabilities, memory, and power consumption [55]. Due to the rapid adoption of IoT in different fields, discussions of potential ransomware attacks have attracted more attention. Ransomware can be a significant

problem as efficient device management becomes more complicated with the introduction of more IoT devices. New vulnerabilities in IoT software and weak IoT security processes provide possible entry points for different malware types. Outdated legacy systems connected to the network can also be a potential source of attack. In addition, as connectivity between devices and systems increases, there is a risk of ransomware spreading to more critical parts of the infrastructure [33].

2.3 Malware detection methods

2.3.1 Classification of malware detection

While there are various ways to classify malware detection, the authors of [31] and [35] provide a simple, straightforward approach. The method they employ determines the malware classification by differentiating between two general methods for malware detection (Anomaly-based and signature-based detection). Note, Specification-based detection is a specialized subset of anomaly-based detection. Dynamic, static, and hybrid analysis are the three subcategories/approaches for each technique.

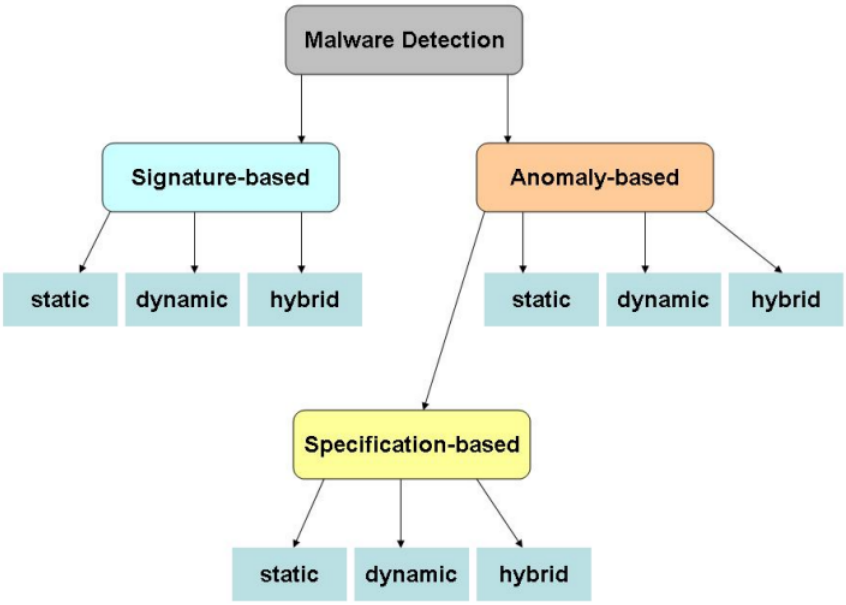

Figure 2.2: Malware detection classification [31]

Signature-based malware detection

A signature refers to some feature of the malware that uniquely identifies it [6]. These signatures can be encountered in the malware's source code, commonly represented by some sequences of bytes. Signature-based malware detection is an approach where malware

scanners (i.e., antivirus systems) use a knowledge base of viral signatures to determine if a program is hazardous or not [63, 31, 67].

As signatures are mainly created by humans and take a long time to develop, this approach is not ideal for all types of malware detection. For example, zero-day attacks present a significant issue, as they cannot be detected due to the missing signatures [31].

Anomaly-based detection

Anomaly-based detection employs a different technique for recognizing malware. This approach utilizes a two-step process: The first step is to create a baseline model of the device's typical normal behavior in a training/learning phase. Then, in the second phase (the monitoring/detection phase), a potential malware attack is recognized if the device's behavior significantly differs from the previously established model. The ability to identify zero-day attacks is a significant benefit of anomaly-based detection. However, a lengthy data collecting phase, high false-positive rates triggered by normal unseen device behavior, and determining the relevant features to be monitored, are some of the disadvantages of this method [31, 34].

Specification-based detection

Specification-based malware detection is similar to anomaly-based detection. However, it uses a set of rules to determine if a program is malicious or not. These rules are developed by manually specifying the normal/valid behavior of the system. This technique mitigates the typical high false-positive rate experienced by anomaly-based detection. It is, however, time-consuming to approximate all legitimate behaviors a system can exhibit [52, 31, 39].

Static, Dynamic and Hybrid Analysis

Static analysis is analyzing a piece of software without running it on the system. On the other hand, dynamic analysis is performed by observing and analyzing a program during its execution [22]. Hybrid analysis is a combination of both static and dynamic analysis [20].

An example of static analysis would be to examine the malware source code for indications of malicious behavior. In dynamic analysis, on the other hand, the malware must be executed to determine its impact on the system. For example, the system call log generated during the execution cycle can be reviewed to identify abnormalities. Finally, to improve the malware identification process, hybrid analysis can be applied. An example would be measuring resource usage, combined with a code review of the malicious executable.

Many ransomware types use compression, obfuscation, and encryption strategies to evade detection by static analysis approaches [1]. Thus, this thesis chooses a dynamic anomaly-based ML / DL approach.

2.4 Behavioral Fingerprinting

Gathering device information in order to classify it is known as device fingerprinting. This technique creates a so-called device fingerprint that summarizes the device-specific observable attributes [41].

Device behavioral fingerprinting, focusing on modeling the device's behavior, has been recognized as an emerging and viable method for the identification and the detection of cyberattacks through dynamic analysis [13, 14, 54].

The authors of [54] provide a very detailed overview of the current research conducted in behavioral fingerprinting and its usage in attack detection. The following five behavior sources have been identified to be the most promising in regard to detecting malicious attacks:

- **Resource Usage**: By monitoring different aspects of the device, such as the current memory consumption, the Central Processing Unit (CPU) & disk usage, etc., a behavioral profile of the device can be created. Monitoring resource usage is a prevalent technique for gathering behavioral data. However, it can consume a lot of resources by itself.

- **Hardware events (HPC)**: Another source of behavioral data are Hardware events. These create an accurate representation of the device's low-level behavior. However, these events are CPU specific and can vary depending on the device.

- **Software and Processes**: Behavioral data is also acquired by monitoring each application running on the device in an isolated manner. This data can then be combined to construct a behavioral model. Software signatures, process properties, system calls, and logs are the main features used to model the device's behavior.

- **Network communications**: A common approach that can be applied to almost any device is monitoring the network traffic. Monitoring network packets can yield a broad range of behavioral features that can accurately represent the device's behavior.

- **Device Sensors and Actuators**: Sensors and actuators gather data to model the device's behavior. Some expertise may be required to properly understand and interpret this data, as these sensors and actuators are device-specific.

Besides presenting a thorough overview of device behavioral fingerprinting, the authors of [54] suggest that ML and DL are the most common techniques employed for anomaly detection. Furthermore, they are gaining more traction due to their adaptability and excellent performance when dealing with larger data sets.

2.5 ML & DL

ML & DL techniques for detecting ransomware have garnered a lot of attention due to their ability to identify zero-day threats successfully [27]. This section briefly introduces what ML is and its use in ransomware detection.

2.5.1 What is ML?

There are many different definitions in the literature on what ML is and entails. However, they all share a common idea of training computers to intelligently perform activities by gaining knowledge about their environment through repetition. This acquired knowledge can then be used to produce the desired output from newly evaluated data. ML can be classified into three distinct categories based on data labeling [23]:

1. Supervised ML
2. Unsupervised ML
3. Semi-Supervised ML

Supervised ML

In *supervised learning* a model is trained to produce the desired output from an input. This is accomplished by training a model on sample data with known inputs and defined outputs [23]. Supervised ML thus uses labeled data. Classification and Regression are the two categories of supervised ML. In Classification ML, an algorithm is trained to classify sample data into different output classes correctly. The algorithm makes an educated guess on how the data should be labeled by identifying specific patterns in the data set. An example would be classifying documents into specific categories such as sports, business, or weather. Linear classifiers such as Decision Trees, Support Vector Machines, k-Nearest Neighbor, and Random Forest are typical algorithms used for classification. However, regression algorithms have a different purpose. They explore the relationship between dependent and independent variables in the data. Predicting a future stock value or forecasting the yearly business revenue are some examples where Regression can be applied. Linear Regression and Polynomial Regression represent typical regression algorithms that are widely used [58, 40].

Unsupervised ML

Unlike supervised ML, unsupervised ML algorithms are trained with data only containing known inputs. The training and testing data does not include the outputs [23]. Typical unsupervised ML approaches are Clustering and Dimensionality Reduction.

Clustering, which is applied to larger data sets, divides unlabeled raw data into homogeneous groupings with the same structure. An example would be to group communities on a social media platform into clusters with similar interests (i.e., Photography, Travel, etc.). Typical Clustering algorithms include K-Means Clustering, Hierarchical Clustering algorithms such as Average linkage or Wards linkage, and Probabilistic clustering methods such as the Gaussian Mixture Model. Clustering algorithms are primarily used in pre-processing data tasks. Dimensionality Reduction is a technique for reducing the number of dimensions or features in a sample dataset while keeping some of the properties of the original representation. Typical examples of Dimensionality Reduction algorithms include Singular Value Decomposition, Principal Component Analysis, and Autoencoders using Neural Networks [62, 40].

Semi-Supervised ML

Semi-Supervised ML uses a training data set containing partially labeled data. This approach can be used to increase performance over supervised learning by exploiting the algorithm's access to labeled and unlabeled data [23, 40].

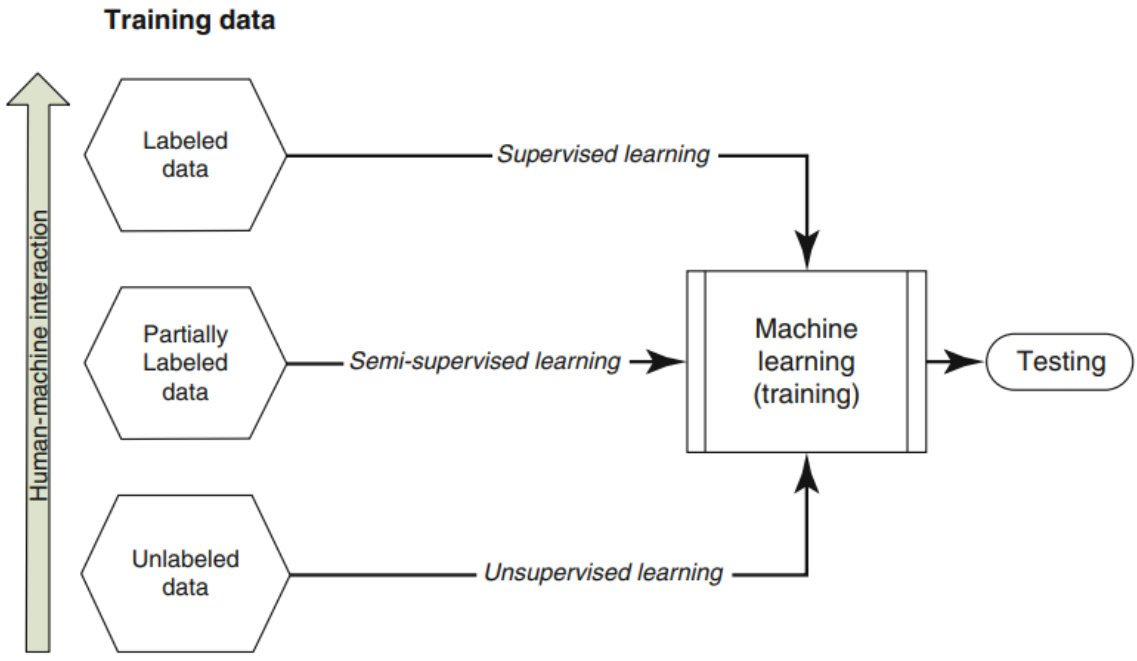

Figure 2.3: ML algorithm categories based on training data [23]

2.5.2 ML and Malware detection

There are various ways in which ML can be leveraged in the context of malware detection. An in-depth review of concrete examples can be found in Chapter 3, which compares different ransomware detection techniques.

Malware can either be detected through static analysis or by executing it on the system and assessing its impact with dynamic analysis. Applying supervised ML in the context of malware detection can be achieved by training/fitting a model with labeled training data (features X with output label Y). For example, the features X might represent specific characteristics of a device's behavior, while the labels Y could indicate if the behavior is malicious or benign. The training goal is to find the optimal parameters for a mapping X to Y that help detect future unseen malware attacks. In unsupervised ML, only the input is provided. The goal is to identify the data's structure or how the output data is generated. For this, algorithms, such as Clustering, can be employed in helping to label new data samples [36].

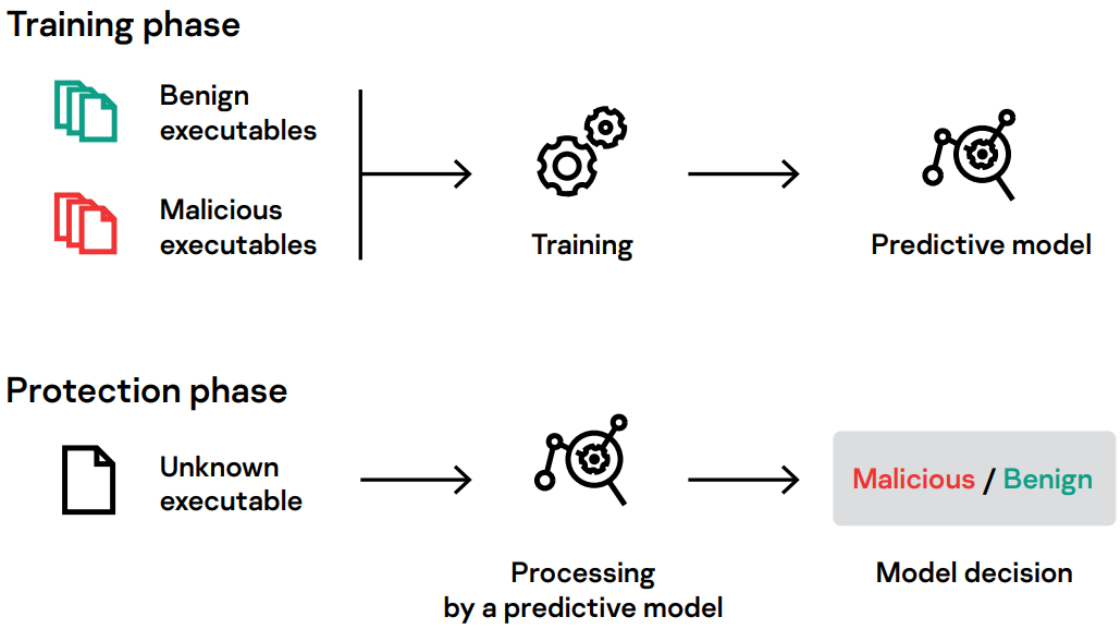

Figure 2.4: Malware detection algorithm life cycle (supervised learning) [36]

2.5.3 ML and DL Algorithms for Anomaly/Novelty Detection

As this thesis focuses on detecting anomalies, specific ML/DL algorithms and techniques are more applicable than others. Novelty detection algorithms can be used to detect anomalies.

In novelty/anomaly detection (unsupervised ML), the goal is to identify if a new data sample is an outlier, referred to as a novelty in this situation. Therefore, to train an anomaly detection algorithm, training data devoid of outliers must be provided [42].

Common ML/DL algorithms that can be utilized in this context include:

- One-Class Support Vector Machine
- Isolation Forest

- Local Outlier Factor
- Autoencoders

One-Class Support Vector Machine (One-Class SVM)

A One-Class SVM is an unsupervised ML algorithm. As the name suggests, it is a Support Vector Machine fitted with training data (not containing anomalies) belonging to only one class. A new data point is then classified as either an outlier or an inlier by the decision boundary. This algorithm works well with higher dimensional data and data sets containing several inlier centers. It is, however, sensitive to outliers, meaning the training set should not include multiple outliers [42, 19]. Figure 2.5 shows the decision boundary and two inlier centers where the regular/normal data is situated. The yellow points represent the new anomalous observations.

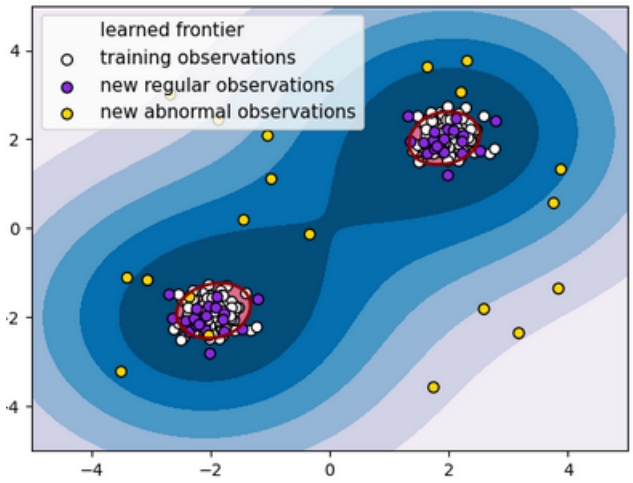

Figure 2.5: One-Class SVM [42]

Isolation Forest (IF)

IF is an unsupervised ML algorithm that tries to isolate outliers from regular data points. As anomalies makeup only a tiny proportion of the entire data set and significantly differ from ”normal” instances, they can easily be isolated. The IF algorithm builds so-called isolation trees, which isolate each data point in the data set separately. Isolation trees are created iteratively by picking a random feature and choosing a random split value (between the min. and max. values of the selected feature). As ”normal” data points are similar (feature similarities), it takes more splits to isolate them from each other. On the other hand, anomalies differ significantly from the rest of the data, requiring fewer splits. The path length from the tree root to the leaf node reflects the anomaly score of a data sample. A shorter path length indicates a higher probability of the sample being an anomaly. Isolation Forest is a suitable algorithm for higher-dimensional data. A training

set with a small number of features may not perform as well, as the number of splits is limited [42, 19].

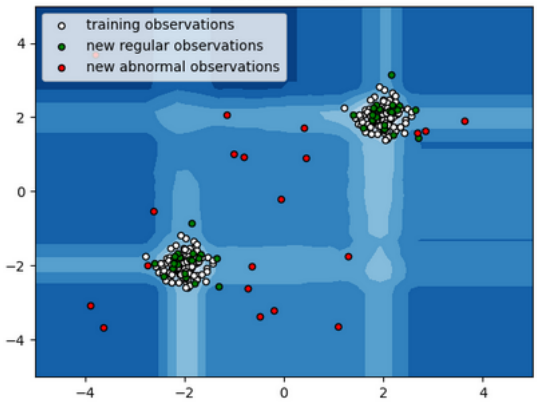

Figure 2.6: IF [42]

Local Outlier Factor (LOF)

LOF is another popular novelty detection method that also falls under unsupervised ML. The algorithm calculates the so-called LOF for each data point in the training set. The LOF is a number that indicates the degree of isolation of a data point related to its surroundings, namely a point's local density compared to its k-neighbors. An anomaly is thus a point whose local density is much lower (high LOF score > 1) than that of its closest neighbors. Compared to distance-based approaches, which are suitable for detecting global outliers, this algorithm can also detect local outliers: (the algorithm also compares density among its local neighbors). In each cluster of data points, different conditions apply to what can be considered an outlier. Furthermore, the algorithm works well with lower and higher-dimensional data and can handle outlier contamination in the training set [15, 19].

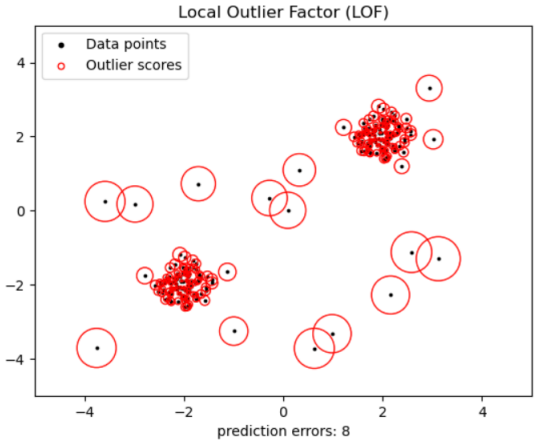

Figure 2.7: LOF [42]

Autoencoders

Autoencoders (unsupervised ML/DL) are a type of artificial neural network trained to output a reconstruction of the input data. The autoencoder achieves this by learning a compressed input representation, thus prioritizing the critical aspects of the training data. Its two main components are the encoder and the decoder [29]. The encoder takes the input data and encodes it into a lower-dimensional representation. The decoder then takes this hidden representation and tries to decode it back into its original input dimensions. The overall goal is to reconstruct as accurately as possible the input data by minimizing the reconstruction errors (difference between output and input). By determining a threshold for this error, anomalies can be detected. The trained autoencoder will struggle to reconstruct anomalous data, resulting in increased reconstruction errors. If this error exceeds the threshold, the data sample will be considered an anomaly [18]. ML algorithms often rely on carefully selected features. On the other hand, Deep Neural Networks yield the best results when given a large number of raw features, thus potentially removing the work-intensive process of feature engineering [19]. Figure 2.8 shows an autoencoder represented as a feed-forward Deep Neural Network. It is configured with 8-4-2-4-8 neurons. The layer in the middle with two neurons (bottleneck) represents the latent/compressed representation of the input data.

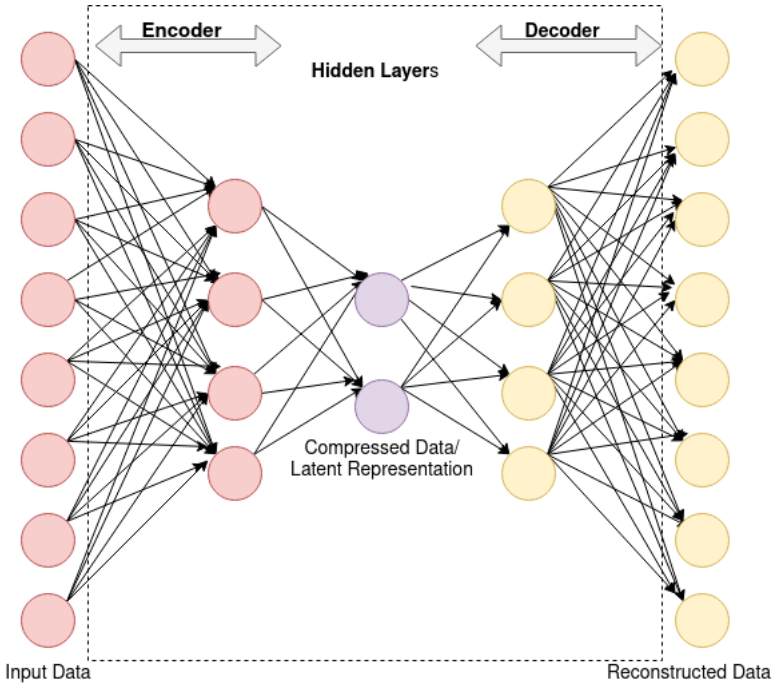

Figure 2.8: Example of an Autoencoder

2.5.4 Feature extraction using NLP

This thesis uses three different behavioral sources to create a behavioral model for the device. ML & DL algorithms require the input data for training and evaluation to be expressed as numbers. However, it is impossible to directly map all of these behavioral

sources to numerical values (i.e., log files). Thus, three different feature extraction methods for textual data are used.

Bag-of-Words (frequency)

A simple natural language processing method is to represent textual data in the so-called bag-of-words representation. This technique converts a textual document or a sentence into a vector of word counts. This feature vector represents how often certain words from a defined vocabulary appear in a sentence/document. The vocabulary includes all possible words appearing in the training sentences/documents. This approach of calculating the frequency of words appearing in textual documents ignores grammar and word order; however, it retains the multiplicity of the words [69].

Example: Assume the two sentences: "This is my thesis." and "It is about detecting ransomware." as training sentences/documents. The vocabulary would then consist of the following words with randomly assigned indexes: {'this': 7, 'is': 2, 'my': 4, 'thesis': 6, 'it': 3, 'about': 0, 'detecting': 1, 'ransomware': 5}. Now assume a third sentence should be expressed as a vector of word counts, using the given vocabulary: "My thesis detects ransomware.". It's feature vector [0 0 0 0 1 1 1 0] represents how many times each word in the vocabulary appears in the sentence:

'about'	'detecting'	'is'	'it'	'my'	'ransomware'	'thesis'	'this'
0	0	0	0	1	1	1	0

Tf-idf

Term frequency-inverse document frequency (Tf-idf) is another method for feature extraction. In this approach, rather than just counting the occurrence of each word, Tf-idf uses a normalized word count, which is based on the following simplified calculation:

tf-idf(w, d) = bow(w, d) * log(N / # documents in which word w appears)

Bow(w,d) is the term frequency, referring to the number of times a word w appears in document d. The fraction inside the logarithm is the inverse document frequency, with N representing the total number of documents. The Tf-idf score will be low for a word appearing in multiple documents and high for a word appearing in hardly any documents [69].

Example: Imagine the word "thesis" appearing in every document of our corpus (i.e., ten documents). The inverse document frequency would then be $log(10/10) = 0$, leaving us with a total Tf-idf score of 0.

Hashing Approach

The hashing approach is another feature extraction method similar to bag-of words. Rather than just storing a dictionary of tokens (vocabulary), this method creates a sparse matrix with the token occurrence counts, utilizing hashing. This hashing trick can be memory efficient when dealing with large data sets, as this extraction method does not require holding a large vocabulary in memory. However, one of the downsides is that the actual token can no longer be retrieved by the column position [30].

Chapter 3

Related Work

3.1 Ransomware detection

This section introduces the reader to 16 recently published scientific papers on ransomware detection. A summary of each, including their most important findings, is included.

Table 3.1 provides an overview of the operating system, the type of analysis, the approach, the domain, the algorithms used, and the ransomware detection accuracy achieved in each work.

Overview of Ransomware Detection Techniques

Ahmed et al. propose in [1] an Application Programming Interface (API) based ML industrial IoT framework for detecting ransomware in the early stages of infection. System-specific API calls were collected by running benign and ransomware samples in a virtual sandbox environment. For further data refinement WEmRmR, an enhanced version of the Maximum Relevance — Minimum Redundancy (mRmR) algorithm, was utilized together with Tf-idf to distinguish the most relevant features and rank them according to their importance. Six different ML classifiers were employed, achieving a maximum accuracy of 98.64% with a low false positive rate of 1.7%.

Another API and ML-based ransomware detection framework is proposed by Almousa et al. [5]. Two hundred forty-nine different ransomware-specific system API calls and 229 benign API calls were extracted by running 58 ransomware and 66 benign samples in a sandboxed environment for 10 minutes each. Before training the ML algorithms, standardization and dimensionality reduction were performed on the extracted data to reduce noise and improve the overall detection performance. As a result, the highest ransomware detection accuracy of 99.18% was achieved using the k-nearest neighbors (kNN) algorithm.

By extracting Windows API invocation sequences, the authors of [8] created a framework that could distinguish between ransomware, malware, and benign files. This feat was

achieved using six different ML algorithms for multi-class classification and yielded an accuracy of 98.65%.

Poudyal and Dasgupta [44] used a hybrid reverse engineering technique to extract ransomware features from three different levels. These features include a list of the Dynamic Link Libraries (DLLs) called by the ransomware code, function calls, and the assembly instructions utilized by the malicious software. The ransomware detector combines a data mining approach with Natural Language Processing (NLP) to create a feature database and uses ML for classification. The highest ransomware detection accuracy of 99.72% with a 0.3% false positive rate was achieved with supervised learning using Support Vector Machine (SVM).

Almomani et al. [4] extracted API and permission features from decompiled Android package (APK) files containing ransomware. For classification, the SVM algorithm was deployed alongside an oversampling technique. The optimization approach known as particle swarm optimization (PSO) was then utilized to enhance the feature selection, improving the overall identification process. As a result, an accuracy of 97.5% was reported.

In [32] Imtiaz et al. compared the effectiveness of detecting malware, including ransomware, on smartphones, using DL and ML malware detection approaches. The model data set included dynamic features such as API calls and power usage and static features such as intents and permissions. The results showed that DeepAMD outperformed other ML techniques in detecting and identifying malware on a static and dynamic layer.

Another technique to detect ransomware by monitoring the energy consumption of smartphones was proposed by Azmoodeh et al. [7]. By subsampling the collected power consumption data and applying the kNN algorithm with dynamic time warping (DTW), a detection rate of 94.27% was achieved.

Rhode et al. [51] tried to identify 3000 ransomware samples by employing a behavior-based model analyzing a snapshot of performance counter benchmarks such as memory and CPU usage in the early stages of infection. Based on recurrent neural networks, this approach correctly identified ransomware with a 94% detection rate in the first 10 seconds of file execution.

A dynamic analysis detection approach was proposed by [47] monitoring the changes regarding system resources, the retention state of applications, and file operations/movement. Finite State Machine (FSM), a model based on state changes and transitions, was created and utilized for detecting ransomware. A state change is triggered whenever an anomaly in the monitored features is identified. Reaching one of three final states indicates a ransomware infection on the system. The paper achieved a breathtaking 99.5% accuracy and 0% false positive rate.

RAPPER is a two-step ransomware detection approach introduced by Alam et al. [2]. First, a Deep Neural Network (DNN) was trained to detect anomalies with time-series data of monitored HPC events representing normal device behavior. Fast Fourier Transformation (FFT) was applied for transforming the time domain values into frequency domain values. This was done to help identify repetitive patterns triggered by a ransomware infection, such as opening, encrypting, and closing a file. In the last step, the

transformed data was fed into a second DNN. The proposed framework was able to detect the WannaCry ransomware in 5.313 seconds.

Berrueta et al. [12] compared the ransomware detection effectiveness of three ML algorithms by searching for specific read and write patterns in the network communication (encrypted and clear-text file sharing) between clients and file servers. The data collected to train the models include both traffic generated by benign programs and ransomware while accessing and operating on shared network file servers. An astonishing detection accuracy of 99.8% and a low false-positive rate of 0.004% was reported using neural networks.

Another network-based approach was proposed by [3], where packet and flow-based network traffic was monitored to detect ransomware. For this, a multi-classifier detection framework based on ML was created, consisting of two parallel operating classifiers. With an accuracy of 97.92% percent, the Random Tree algorithm was found to be the most accurate in detecting packet-based ransomware network activity. At the same time, the Bayes Network model provided the highest scores in flow-based ransomware network detection with 97.08%.

Faghihi & Zulkernine [26] proposed RansomCare, a hybrid analysis system detecting novel crypto-ransomware on smartphones and recovering lost data from an attack. The proposed framework identifies known ransomware types in a signature-based static analysis component by their hashes (SHA256). At the same time, zero-day crypto-ransomware strains were detected by looking for anomalies in file modification and deletion I/O events. This was accomplished by calculating the entropy of files that were modified or deleted and comparing it to a predefined threshold. As encrypted files are usually unstructured and contain a large amount of random data, they will also have a high entropy score, indicating a ransomware infection. An accuracy of 99.24% was observed with a false positive rate of 0.49%.

Tang et al. [59] introduced RansomSpector, a policy-based dynamic crypto-ransomware detection approach that uses virtual machine introspection (VMI) to monitor file I/O and network activities. RansomSpector is unique compared to other detection techniques because it is hidden in the hypervisor layer, thus making it undetectable for ransomware. With a zero false-positive rate, the paper claims to have successfully identified 771 crypto ransomware strains from a dataset containing 2,117 malware samples.

Using reverse engineering, Sharma et al. [56] extracted different ransomware features. These included permissions, images, intents, etc., from a total of 2076 ransomware and 2000 benign APKs. Using Gaussian Mixture Model (GMM), an unsupervised clustering-based ML technique as a classifier, Ransomdroid achieved a detection accuracy of 98.08%.

Huertas et al. [17] analyzed ransomware affecting resource-constraint devices used as spectrum sensors in the crowdsensing platform Electrosense. Ransomware was identified by finding anomalies in Memory usage, CPU usage, I/O activities, HPC, and kernel events of the device. The administrator's policies would then classify the device's behavior as malicious or normal. Anomalies generated by two different ransomware samples were correctly identified with an 89.80% true positive rate.

Table 3.1: Overview of Ransomware detection techniques

Paper	OS	Analysis	Approach	ML/DL Algorithms	Domain	Accuracy
[1] (2022)	Windows	dynamic	ML	LR, DT, kNN, SVM, AD, RF, mRmR	Software & Processes	98.64%
[5] (2021)	Windows	dynamic	ML	RF, SVM, kNN	Software & Processes	99.18%
[8] (2020)	Windows	dynamic	ML	RF, LR, NB, SGD, KNN, SVM	Software & Processes	98.65%
[44] (2021)	Windows	hybrid	ML, DL	LR, AD, SVM, NN, RF, NLP	Other	99.72%
[4] (2021)	Android	static	ML	SVM	Other	97.50%
[32] (2021)	Android	hybrid	DL	DNN	Ressource Usage, Other	
[7] (2018)	Android	dynamic	ML, DL	kNN, SVM, NN, RF	Resource Usage	94.27%
[51](2018)	Windows	dynamic	DL	RNN	Resource Usage	94.00%
[47](2020)	Windows	dynamic	anomaly-based		Resource Usage	99.50%
[2] (2020)	Linux	dynamic	DL	DNN (LTSM)	HPC	
[12] (2022)	Windows	dynamic	ML,DL	DT, TEs, NN	Network	99.80%
[3] (2019)	Windows	dynamic	ML	RT,BN	Network	97.92 97.08%
[26] (2021)	Android	hybrid	data-centric		Software & Processes, Other	99.24%
[59] (2020)	Windows	dynamic	policy-based		Network, Software & Processes	
[56] (2021)	Android	static	ML	GMM	Other	98.08%
[17] (2022)	Linux	dynamic	policy-based		HPC, Resource Usage	89.80%

Summary of Related Work

The following points represent the findings from the literature:

- Most of the research in regards to ransomware detection has been conducted on platforms running Windows.
- There seems to be very little research into detecting ransomware affecting resource constraint devices. The only one mentioning resource constrain devices is [17].

- Dynamic and hybrid analysis appear to be the most popular approaches in the reviewed literature.
- ML and DL approaches were mostly used for ransomware classification/detection.
- Different behavioral data domains have been used for detecting ransomware. Resource Usage and Software & Processes are the most popular. However, there aren't many papers comparing their efficiency in regard to detection and classification.
- None of the papers compare a policy-based approach with an ML & DL approach
- Most ransomware detection frameworks seem to be a proof of concept and do not cover aspects such as data protection, resource usage, and distribution.

Chapter 4

Scenario

This chapter will provide the reader with an overview of ElectroSense platform and the hardware/software configuration used as the basis for this thesis. In addition, a high-level summary of the different ransomware samples and their functionalities is provided.

4.1 ElectroSense

Measuring electrosmog, boosting home WiFi connectivity, or finding and locating concealed signals are some use cases where spectrum resource monitoring is crucial. Due to the growth in wirelessly connected devices, spectrum usage has increased considerably over time. As a result, an effective, secure, and dependent method of monitoring spectrum data is needed. ElectroSense, a crowdsourced network, provides a credible and cost-effective solution to this problem [46].

ElectroSense is an open initiative and crowd-sourcing project that employs affordable radio sensors to gather spectrum data for further analysis. Becoming part of this network requires the purchase of a single-board computer such as a Raspberry Pi with a stable internet connection linked via a dongle (Radio Frontend) to an antenna. The data collected by the different sensors can then be accessed either through an API or by visiting the member area on their website [24]. As the project is open source, the software used to run the sensors is available on Github [25].

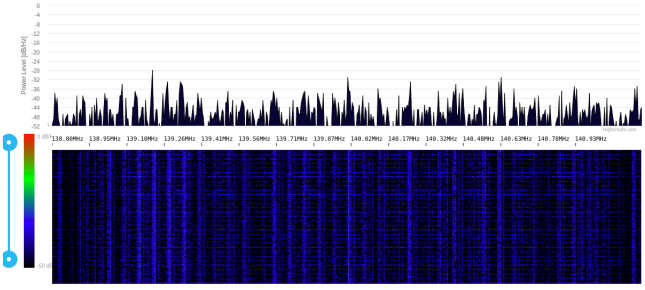

Figure 4.1: Spectrum Decoder for FM Radio on ElectroSense platform

4.1.1 Thesis Setup

For this thesis, the resource-constrained spectrum sensor is a Raspberry Pi 3 Model B with 1 GB of RAM and a Quad-Core 1.2GHz Broadcom BCM2837 64bit CPU. The ElectroSense Software, which is based on Raspbian, is utilized as an operating system. The Raspberry Pi is connected via a Radio Frontend (RTL-SDR Silver v3) to an antenna, creating a suitable sensor for the ElectroSense platform. To guarantee a continuous internet connection, the device must be connected via Ethernet cable to a Broadband internet provider (such as UPC Cablecom).

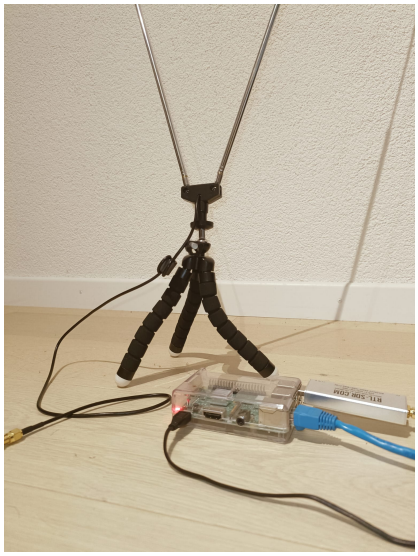

Figure 4.2: Sensor for thesis

4.2 Ransomware

This section provides a brief overview of the three ransomware samples used for this thesis and outlines their functionality.

4.2.1 Ransomware-PoC

Ransomware-PoC is a proof of concept open-source Python ransomware payload that can be downloaded from the following GitHub repository [48]. This software allows a user to encrypt or decrypt files by providing a starting directory. The ransomware scans the 'start' directory and its sub-directories for files with the proper extensions (list of file extensions in the source code). It then encrypts an AES (256-bit) key with an RSA public key, which is used for file encryption. As this ransomware allows the user to decrypt their files after encryption, the private RSA server key is also hardcoded into the payload. The source code was modified slightly to display the starting and ending timestamp of the encryption phase.

```
# Recursively go through folders and encrypt/decrypt files
print("Starting encryption/decryption...{}".format(round(time.time())))
for currentDir in startdirs:
    for file in discover.discoverFiles(currentDir):
        if encrypt and not file.endswith(extension):
            print("Encrypting file {}".format(file))
            modify.modify_file_inplace(file, crypt.encrypt)
            os.rename(file, file + extension)
            print("File changed from " + file + " to " + file + extension)
        if decrypt and file.endswith(extension):
            modify.modify_file_inplace(file, crypt.encrypt)
            file_original = os.path.splitext(file)[0]
            os.rename(file, file_original)
            print("File changed from " + file + " to " + file_original)
print("Finished encryption/decryption...{}".format(round(time.time())))
```

Listing 4.1: Main encryption/decryption function of Ransomware-PoC

4.2.2 DarkRadiation

DarkRadiation is a type of ransomware that attacks Linux-based systems. This sophisticated ransomware is implemented entirely in bash, using Telegram, a messaging application, as its command-and-control server. DarkRadiation employs an SSH worm that downloads the ransomware payload after establishing a successful connection with the victim's device. The ransomware communicates with the attackers using the Telegram API and encrypts files using the OpenSSL AES algorithm (256-bit key length). As a first step, the ransomware checks to determine if it has root privileges and then downloads all the needed dependencies (curl & OpenSSL). Next, changes in user activity (new logins, log-outs) are transmitted to the C2 server [68].

Three different file encryption functions are present in the DarkRadiation ransomware. Encrypt_grep_files() is the first function to be run. As its name implies, the grep command searches the file system for files with the extensions (.txt,.py, and .sh) and encrypts them. After that, it deletes the original file and uploads the encrypted version to the original location.

Both encrypt_home() and encrypt_db() operate similarly. The /home directory or database-specific files are the targets of these functions. During each phase, the malware updates the attackers on its progress. Timestamps have been added to the different encryption methods to see when each function started and ended.

Figure 4.3: Telegram DarkRadiation Bot

4.2.3 RAASNet

As ransomware attacks have become more lucrative and prevalent in recent years, a new type of service, 'Ransomware-as-a-Service (RaaS), has appeared on the darknet. This service is offered as a franchise model and is marketed toward attackers with little or no previous programming experience. This new way of selling a tool to commit a crime enables ordinary (non-technical) individuals to participate in the ransomware economy [38].

An open-source Ransomware-as-a-Service written in Python called RAASNet shows how simple it is to create and deploy ransomware and can be downloaded from GitHub [45]. This cross-platform tool offers an intuitive graphical user interface, allowing any user to develop customized ransomware. In addition, it includes a built-in command and control server for receiving private encryption keys.

This tool offers different ransomware payload customization options. These include defining:

- The targeted directories and sub-directories
- The ransom message with the payment instructions
- The file encryption method and file extensions for encryption

- The IP address of the command and control server

Furthermore, it offers the option to compile the payload and decryption program as an executable. This creates a ransomware payload that can be deployed to targeted machines. A decryption key is generated and forwarded to the attacker during machine infection.

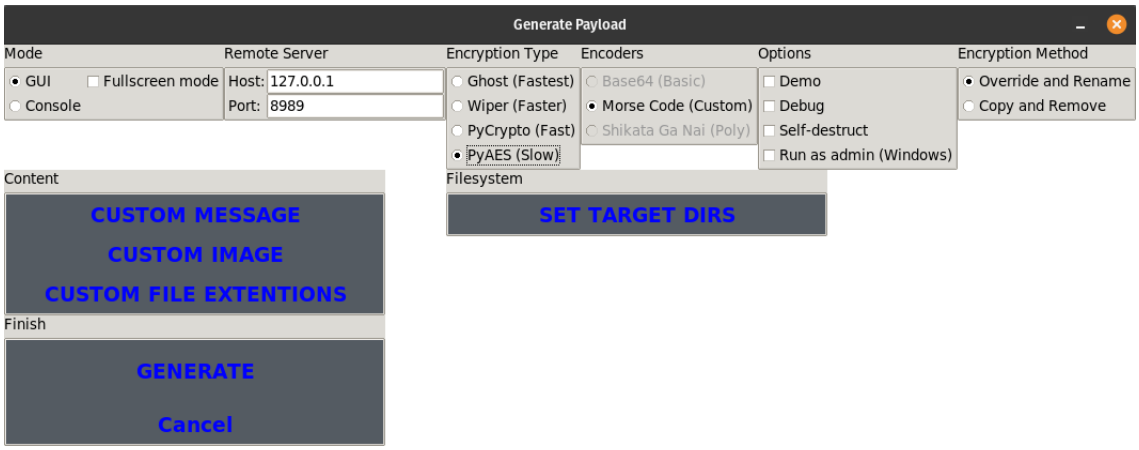

Figure 4.4: RAASNet graphical user interface for customizing Ransomware payload

Thesis specific RAASNet payload

As Ransomware-PoC already uses the pycrypto Python package for encryption, pyaes was selected for this specific ransomware variant. To encrypt files, the ransomware firstly creates a 32-byte key (256 bit) and sends it to the attacker's command and control server. It then scan the defined directories and subdirectories(/home, /mnt) for files based on a list of file extensions. The following Linux-specific file extensions have been selected:

```
['.txt', '.py', '.html','.sh','.asc','.awk',
'.bak','.bz2','.c', '.C','.cc', '.dat','.doc',
'.dvi','.el','.elc','.f','.f77','.log','.info',
'.jpeg','.tar','.tar.gz','.tgz','.zip']
```

After the scanning phase, the ransomware encrypts the files with AES's encryption algorithm. The GUI option was selected for compiling the payload as there was an issue generating ransomware for the Command-Line Console. Deployment via the command line is possible by manually removing all GUI-specific elements.

```
1 def encrypt_file(file_name, key):
2     aes = pyaes.AESModeOfOperationCTR(key)
3
4     with open(file_name, 'rb') as fo:
5         plaintext = fo.read()
6     enc = aes.encrypt(plaintext)
7
8
```

```
9     with open(file_name, 'wb') as fo:
10        fo.write(enc)
11    os.rename(file_name, file_name + '.DEMON')
```

Listing 4.2: Main encryption function of RAASNet

Chapter 5

Framework Design

This chapter introduces the reader to the design of the framework used for ransomware detection on resource-constrained devices. It includes an overview of the components that make up the framework and key design decisions adopted in its development. The reader may refer to Chapter 6 for a more detailed presentation of the framework's implementation.

5.1 Purpose and Intended Audience

Incidences of ransomware attacks have become ever more frequent and lucrative. With increased connectivity among devices and the internet as a means of distribution, cybercriminals are now finding it easier to infect machines and their networks. Growing concerns about ransomware targeting IoT devices have emerged due to their rapid adoption, high inter-connectivity, and poor maintenance. In addition, customized ransomware such as DarkRadiation has started to appear, explicitly targeting Linux-based systems. The ElectroSense platform, which uses resource constraint sensors that run a distribution of Linux, could fall victim to ransomware attacks. Sensors infected with ransomware will have their data encrypted and thus not operate correctly. This would, in turn, significantly impact a platform's overall operation. The framework developed in this thesis can detect ransomware infections on resource constraint devices.

5.2 Requirements

The ElectroSense Pi (sensor) collects spectrum data and sends this information to the ElectroSense 'data collection' backend for analysis and processing.

The ransomware detection framework has been designed to integrate into the ElectroSense system architecture. While gathering and understanding all user requirements is impossible, the framework design has tried to anticipate stakeholder needs. These were reviewed and confirmed with the university project sponsors.

For this thesis, a distributed system architecture was adopted with one component running on a Raspberry Pi and another on a data collection/processing server. As the Raspberry Pi sensor is a resource-constrained device running Linux, processing limitations on this platform exist.

The following functional requirements were defined for the framework.

5.2.1 Data Collection/Monitoring Requirements R1 (On the Raspberry Pi)

R1.1: The data collection process should not interfere with the regular operation of the sensor.

R1.2: The data collected should be safely sent to the server and only briefly stored on the sensor because of its limited storage capacity.

R1.3: Specific data protection mechanisms must be in place to securely collect and transfer the data.

R1.4: The framework must be able to monitor multiple behavioral dimensions on the ElectroSense Pi (sensor) and identify the best algorithms and features for anomaly detection and classification.

R1.5: The monitoring service should be easy to deploy and use on the sensor.

R1.6: The service should be standardized for each behavioral source and expandable to manage new behavioral dimensions.

R1.7: The user should be able to customize how the device is monitored.

R1.8: The framework architecture needs to support both online and offline data capture and analysis.

5.2.2 Data Pre-processing, Training and Evaluation Requirements R2 (On the Server or Raspberry Pi)

R2.1: The application running on the server must be able to clean the data automatically without user intervention.

R2.2: Training/Evaluating ML and DL algorithms should be applied without user interaction.

R2.3: The framework should be able to detect anomalies and classify ransomware correctly.

R2.4: Evaluation metrics must be stored in a database.

R2.5: The application must be able to handle requests from multiple sensors.

R2.6: The framework should be easily extendable with other ML/DL algorithms.

R2.7: The application should be user-friendly and easy to deploy.

R2.8: The application should be able to detect other malware types.

5.2.3 Graphical User Interface Requirements R3 (On the Server or Raspberry Pi)

R3.1: Access to the Graphical User Interface is protected and only accessible to authorized users.

R3.2: The user can access different evaluation metrics by selecting: a sensor device, a monitoring script, and the required ML/DL models.

R3.3: Live monitoring evaluation of classification and anomaly detection must be graphically displayed to the user.

5.3 Design Overview

The proposed ransomware detection framework utilizes a distributed architecture. The design decision ensures the workload on the Electrosense sensor remains as low as possible, given the resource-constrained nature of the device itself. As a result, only a small subset of the framework's features is deployed on the Electrosense sensor. The following diagram 5.1 presents an overview of the system design chosen for the thesis. It consists of two components:

1. A middleware Monitor Controller running on the Raspberry Pi.
2. A Data Analysis Application running on the server, consisting of a pre-processing layer, detection layer, and visualization layer.

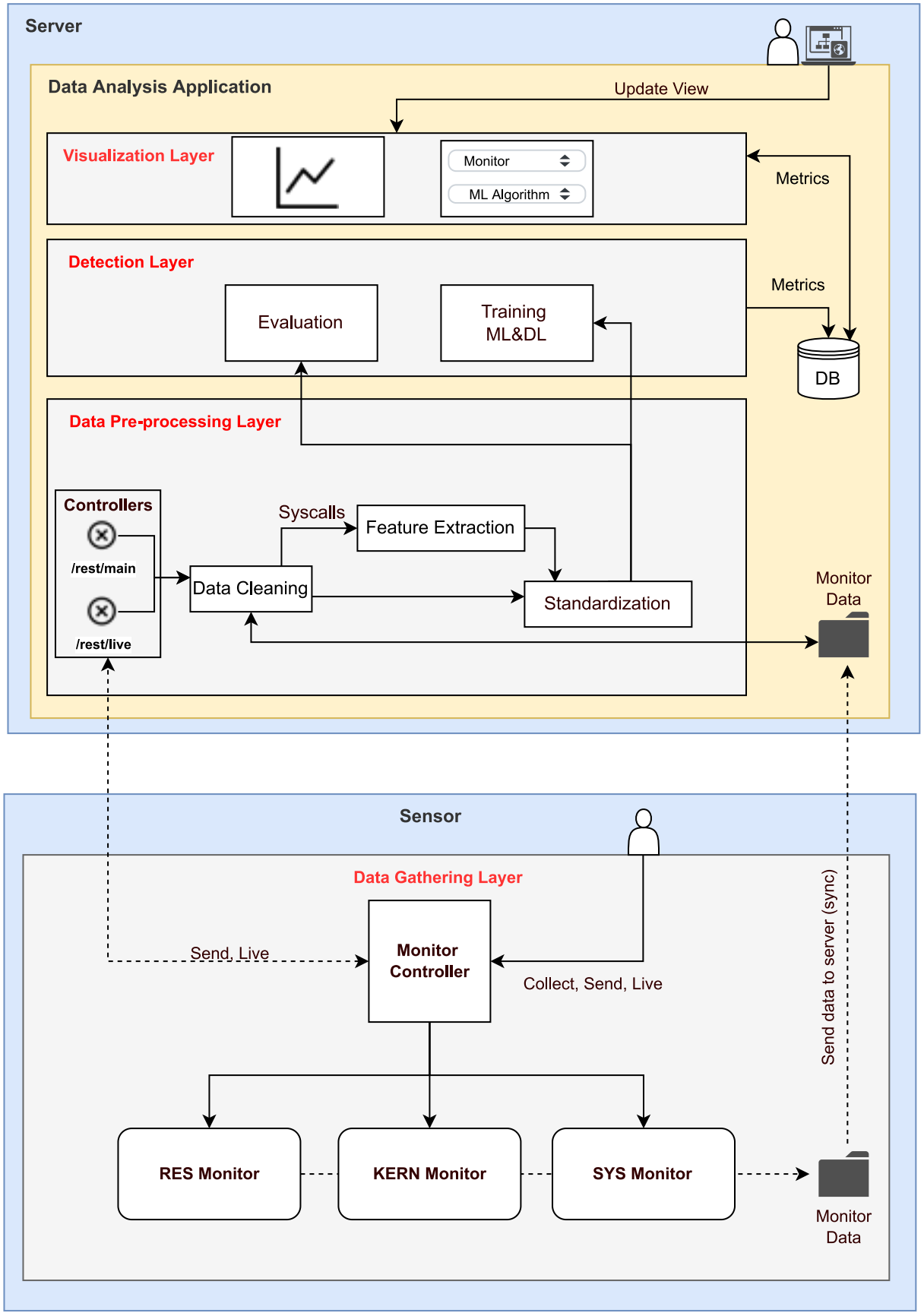

Figure 5.1: Design Overview of the Framework

5.3.1 Data Gathering Layer

The data gathering layer resides on the Raspberry Pi sensor. At its heart are three monitoring scripts (Resource Consumption Monitor, Kernel Monitor, and System Call Monitor) that capture specific behavioral data. A separate tool, the Monitor Controller, controls the three monitoring scripts. This program is an intermediary between the Raspberry Pi sensor and the server. Its primary function is to initiate, halt, and control the various monitoring scripts while they are running. With this application, the user can customize different monitoring elements, including the time to monitor, which scripts to run in parallel, the server address to send the data, and the type of monitoring (i.e., capturing training or evaluation data). A secure data exchange connection is established between the two machines using public key authentication. Data collected from the three monitoring scripts are sent every ten seconds to the server to avoid any data loss during ransomware execution. The Monitor Controller provides three main functions:

1. **Command Collect**: This function allows the user to collect training and evaluation data for anomaly detection or classification. In addition, the user can define how long the chosen monitoring scripts will run. Collected monitoring data is sent to the server every ten seconds.

2. **Command Send**: Sends information (metadata) about the gathered data from the Command Collect process to the server to initiate training or testing. This information includes the path where the data is stored, the monitors used for the monitoring session, etc.

3. **Command Live**: Allows the user to start a live (online) monitoring session on the sensor. This process will collect evaluation data for a defined time frame and forward it to the server for evaluation on a per sample basis.

The text below provides more detailed descriptions of the three monitoring scripts managed by the Monitor Controller on the Raspberry Pi sensor.

Resource Consumption Monitor (RES Monitor)

This script was developed by Huertas et al. [17] as part of a bachelor thesis. It offers specific customization options to the administrator. In addition, it tracks specific behavioral metrics every 5 seconds linked to the device's hardware. These include information on CPU & memory usage, disk utilization, kernel tracepoint events, and HPC. The table below highlights the various metrics that the RES Monitor tracks.

Table 5.1: Features Monitored RES Monitor

time			net:netif_rx
ioread			timer:tick_stop
iowrite	seconds	filemap:mm_filemap_delete_from_page_cache	sched:sched_process_exec
ioreadbytes	block:block_bio_frontmerge	gpio:gpio_value	sched:sched_waking
iowritebytes	block:block_dirty_buffer	irq:softirq_exit	task:task_newtask
ioreadtime	block:block_split	pagemap:mm_lru_activate	sched:sched_stat_runtime
iowritetime	block:block_touch_buffer	rpm:rpm_return_int	timer:timer_cancel
iobusytime	ext4:ext4_es_lookup_extent_enter	fib:fib_table_lookup	timer:timer_init
read_merge	ext4:ext4_ext_load_extent	raw_syscalls:sys_enter	timer:timer_start
write_merge	ext4:ext4_writepages_result	random:credit_entropy_bits	workqueue:workqueue_execute_start
memory	ext4:ext4_journal_start	kmem:kfree	branch-instructions
net_in	filemap:mm_filemap_add_to_page_cache	kmem:kmem_cache_alloc	branch-misses
net_out	jbd2:jbd2_handle_stats	kmem:mm_page_alloc_zone_locked	bus-cycles
pkt_in	ext4:ext4_da_update_reserve_space	kmem:mm_page_free	cache-misses
pkt_out	ext4:ext4_sync_file_enter	mmc:mmc_request_done	cache-references
err_in	jbd2:jbd2_checkpoint_stats	writeback:global_dirty_state	cpu-cycles
err_out	ext4:ext4_free_inode	writeback:sb_clear_inode_writeback	instructions
drop_in	ext4:ext4_evict_inode	writeback:wait_on_page_writeback	context-switches
drop_out	ext4:ext4_releasepage	napi:napi_poll	armv7_cortex_a7/l1d_cache_wb/
cpu	ext4:ext4_unlink_enter	tcp:tcp_probearmv7_cortex_a7/br_immed_retired/	armv7_cortex_a7/l1d_tlb_refill/
cpu-migrations	block:block_bio_remap	armv7_cortex_a7/br_mis_pred/	armv7_cortex_a7/l1i_cache/
minor-faults	LLC-store-misses	armv7_cortex_a7/br_pred/	armv7_cortex_a7/l1i_cache_refill/
page-faults	LLC-stores	armv7_cortex_a7/bus_cycles/	armv7_cortex_a7/l1i_tlb_refill/
L1-dcache-load-misses	branch-load-misses	armv7_cortex_a7/cpu_cycles/	armv7_cortex_a7/l2d_cache/
L1-dcache-loads	branch-loads	armv7_cortex_a7/exc_return/	armv7_cortex_a7/l2d_cache_wb/
L1-dcache-store-misses	dTLB-load-misses	armv7_cortex_a7/exc_taken/	armv7_cortex_a7/ld_retired/
L1-dcache-stores	dTLB-store-misses	armv7_cortex_a7/inst_retired/	armv7_cortex_a7/mem_access/
L1-icache-load-misses	iTLB-load-misses	armv7_cortex_a7/l1d_cache/	armv7_cortex_a7/pc_write_retired/
L1-icache-loads		armv7_cortex_a7/l1d_cache_refill/	armv7_cortex_a7/st_retired/
LLC-load-misses			armv7_cortex_a7/unaligned_ldst_retired/
LLC-loads			armv7_cortex_a7/cid_write_retired/

Kernel Monitor (KERN Monitor)

The KERN Monitor is another monitoring script developed by Dr. Alberto Huertas. It monitors specific aspects of the device every 5 seconds and tracks events related to disk I/O, CPU, kernel memory, and system calls. Features tracked by the KERN Monitor are shown below:

Table 5.2: Features Monitored KERN Monitor

time		random:get_random_bytes	
timestamp		random:mix_pool_bytes_nolock	
seconds	irq:irq_handler_entry	random:urandom_read	
connectivity	irq:softirq_entry	raw_syscalls:sys_enter	
alarmtimer:alarmtimer_fired	jbd2:jbd2_handle_start	raw_syscalls:sys_exit	
alarmtimer:alarmtimer_start	jbd2:jbd2_start_commit	rpm:rpm_resume	udp:udp_fail_queue_rcv_skb
block:block_bio_backmerge	kmem:kfree	rpm:rpm_suspend	workqueue:workqueue_activate_work
block:block_bio_remap	kmem:kmalloc	sched:sched_process_exec	writeback:global_dirty_state
block:block_dirty_buffer	kmem:kmem_cache_alloc	sched:sched_process_free	writeback:sb_clear_inode_writeback
block:block_getrq	kmem:kmem_cache_free	sched:sched_process_wait	writeback:wbc_writepage
block:block_touch_buffer	kmem:mm_page_alloc	sched:sched_switch	writeback:writeback_dirty_inode
block:block_unplug	kmem:mm_page_alloc_zone_locked	sched:sched_wakeup	writeback:writeback_dirty_inode_enqueue
cachefiles:cachefiles_create	kmem:mm_page_free	signal:signal_deliver	writeback:writeback_dirty_page
cachefiles:cachefiles_lookup	kmem:mm_page_pcpu_drain	signal:signal_generate	writeback:writeback_mark_inode_dirty
cachefiles:cachefiles_mark_active	mmc:mmc_request_start	skb:consume_skb	writeback:writeback_pages_written
clk:clk_set_rate	net:net_dev_queue	skb:kfree_skb	writeback:writeback_single_inode
cpu-migrations	net:net_dev_xmit	skb:skb_copy_datagram_iovec	writeback:writeback_write_inode
cs	net:netif_rx	sock:inet_sock_set_state	writeback:writeback_written
dma_fence:dma_fence_init	page-faults	task:task_newtask	
fib:fib_table_lookup	pagemap:mm_lru_insertion	tcp:tcp_destroy_sock	
filemap:mm_filemap_add_to_page_cache	preemptirq:irq_enable	tcp:tcp_probe	
gpio:gpio_value	qdisc:qdisc_dequeue	timer:hrtimer_start	
ipi:ipi_raise		timer:timer_start	

System Call Monitor (SYS Monitor)

The third monitor collects system call data for the entire device to properly monitor the requests made from the device to the OS kernel. These system calls are collected every 10 seconds. As the data collected is relatively large, the system call files are only stored temporarily on the Raspberry Pi and then sent to the server.

Usability of the Monitoring Service

The following two pictures 5.2 and 5.3 show the middleware Monitor Controller deployed on a device and activated by a user. The monitoring table allows tracking of previous monitoring events and their current status. The second image depicts an active monitoring session demonstrating the many customization possibilities.

```
root@sensor(rw):~/BA_Thesis_PI/middleware# python3 cli.py show
Table for the device with the following cpu-id: 000000005426d851
```

#	Description	Task	Category	Type	Monitors	Seconds	Done	Sent	Added	Completed

Figure 5.2: Table with all tasks monitored

```
root@sensor(rw):~/BA_Thesis_PI/middleware# python3 cli.py collect
Please add a short description for this task: Collecting training data
normal, poc, dark or raas (normal, poc, dark, raas): normal
Which category testing or training (training, testing): training
time in seconds [3600]: 300
Which monitors (i.e m1,m2,m3): m1,m2,m3
Server path (i.e root@194.233.160.46:/root/data): dennis@192.168.0.253:/home/dennis/Documents/new
Type anomaly or classification (anomaly, classification): anomaly
Stopping Services if they are still running...
Starting Monitor Services and running it for 300 seconds
Please wait 300 seconds to start a new Monitoring Todo
  [#####------------------------------]   14%  00:04:16
```

Figure 5.3: Monitoring session

5.3.2 Data Pre-Processing Layer

After transmitting the monitored data, the server application will start pre-processing data tasks. For the RES Monitor and KERN Monitor scripts, raw data is stored as a simple file containing only numerical values.

On the other hand, the system call data (from the SYS Monitor) consists of multiple files (one for every 10 seconds of monitoring) and contains both strings and numbers. Pre-processing is necessary, as ML and DL algorithms rely on numerically expressed data. Additionally, processing the data before the training and evaluating phase can dramatically increase the overall performance and accuracy of the model. Although there are variations in the input data, a general pre-processing procedure can be defined as follows:

1. Data Cleaning
2. Feature Extraction / Selection
3. Standardization

Data Cleaning

Data cleaning aims to remove unwanted data and transform it into a form usable for ML/DL algorithms. As mentioned before, the steps vary depending on the monitored behavioral sources. The following procedure is applicable for the RES and KERN monitoring scripts:

- Rows containing duplicate or empty values are removed.
- Unwanted features and temporal features such as timestamps are dropped.

Preparing the system call data is a more work-intensive process. Files containing system call information can be several megabytes in size and have thousands of rows. As a first step, all system call names are extracted from each of the files. These are then stored as a string (separated by spaces). The final step is to create an array of the individual strings extracted in the previous step. This array represents the corpus (body) of all system call entries. For example, if we had two files, each containing five system call entries, our corpus would consist of an array of two strings, with each string having five values.

Feature Extraction & Feature Selection

It is not necessary to extract features from the RES and KERN monitoring scripts, as the data is already expressed as numeric values. Feature selection, however, becomes necessary as this thesis used the unmodified versions of the monitoring scripts and ran them in parallel (RES, KERN, and SYS). As these can influence each other during the monitoring process, certain features exhibit substantial deviations when establishing a baseline. In the graph below, the green region measures the device's normal behavior, whereas the blue reflects subsequent monitored behavior. As can be seen, there are significant differences between the two data samples. These should theoretically be similar.

Plot 5.4 shows the values of a feature over time.

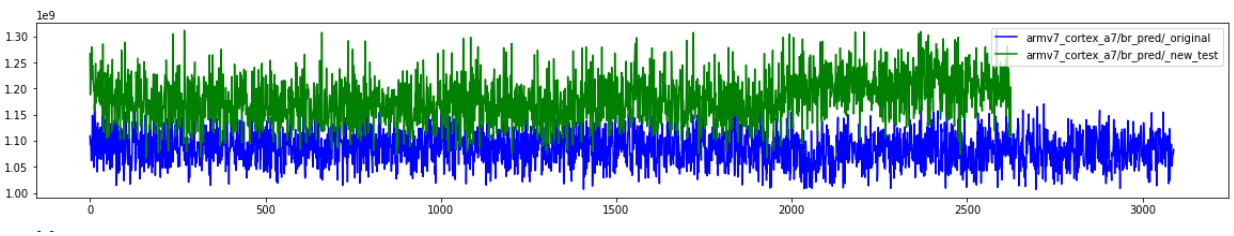

Figure 5.4: Highly fluctuating feature

In a separate processing step, all features having high fluctuations between measurements were removed. Not removing them would lead to high false positive rates when evaluating "normal" data.

Since the system call data is still in textual form, feature extraction is necessary. Extraction techniques such as Bag-of-words, Tf-idf, and hashing are used to construct unigram feature vectors. These vectors are then utilized to train or evaluate ML/DL algorithms.

Standardization

Standardization is a standard data pre-processing step used in data science. The method involves rescaling the data to have a standard deviation of 1 and a mean of 0, thus preventing issues later during ML training due to features having a diverse range of values [19].

This process of normalizing/standardizing the data for ML training involves three steps. As a first step, the cleaned data is split into two parts. For this thesis, 90% of the data is used for ML training, while the remaining 10% is utilized for validation.

The standardization scaler is then fitted/trained with the 90% data portion. This prevents information from the validation data set from influencing the trained model (data leakage). Next, the validation and ML training data are converted into a normalized format. Finally, the trained scaler is saved and can be reused to convert new testing data.

5.3.3 Detection Layer

The detection layer is where the actual ML and DL training/evaluation takes place. This layer is split into two parts, with the first component covering anomaly detection and the second one covering classification. The anomaly detection component can detect zero-day ransomware attacks, while the classification component identifies the ransomware family and its behavior. Anomaly detection and classification algorithms are evaluated in parallel during live monitoring to determine the sensor's health (infected / not infected).

Anomaly Detection Component

The component that detects anomalies determines whether the observed behavior is normal or abnormal. For this purpose, the framework employs both ML and DL algorithms. For ML / DL training, the training data represents the sensor's "normal" state. Evaluation data, on the other hand, can contain both "normal" and "under attack" scenarios.

The ML algorithms used include One-Class SVM, LOF, and IF. For DL anomaly detection, Autoencoders are utilized.

This thesis compares the raw performance of these algorithms. Thus, no specific hyper-parameter tuning was performed (i.e., with cross-validation). Furthermore, future monitoring sessions may generate divergent data, i.e., when using a different Raspberry Pi computer or customizing the monitoring. As a result, the ideal parameters would differ.

Classification Component

The classification component's primary function is to classify a data sample appropriately. Normal behavior, Ransomware-PoC, DarkRadiation, and RAASNet are the possible classes to which a data sample can be assigned. Compared to anomaly detection, this method requires training data from each class. Furthermore, this data should be labeled and balanced. For multi-class classification, the supervised machine learning algorithms Support Vector Machine, Logistic Regression, Decision Tree, and Random Forest are utilized.

Evaluation

Specific metrics help evaluate the ML and DL algorithms. Depending on the category of training (supervised, unsupervised & semi-supervised), different performance indicators are employed. In this thesis, the "confusion matrix" is used as a basis for the evaluation. Diagram 5.5 shows the confusion matrix used for anomaly detection. The true positives (TP) refer to the number of ransomware infections correctly identified by the ML & DL algorithms, while the true negatives (TN) reflect correctly identified instances of normal behavior. False positive (FP) and false negative (FN) values indicate the number of incorrect predictions.

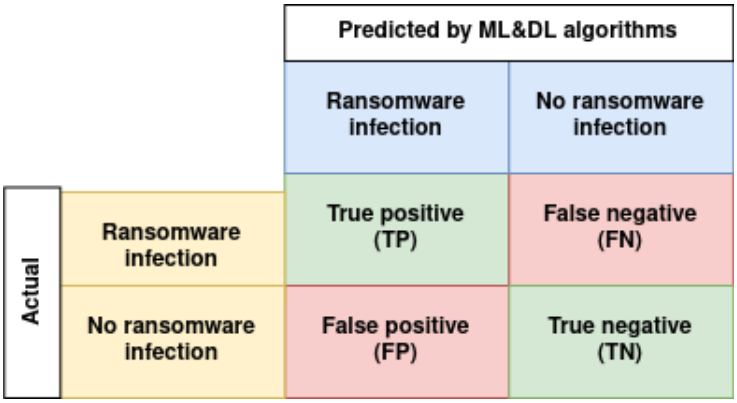

Figure 5.5: Confusion matrix in ransomware detection

In regards to anomaly detection, the following two performance metrics can be calculated:

1. The **True Positive Rate (TPR)**, indicating the proportion of correctly predicted ransomware infections from all actual ransomware infections:

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

2. The **True Negative Rate (TNR)**, indicating the proportion of correctly predicted instances of normal behavior from all actual instances of normal behavior:

$$TNR = \frac{TN}{TN + FP}$$

Similarly, one can build a confusion matrix for multi-class classification. Matrix labels would indicate different classes instead of the state of infection. In classification problems, the **TPR** also known as **recall** indicates the number of correct class predictions divided by the actual total number of class instances.

Example: Assume you have ten data samples with the known class DarkRadiation. The ML algorithm classifies five of these samples correctly and the other five incorrectly (into different classes). DarkRadiation's recall value would then be $\frac{5}{5+5} = 0.5$.

The following further performance metrics were used to evaluate classification performance:

1. **Accuracy** a metric comparing the accurate predictions compared to all predictions:

$$ACC = \frac{TP + TN}{TN + TP + FP + FN}$$

2. **Precision** measures the overall precision of correctly predicting a class out of all predictions of this class:

$$PPV = \frac{TP}{TP + FP}$$

Example: Assume you have eight data samples. You know that each class (Normal, DarkRadiation, RAASNet and Ransomware-PoC) has two samples assigned to them. The algorithm now determines, that five out of the eight samples belong to the Normal class. It's precision would then be calculated as follows: $\frac{2}{2+3} = 0.4$.

3. **F1-Score** combines the recall and precision metrics:

$$\text{F1-Score} = \frac{2TP}{2TP + FP + FN} = 2 * \frac{Precision * Recall}{Precision + Recall}$$

5.3.4 Visualization Layer

As noted earlier, the application contains a small "proof-of-concept" graphical user interface for presenting live evaluation data. The user can choose the algorithm and monitoring data they wish to see. Graphs help display this live evaluation data. A more detailed architectural overview, together with screenshots, is provided in the following chapter.

Chapter 6

Framework Implementation

This chapter reviews the implementation of the ransomware detection framework. As noted earlier, the framework utilizes a distributed architecture comprising two key components. The first is the Monitor Controller component which manages the three monitoring scripts. The second component is a Flask web application (Data Analysis Application) used for data pre-processing, ML training, and evaluation. In addition, REST API calls with HTTP Basic Authentication facilitate client-server (Monitor Controller-Flask web app) communication.

Note that installing both components on the Raspberry Pi platform is possible, but a number of prerequisites need to be full-filled. These include upgrading the Python version and ensuring machine resource consumption does not negatively impact spectrum monitoring.

6.1 Monitoring Scripts

In order to capture behavioral data on the Raspberry Pi sensor, three different monitoring scripts are utilized. Each of these scripts captures different aspects of the sensor's operation. In addition, each of the monitors utilizes the Linux Perf performance analysis tool to define the relevant machine features to monitor.

RES Monitor

This monitoring script tracks the resource consumption and device-specific events from the Raspberry Pi sensor. The program, written in Python, utilizes the library Psutil and the Linux utility Perf. The main code logic is split into classes, each tracking specific sensor metrics. There are two customization options for data output: CSV file creation or asynchronous messaging. The code snippet 6.1 shows the main monitoring loop, which executes every five seconds. Every data sample represents five seconds of monitoring data.

```
def start(self):
   while self.monitor:
     self.log.verbose('Measuring...')
     data = self.monitorService.monitor()
     if (data == 0):
       self.monitor = False
   self.log.log('END')
   self.graceful_shutdown()
```

Listing 6.1: Code snippet from main monitoring loop of monitor RES

KERN Monitor

The KERN Monitor script monitors events related to disk I/O, CPU, kernel memory, and system calls and is written in bash. The code snippet 6.2 shows the body of the main monitoring loop. This code is executed every five seconds. A DNS server is pinged as a first step to confirm the sensor's connection to the internet. Perf will then monitor the user-defined events for the specified time window. In the case of the thesis, this was set to a five-second interval.

```
# Checks the internet connection of the sensor:
if ping -q -c 1 -W 1.5 8.8.8.8 >/dev/null; then
       connectivity="1"
else
       connectivity="0"
fi

timestamp=$(date +%s)
if [ "$resourceMonitor" = true ]
then
       oldNetworkTraffic=$(ifconfig | grep -oP -e "bytes \K\w+" | head -n
           4)
fi

# Perf will monitor the events:
perf stat -e "$targetEvents" --o "$tempOutput" -a sleep "
   $timeWindowSeconds"
```

Listing 6.2: Code snippet from main monitoring loop of monitor KERN

SYS Monitor

The SYS Monitor bash script records system call information repeatedly over a ten-second interval. The Perf utility uses the command "trace" to capture all system calls executed on the sensor and saves these in a log file.

```
1 echo "Time now: " "$Unix_time_current"
2 echo "Gathering Syscalls"
3 UPTIME=$(cat /proc/uptime | awk '{print $1}')
4 EPOCH=$(date +%s.%3N)
5 Date_Hourly=$(date +%s)
6 perf trace -S -T -o "$RESULTS_PATH/""${Date_Hourly}".log -e !nanosleep -a
      -- sleep "$SLEEP"
7 echo -e "EPOCH: $EPOCH \nUPTIME:$UPTIME" >> "$RESULTS_PATH/""${
     Date_Hourly}".log &
8 Unix_time_current=$(date +%s)
9 counter=$((counter+1))
```

Listing 6.3: Code snippet from main monitoring loop of monitor SYS

6.2 Monitor Controller

The Monitor Controller is installed on the Raspberry Pi sensor. It provides multiple customization possibilities while controlling the entire monitoring process. It also manages and records the status and elapsed time of current and past monitoring sessions. This command-line interface tool is written in Python 3.5 and uses an SQLite database for data persistence. In addition, the Python module "click" is utilized for argument parsing and allows for simple user commands to be supplied, such as show, collect and send. Finally, the "requests" Python package facilitates communication with the server via REST API calls. A simple installer script was also developed to download and install the required dependencies automatically.

The monitoring scripts (RES, KERN, and SYS) are controlled by the Monitor Controller and implemented as "systemd" services. For example, the code snippet 6.4 shows the "systemd" service implemented for the KERN Monitor script. The following identifies the key benefits of deploying the scripts as a service:

1. Provides a simple way to start, stop and control the monitoring scripts.
2. Provides the ability to manage the monitoring scripts as services with systemctl.
3. Allows for defining the behavior after a program failure or reboot of the device.
4. Streamlines the integration of the monitoring scripts into the Monitor Controller.

```
1 [Unit]
2 Description=Monitor HPC & Ressources Dr.Huertas
3 After=multi-user.target
4
5 [Service]
6 Type=simple
```

```
7  Restart=on-failure
8  User=root
9  ExecStart=/bin/bash /root/BA_Thesis_PI/monitors/monitor2/
       new_sampler_50tmp.sh
10
11 [Install]
12 WantedBy=multi-user.target
```

Listing 6.4: systemd service of the KERN Monitor

6.2.1 Monitoring Procedure

Flow diagram 6.1 illustrates the procedure of collecting training or evaluation data with the Monitor Controller. As a first step, the user calls the endpoint "collect." Here, they can define the parameters for monitoring, such as the time, the server address, and the monitoring services they wish to use. The Monitor Controller then validates these arguments and tries to establish an SSH connection to the server.

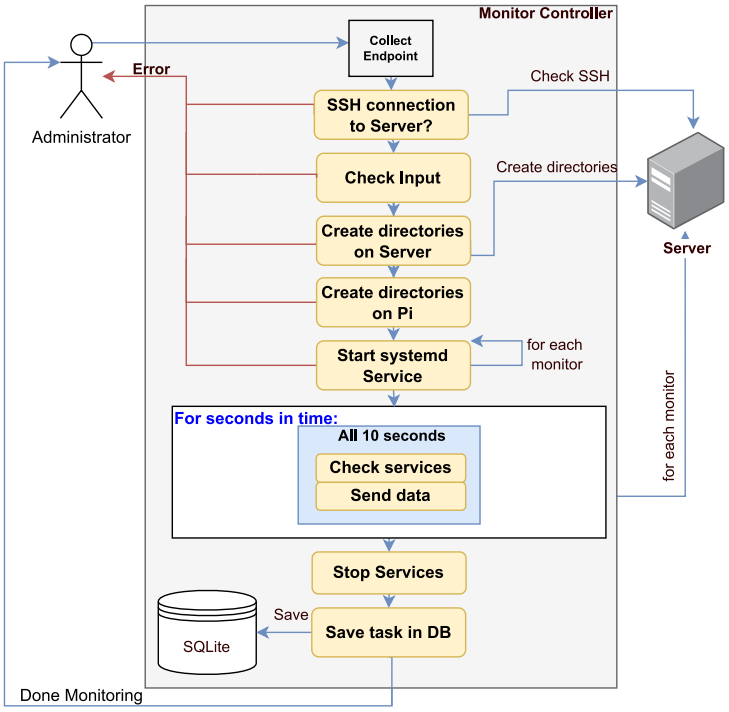

Figure 6.1: Monitoring procedure of Monitor Controller

Finally, the main loop 6.5 is executed after creating the monitoring directories on both the device and the server and starting the specified monitoring services.

```
1 start = time.perf_counter()
2 with click.progressbar(range(seconds)) as progress:
3     for value in progress:
```

```
4         time.sleep(1)
5         if (total % 10 == 0) and (total != 0):
6             t1 = threading.Thread(target=thread_work, args=(server,
                  active_services, total))
7             t1.start()
8         total += 1
9 finish = time.perf_counter()
```

Listing 6.5: Main loop Monitor Controller

The main monitoring loop runs every second for the specified duration. Services are restarted if necessary during their execution, and newly collected monitoring data is sent to the server. After exiting the main program loop, the Monitor Controller will stop all services and save the monitoring task data in the database.

6.3 Data Analysis Application

This application was developed using the Flask open-source web microframework for Python. Flask utilizes a Model-View-Controller (MVC) design pattern as its architectural foundation. It is a well-documented and user-friendly framework that integrates well with ML/DL-specific Python packages.

The following diagram 6.2 provides an overview of the MVC design architecture. A more detailed description of each component is shown below.

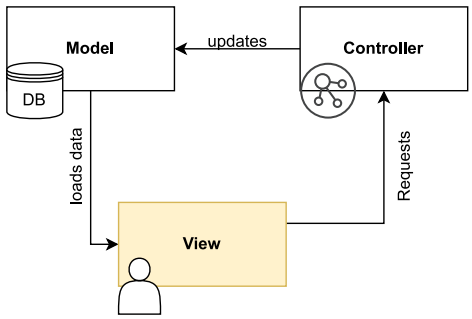

Figure 6.2: Overview of the MVC Model including the View component

6.3.1 MVC Controllers

The controllers are the REST endpoints through which the sensor's Monitor Controller and user can communicate with the Data Analysis Application. The controller's primary function is to manage the request flow by acting as an intermediary between the view and the model. The controller can accept requests from the sensor's Monitor Controller or the user. The received data is then validated and transformed into a format usable by the program and sent to the model for further processing.

The sensor and the user each have their own controllers. Sensor-specific controllers forward incoming request data to the model in a background thread. A response is then directly sent back to the sensor Monitor Controller middleware.

User controllers, on the other hand, handle user input from the Flask graphical user interface (Visualization Layer). Data is exchanged between the controller, the model, and the view in this instance. The data is presented to the user via the application View. The following represent the main controllers of the application:

Sensor-specific controllers:

- **/rest/main** Main endpoint for training & evaluating ML models
- **/rest/live** Handles the evaluation of live data (all 5 seconds)
- **/rest/test** Controller to check, if the Flask application is running on the server

User-specific controllers:

- **/live** Graphical representation of classification and anomaly detection evaluation. Endpoint for accessing the GUI.

6.3.2 MVC Model

The model contains the core application logic, including ML training/evaluation, data pre-processing, and interactions with the database. Service classes, each with specific functionality, manage the application's core logic. This design allows for future expansion of the web framework and simplifies code maintenance. The following service classes are implemented in the framework and cover the main functionalities:

1. **KERN/RES/SYS**: Pre-process data for monitor RES, KERN and SYS
2. **KERNlive/RESlive/SYSlive**: Pre-process data for the tree monitors (live monitoring)
3. **anomalyml/anomalydl**: Train & evaluate anomaly detection algorithms
4. **classificationml**: Train & evaluate classification algorithms

Training Procedure Diagram

The following diagram 6.3 displays the main steps and class interactions implemented within the Flask application, as part of the training process.

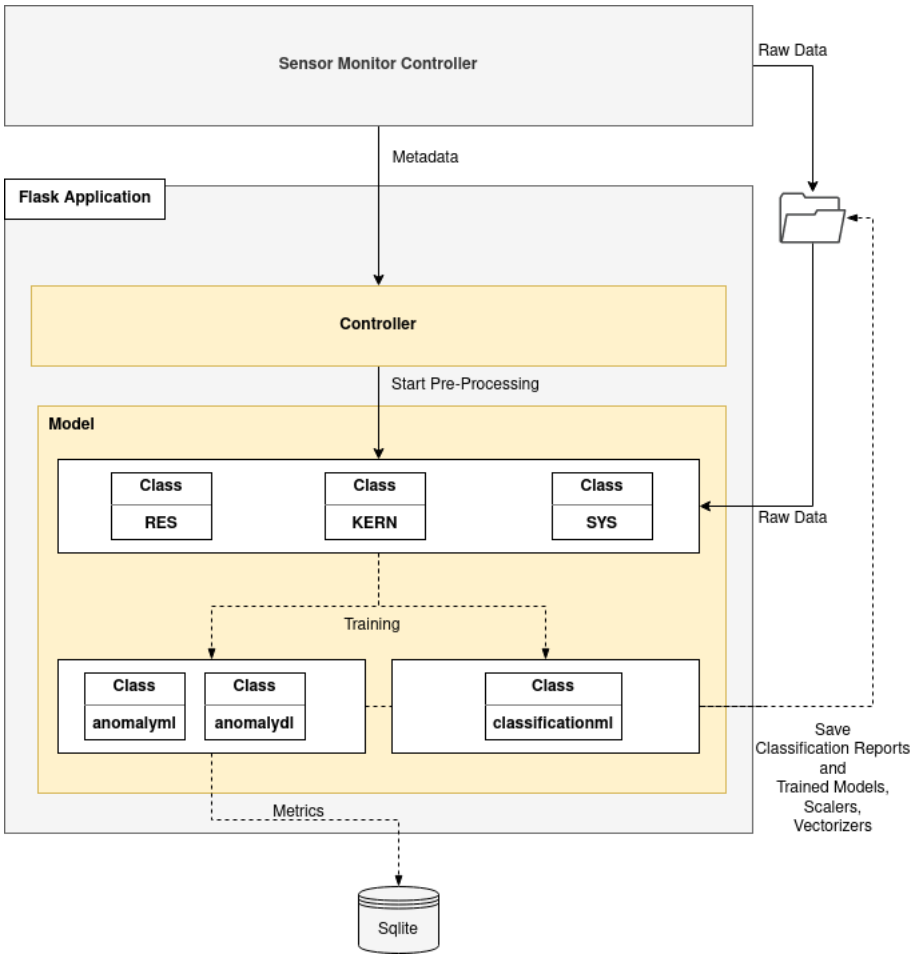

Figure 6.3: Interaction between service classes in training

Live Evaluation Procedure Diagram

Figure 6.5 documents the live evaluation process.

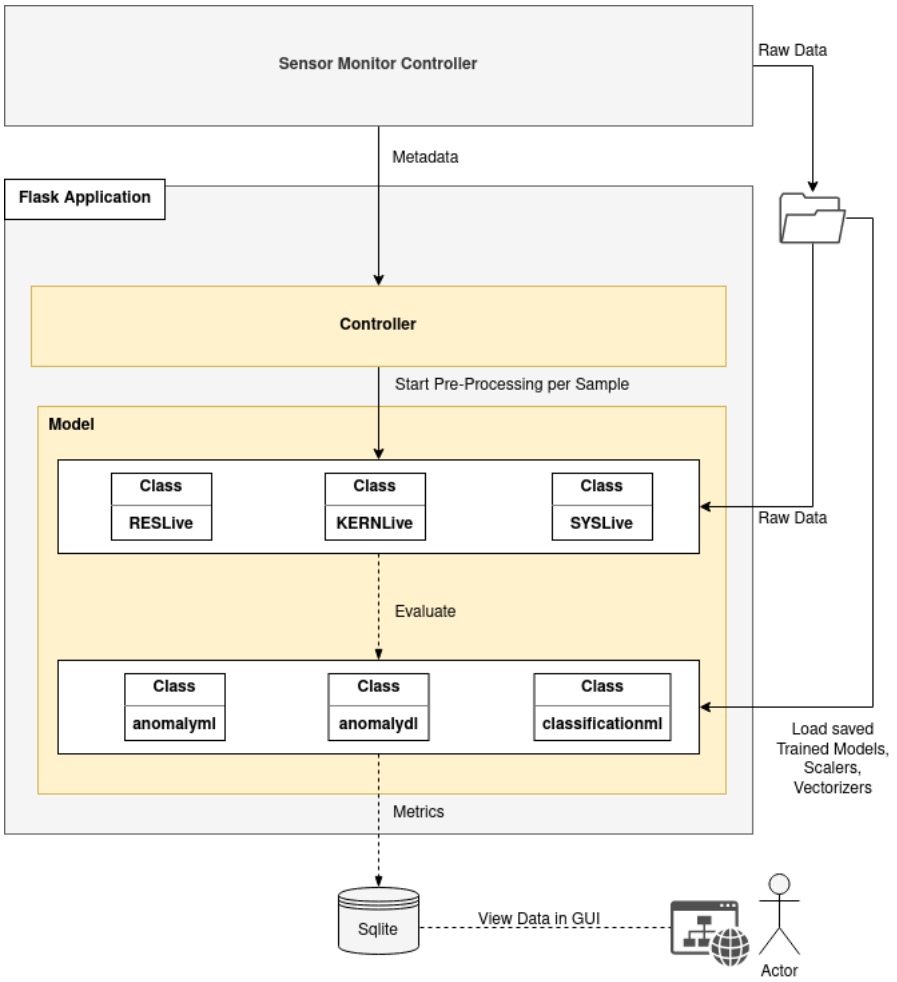

Figure 6.4: Interaction between service classes during live evaluation

Anomaly Detection and Classification

Pyod and Tensorflow are the two main Python packages employed for ML and DL. After pre-processing the incoming data, the ML/DL models are run in either training or evaluation modes. The following sub-sections will introduce the main training procedures for anomaly detection and classification.

ML Anomaly Detection

The following code snippet 6.6 shows a part of the training loop used for ML anomaly detection (service class anomalyml). As a first step, the anomaly detectors are defined. The fraction of outliers present in the training data set, also known as the contamination factor, is set to 5% in the code.

Next, a training session is initialized for each pre-processed monitoring data set.

After this, the input data is split into a training data set (90%) and validation data set (10%), and both are standardized. The program will then perform the following processing steps:

1. Using the training data set, the detectors are trained. This results in trained models which can detect anomalies.
2. Next, the validation data is evaluated with the trained models to predict its anomaly scores (normal=0/abnormal=1)
3. The accuracy of the model is determined by comparing the predicted and actual anomaly scores. As only the TN's and FP's can be determined, the accuracy equals the TNR in this case.

```
1  #Defined contamination + algorithms to train:
2  contamination_factor = 0.05
3  detectors = {
4      'IsolationForest':IForest(random_state=42, contamination=
           contamination_factor),
5      'OneClassSVM':OCSVM(kernel='rbf',gamma=0.0001, nu=0.3, contamination=
           contamination_factor),
6      'LocalOutlierFactor': LOF(n_neighbors=50, contamination=
           contamination_factor)
7  }
8  # Checking all paths:
9  ...
10 for featurename, corpus in features[2].items():
11     y = [0 for i in range(0,len(corpus))]
12     # Split the data into training and testing:
13     X_train, X_test, y_train, y_test = train_test_split(corpus, y,
           test_size=0.1, random_state=42, shuffle=True)
14     # Scaling the data:
15     scaler = StandardScaler()
16     scaler.fit(X_train)
17     X_train = scaler.transform(X_train)
18     X_test = scaler.transform(X_test)
19     # Save the scaler:
20     ...
21     # Save preprocessed data:
22     ...
23     # Train Loop:
24     for detector_name, dectector in detectors.items():
25         start_train = time.time()
26         dectector.fit(X_train)
27         end_train = time.time()
28         ...
```

```
        # Save the model:
        ...
        # Predict the test data:
        start_prediction = time.time()
        y_pred = dectector.predict(X_test)
        end_prediction = time.time()
        ...
        # Calculate the accuracy:
        accuracy = accuracy_score(y_test, y_pred)
        # Save the metrics in db:
        ...
```

Listing 6.6: Anomaly detection training loop

DL Anomaly Detection

Training for DL anomaly detection is performed in a similar manner. Code snippet 6.7 presents the Autoencoder class used for detecting anomalies.

```
class AnomalyDetector(Model):
    def __init__(self, input_dim, layer_one, layer_two, layer_three,
        layer_four):
        super(AnomalyDetector, self).__init__()
        self.encoder = Sequential([
            Dense(layer_one, activation="relu"),
            Dense(layer_two, activation="relu"),
            Dense(layer_three, activation="relu"),
            Dense(layer_four, activation="relu")])

        self.decoder = Sequential([
            Dense(layer_three, activation="relu"),
            Dense(layer_two, activation="relu"),
            Dense(layer_one, activation="relu"),
            Dense(input_dim, activation="sigmoid")])

    def call(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded
```

Listing 6.7: DL Autoencoder

After splitting and standardizing the training data into the 90%-10% split, the Autoencoder is ready to be trained. As the number of features can vary depending on the training data set, the dimensions of the Autoencoder layers are calculated before instantiation. In addition, the TensorFlow "Early Stopping Callback" prevents overfitting the

model by stopping the training procedure when the validation error (mean absolute error) no longer decreases.

After training the Autoencoder, the validation data set is processed and its reconstruction errors calculated. The Autoencoder uses two different thresholds for detecting anomalies. The first threshold is specified by the interquartile range rule, while the second is defined as two standard deviations away from the mean of the reconstruction errors. The anomaly detection thresholds and reconstruction errors are then used to assign the anomaly scores to the validation data (0=normal, 1=abnormal). From this, the TNR can be determined.

```
1  # Check paths:
2  ...
3  for featurename, corpus in features[2].items():
4      y = [0 for i in range(0,len(corpus))]
5      X_train, X_test, y_train, y_test = train_test_split(corpus, y,
           test_size=0.1, random_state=42, shuffle=True)
6      # Load scalers and scale data:
7      ...
8      # Define hidden layers:
9      ...
10     # Train Model:
11     model = AnomalyDetector(input_dim, layer_one, layer_two, layer_three,
           layer_four)
12     early_stopping = EarlyStopping(monitor="val_loss", patience=10, mode="
           min")
13     model.compile(optimizer='adam', loss="mae", metrics=["mae","accuracy"
           ])
14     start_training = time.time()
15     model.fit(X_train, X_train, epochs=500, batch_size=120, shuffle=True,
           validation_data=(X_test, X_test), callbacks=[early_stopping],
           verbose=0)
16     end_training = time.time()
17     ...
18     # Calculate the thresholds for anomaly detection:
19     start_prediction = time.time()
20     IQR_lower, IQR_upper, STD_lower, STD_upper, train_loss = anomalydl.
           find_threshold(model, X_test)
21     end_prediction = time.time()
22     ...
23     # Anomaly score using standard deviation:
24     y_pred_STD = anomalydl.append_labels(train_loss, STD_lower, STD_upper)
25     accuracy_STD = accuracy_score(y_test, y_pred_STD)
26     # Anomaly score using interquartile range rule:
27     y_pred_IQR = anomalydl.append_labels(train_loss, IQR_lower, IQR_upper)
28     accuracy_IQR = accuracy_score(y_test, y_pred_IQR)
29     # Save model + metrics:
30     ...
```

Listing 6.8: Main DL training function calls

Classification

The following classification algorithms are used as part of multi-class classification:

1. Logistic Regression
2. SVM
3. Random Forest
4. Decision Tree

Unlike anomaly detection, classification algorithms require labeled and balanced data from all classes. Code snippet 6.9 shows the main training steps. The procedure is similar to that used in training ML anomaly detection algorithms. Each pre-processed incoming data set is split and standardized. After this, the classifiers are trained (using the training data with its respective class labels). In this scenario, the labels are the class names (Normal, DarkRadiation, Ransomware-Poc, and RAASNet).

A classification report is generated by comparing the actual class labels with the predicted labels of the validation data. This report contains the overall classification accuracy as well as the recall, precision, and F1-Score values of each class.

```
1  classifiers = {
2            'LogisticRegression': LogisticRegression(solver='saga',
                multi_class='ovr', max_iter=5000),
3            'DecisionTreeClassifier': DecisionTreeClassifier(),
4            'SVC': SVC(gamma='auto', kernel='rbf'),
5            'RandomForestClassifier': RandomForestClassifier()
6        }
7        for featurename, corpus in features[2].items():
8            # Behavioral features:
9            y = features[1]
10
11           # Create a train-test split:
12           X_train, X_val, y_train, y_val = train_test_split(corpus, y,
                test_size=.1, shuffle=True, random_state=42)
13
14           # Standardize the data:
15           scaler = StandardScaler()
16           scaler.fit(X_train)
17           X_train = scaler.transform(X_train)
18           X_val = scaler.transform(X_val)
```

```
19            corpus = scaler.transform(corpus)
20            # Save the scaler:
21            ...
22            # Create a datatframe:
23            ...
24            # Train the models:
25            for name, clf in classifiers.items():
26                clf.fit(X_train, y_train)
27                # Save the model:
28                ...
29                # Evaluate the model:
30                y_pred = clf.predict(X_val)
31                # Create and save classification report:
32                ...
```

Listing 6.9: Logistic Regression Training

6.3.3 MVC View

The MVC View dynamically presents data to the user (Visualization Layer). User inputs are processed by the view and sent to the controller to generate an appropriate response. This thesis has implemented a small "proof-of-concept" graphical user interface. Its primary function is to process user input and visually present the results of anomaly detection and classification (from live/online monitoring). This tool uses the following technologies:

- Jinja2 as the templating engine
- JQuery for DOM manipulation
- Bulma CSS for styling
- Chart js for graphing data

The following is a sample screenshot from the Flask GUI:

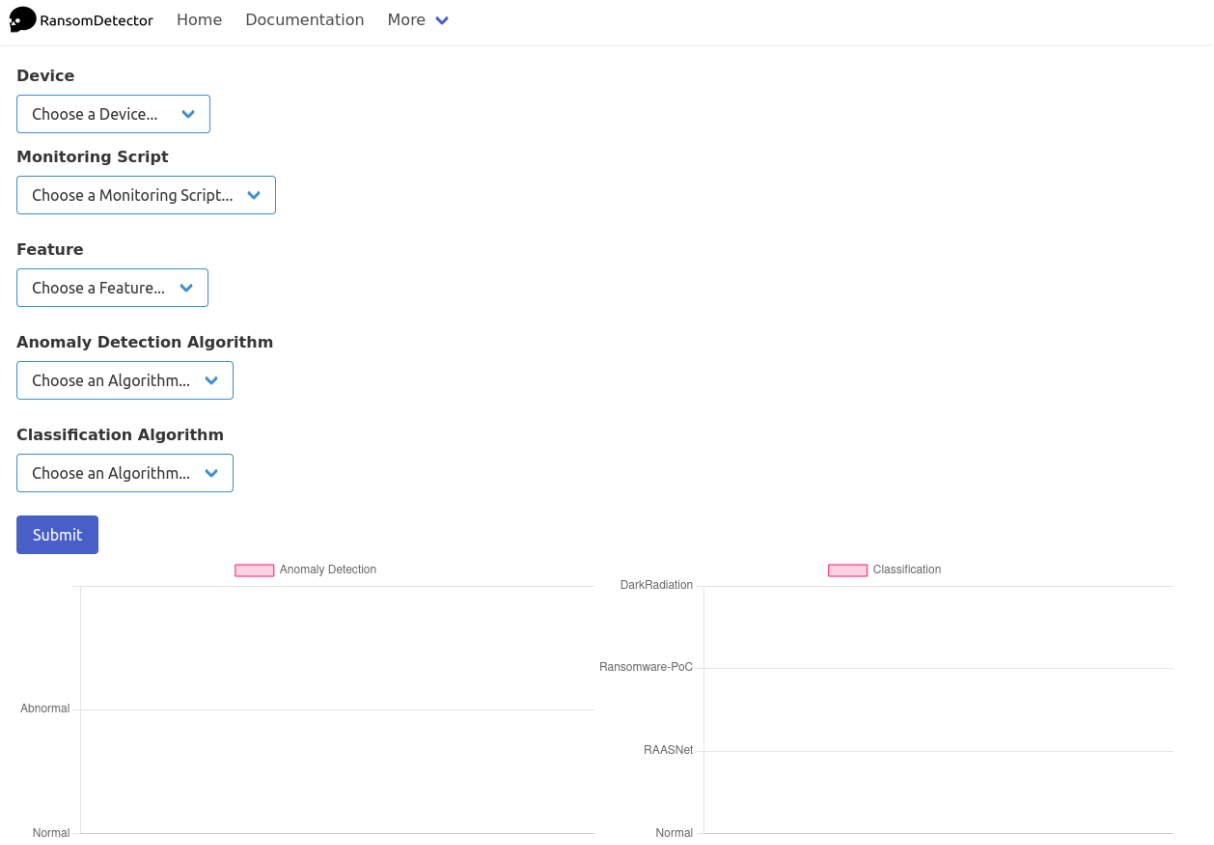

Figure 6.5: Proof of Concept Flask GUI

Chapter 7

Experiments & Analysis

This chapter introduces the reader to the different anomaly detection and classification experiments conducted as part of this thesis, together with the results obtained and associated resource utilization. In addition, observations and associated recommendations are provided. Finally, this chapter compares ML/DL and policy-based anomaly detection and classification.

7.1 Anomaly Detection Experiment

The goal of the anomaly detection experiment was to evaluate and compare the performance and effectiveness of varying ML/DL algorithms in identifying ransomware. Furthermore, this experiment attempted to distinguish the best algorithms and features for detecting ransomware.

7.1.1 Experimental Setup

The first stage of this experiment was to gather the required training data. Anomaly detection algorithms need large volumes of training data devoid of outliers (normal behavior). The Monitor Controller and the monitoring scripts were deployed on the sensor to achieve this. The monitoring session was then run for 15 hours to ensure an adequate training data sample size. Next, the three monitoring scripts (RES, KERN, SYS) were run in parallel by executing the command "Collect" on the Monitor Controller. Once this data gathering process was completed, the "Send" command was called to initiate the server's training procedure.

The server used for this experiment was a Lenovo ThinkPad with an Intel i7-8750H processor and 32 GB of RAM running EndeavourOS Linux.

Before executing each of the ransomware samples on the sensor, the following tasks were completed:

1. A fresh copy of the ElectroSense software was installed alongside the Monitor Controller.

2. Data partition three of the Raspberry Pi was resized to extend its limited storage capacity.

3. Sample files with specific file extensions were generated using a file creation script in a specified directory.

RAASNet and Ransomware-PoC begin the file encryption process shortly after execution starts (after approximately two seconds). The DarkRadiation ransomware, on the other hand, first installs several prerequisite dependencies before starting the file encryption process. To compare the three ransomware strains, the monitoring procedure of DarkRadiation was initiated after receiving the "encrypt home files began" notification.

Each ransomware monitoring session ran for 15 minutes, with all monitoring scripts running in parallel. This data was then used to evaluate the detection capabilities of the ML/DL algorithms.

Ransomware-PoC was utilized in a second attack scenario to test the capability of encrypting the entire sensor. It took only 2 minutes and 27 seconds for the ransomware to crash the sensor due to the encryption of system and library files.

7.1.2 Results & Findings

The first step in the analysis process was to verify the ML/DL algorithms to identify normal behavior on the sensor correctly. As the collected data should reflect the normal state of the sensor, a TNR close to 100% was expected. The results received from the 10% validation data sample were as anticipated.

A smaller "normal" data sample was monitored and validated two hours later. It was expected to return similar results. This is where the first deviation was observed. LOF, One-Class SVM, and the Autoencoders trained with hashed system call features predicted the normal behavior to be anomalous (0.00% TNR). As these algorithms were unable to detect normal sensor behavior, their ransomware detection performance should be ignored. This can be seen in the column TNR Val in table 7.1.

Training time seems to be dependent on the ML/DL model and the number of features used. DL Autoencoders showed the most extended run-times, followed by the One-Class SVM algorithm. As more features were included in the extracted system-call data, the processing time required increased.

ML and DL models trained with system call monitoring data (Monitor SYS) detected the ransomware DarkRadiation without issue. However, most of these ML algorithms failed to identify a Ransomware-PoC or RAASNet infection. Deep Learning seems to be more promising. The DL Autoencoder with an interquartile-based threshold using the Tf-idf system call features were able to recognize all of the ransomware samples (TPR >50%).

Similarities emerge when comparing the detection performance of algorithms trained with the RES and KERN monitoring data. Both DarkRadiation and Ransomware-PoC were detected accurately. However, a RAASNet ransomware infection appeared to be more challenging for the algorithms to identify. This is most likely due to its slow encryption process (slowest out of the ransomware samples).

The best performing algorithm for KERN and RES seems to be LOF, having a good TNR, the fastest training time, and the best ransomware detection accuracy of 100% over all ransomware samples.

Table 7.1: Anomaly Detection Results

Monitor	Features	Algorithm (ML/DL)	Training Time (s)	Testing time per sample (s)	TNR Val (%)	TPR Dark (%)	TPR PoC (%)	TPR RAAS (%)
RES Monitor	114	IF	0.79	0.030	98.10	95.88	96.94	75.78
	114	One-Class SVM	2.05	0.001	96.19	100.00	100.00	94.53
	114	LOF	0.19	0.240	93.33	100.00	100.00	100.00
	114	Autoencoder STD	16.49	0.160	97.14	100.00	100.00	38.28
	114	Autoencoder IQR	16.49	0.160	96.19	100.00	100.00	80.47
KERN Monitor	77	IF	0.62	0.033	98.21	94.74	93.52	55.30
	77	One-Class SVM	2.20	0.001	94.64	100.00	100.00	90.90
	77	LOF	0.16	0.009	90.17	100.00	100.00	100.00
	77	Autoencoder STD	17.49	0.083	98.21	100.00	100.00	30.30
	77	Autoencoder IQR	17.49	0.083	98.21	100.00	100.00	56.06
Hashing 1-gram SYS	1024	IF	7.81	0.032	96.29	91.89	67.57	52.69
	1024	One-Class SVM	8.72	0.002	0.00	100.00	100.00	17.20
	1024	LOF	0.51	0.016	0.00	100.00	100.00	19.35
	1024	Autoencoder STD	7.04	0.102	0.00	100.00	100.00	12.90
	1024	Autoencoder IQR	7.04	0.102	0.00	100.00	100.00	27.96
Frequency 1-gram SYS	2443	IF	9.36	0.038	96.29	86.49	56.76	41.94
	2443	One-Class SVM	24.03	0.004	100.00	100.00	2.70	15.05
	2443	LOF	0.93	0.029	100.00	62.16	0.00	6.45
	2443	Autoencoder STD	36.82	0.098	100.00	100.00	37.84	30.11
	2443	Autoencoder IQR	36.82	0.098	97.53	100.00	59.46	64.51
TF-IDF 1-gram SYS	2443	IF	9.68	0.051	96.29	89.18	67.57	51.61
	2443	One-Class SVM	24.49	0.005	100.00	100.00	8.11	17.20
	2443	LOF	1.00	0.038	100.00	56.76	1.35	9.68
	2443	Autoencoder STD	36.27	0.092	100.00	100.00	51.35	26.88
	2443	Autoencoder IQR	36.27	0.092	96.29	100.00	71.62	79.57

7.1.3 Recommendations

After completing the experiments and reviewing the results, the following recommendations can be made for future investigation.

- As noted above, the 15-hour monitoring time was appropriate for this experiment. Still, in a production environment, optimal results would be achieved by having a more representative (more significant) data sample of normal sensor behavior.

- The RES and KERN Monitor generally achieved the most accurate ransomware detection results. However, as system call monitoring tends to be more inaccurate, it is recommended not to use these for future ransomware detection.

7.2 Classification Experiment

The goal of the classification experiment was to evaluate and compare the performance and effectiveness of different ML algorithms in the classification of ransomware. For this experiment, four behavioral classes were defined. These include:

- The sensor's normal behavior
- DarkRadiation infection
- RAASNet infection
- Ransomware-PoC infection

7.2.1 Experimental Setup

The first step of this experiment was to collect four separate samples of training data. These reflect the different states of the machine and are the aforementioned behavioral classes.

Classification algorithms need training data from each class and perform best when sample sizes are comparable. Each of these data collection sessions ran for approximately four hours. Note that the standard storage capacity of the sensor was insufficient to provide the required number of target files for encryption. Therefore, an external solid state drive (SSD) containing 100 gigabytes of sample "dummy" files was used to achieve a four-hour monitoring session. These include files of varying sizes and file types.

As part of the pre-processing, the server application labeled each monitored sample with the appropriate behavioral class label and combined these into a single file. Then, the data was prepared for training by randomly shuffling, splitting, and normalizing. The classification label is also required for training as this is supervised ML. The server for this experiment is the identical Lenovo laptop utilized in the anomaly detection experiment.

7.2.2 Results & Findings

Exceptional classification results were observed for all monitoring scripts (SYS, RES, and KERN) with an overall accuracy of 90-100%. However, it is not known whether the short monitoring periods impacted the outcome of this experiment (only 4 hours per class). Table 7.2 presents the macro and weighted average F1-scores calculated for the different classification algorithms. Note that the reports on table 7.3 reflect only the results from the Decision Tree Classifier algorithm (Best performance to training time). Training duration appears to be linked to the number of extracted features. Moreover, the training period for logistic regression is the longest for all data sets (RES, KERN & SYS).

Table 7.2: F1-Scores Classification

Monitor	Features	Algorithm	Training Time (s)	Testing time per sample (s)	Macro avg F1-score	Weighted avg F1-score
RES Monitor	114	Logistic Regression	50.00	0.0001	0.99	0.99
	114	Decision Tree	0.27	0.0003	0.99	0.99
	114	Support Vector Machine	0.22	0.0006	0.99	0.99
	114	Random Forest	1.50	0.0130	0.99	0.99
KERN Monitor	77	Logistic Regression	34.94	0.0002	0.99	0.99
	77	Decision Tree	0.15	0.0002	0.99	0.99
	77	Support Vector Machine	0.26	0.0004	0.98	0.98
	77	Random Forest	1.15	0.0120	0.98	0.98
Hashing 1-gram SYS	1024	Logistic Regression	124.93	0.0001	1.00	1.00
	1024	Decision Tree	0.18	0.0001	1.00	1.00
	1024	Support Vector Machine	3.03	0.0013	0.93	0.93
	1024	Random Forest	0.69	0.0159	1.00	1.00
Freq. 1-gram SYS	2084	Logistic Regression	349.24	0.0003	1.00	1.00
	2084	Decision Tree	0.21	0.0002	1.00	1.00
	2084	Support Vector Machine	7.99	0.0082	0.99	0.99
	2084	Random Forest	0.65	0.0130	1.00	1.00
TF-IDF 1-gram SYS	2084	Logistic Regression	326.19	0.0004	1.00	1.00
	2084	Decision Tree	0.25	0.0003	1.00	1.00
	2084	Support Vector Machine	7.88	0.0029	0.99	0.99
	2084	Random Forest	0.81	0.0120	1.00	1.00

Table 7.3: Classification Reports of Decision Tree Classifier algorithm

Monitor	Accuracy	Classes	Precision	Recall	F1-score
RES Monitor	0.99	DarkRadiation	0.99	0.98	0.98
		Normal	0.97	0.99	0.98
		Ransomware-PoC	1.00	1.00	1.00
		RAASNet	0.99	0.99	0.99
KERN Monitor	0.99	DarkRadiation	0.99	0.98	0.98
		Normal	0.98	0.98	0.98
		Ransomware-PoC	1.00	1.00	1.00
		RAASNet	0.99	1.00	0.99
Hashing 1-gram SYS	1.00	DarkRadiation	1.00	1.00	1.00
		Normal	1.00	1.00	1.00
		Ransomware-PoC	1.00	1.00	1.00
		RAASNet	1.00	1.00	1.00
Freq. 1-gram SYS	1.00	DarkRadiation	1.00	1.00	1.00
		Normal	1.00	1.00	1.00
		Ransomware-PoC	1.00	1.00	1.00
		RAASNet	1.00	1.00	1.00
TF-IDF 1-gram SYS	1.00	DarkRadiation	1.00	1.00	1.00
		Normal	1.00	1.00	1.00
		Ransomware-PoC	1.00	1.00	1.00
		RAASNet	1.00	1.00	1.00

7.2.3 Recommendations

The experiment yielded excellent results. However, it raises the question of why anomaly detection was not as accurate, specifically with system calls. Therefore, it is recommended to conduct further study in this area.

As with the anomaly detection experiment, it is felt that using a more significant "normal" data sample would improve the accuracy of the classification results. This would, of course, place additional demands on the sensor's resources, specifically data storage and processing time.

7.3 Performance Metrics Experiment

This experiment aimed to determine the resource utilization of the entire framework when deployed on a standard sensor. The objective was to decide whether the Raspberry Pi sensor has the CPU and memory capacity to perform spectrum monitoring and successfully run the framework in parallel.

7.3.1 Experimental Setup

This thesis defines a standard sensor as a Raspberry Pi 3 Model B with 1GB of RAM and a Quad-Core 1.2GHz Broadcom BCM2837 64bit CPU with a 16 gigabyte SD card for storage.

The analysis established a baseline by measuring the sensor's CPU and memory consumption during normal operation (10 minutes) and averaging the results.

Similarly, resource consumption data was obtained while running the Monitor Controller and the Flask application separately on the sensor. By subtracting baseline values from the new measurements, the resource consumption of the framework's components were isolated.

Code snippet 7.1 shows the main monitoring loop of the resource consumption monitoring script. It tracks the CPU and memory utilization for every 0.01-second interval.

```
1 while True:
2     cpu_usage = psutil.cpu_percent(percpu=False)
3     memory_usage = psutil.virtual_memory().percent
4     # If a keyboard interrupt is received, break the loop
5     try:
6         # Write the current ressources to the .csv file
7         file.write(str(time.time()) + "," + str(cpu_usage) + "," + str(
              memory_usage) + "\n")
8         # Sleep for 0.01 seconds
9         time.sleep(0.01)
```

```
10      except KeyboardInterrupt:
11          break
```

Listing 7.1: Main Loop Resource Consumption Script

7.3.2 Resource Usage of the Monitor Controller

The resource consumption of the Monitor Controller was assessed for four distinct monitoring configurations, shown in table 7.4. The results were obtained by performing ten-minute monitoring sessions and subtracting them from the baseline.

Table 7.4: Monitor Controller Resource Usage

Monitor configuration in MC	**CPU consumption**	**Memory consumption**
Monitor Controller + RES Monitor	2.02%	2.80%
Monitor Controller + KERN Monitor	4.47%	2.37%
Monitor Controller + SYS Monitor	13.58%	11.90%
Monitor Controller + KERN + RES + SYS	29.50%	13.24%

7.3.3 Resource Usage of Data Analysis Application

The resource consumption of the Flask application on the sensor was evaluated by running a classification and anomaly detection training procedure for each monitor (KERN,RES & SYS). In addition, timestamps were employed to track specific events such as the pre-processing, training, and evaluation phases. Only the performance metrics of pre-processing the data for anomaly detection were measured, as the classification data pre-processing procedure is almost the same. The values indicated by ”nan” mean that no resource metrics could be determined for the given interval. It was observed that the RES and KERN Monitors produced data volumes manageable on the sensor. The SYS Monitor, on the other hand, resulted in significantly larger data samples. Given the duration of monitoring performed during the experiments and the thesis recommendation to extend these, it is felt that the sensor's machine resources would be inadequate for processing system call data.

Resource Usage Training/Evaluation of RES

The goal of this experiment was to measure the resource utilization of the sensor when training/evaluating ML/DL algorithms using RES monitoring data. For anomaly detection, 2133 data samples were used and 4266 for classification. 10% of the data was utilized for evaluation. The following table 7.5 summarizes the findings:

Table 7.5: Training & Evaluation of RES

Category	Task	CPU (%)	Memory (%)	Seconds (s)
Pre-Processing	Pre-processing RES	17.74	50.90	1.01
Anomaly detection ML	Training IF	31.19	52.40	3.75
	Evaluating IF	27.22	52.40	1.04
	Training One-Class SVM	26.48	52.45	1.11
	Evaluating One-Class SVM	39.74	52.50	0.52
	Training LOF	31.18	52.69	0.85
	Evaluating LOF	56.05	53.18	0.20
Anomaly detection DL	Training Autoencoder	38.41	54.59	98.40
	Evaluating Autoencoder	30.40	55.37	1.41
Classification ML	Training Logistic Regression	29.38	55.38	198.87
	Evaluating Logistic Regression	47.37	55.40	0.07
	Training Decision Tree	29.00	55.40	2.40
	Evaluating Decision Tree	nan	nan	0.006
	Training Support Vector Machine	30.84	56.99	13.67
	Evaluating Support Vector Machine	30.60	55.60	6.61
	Training Random Forest	29.35	56.05	13.45
	Evaluating Random Forest	11.26	57.60	1.32

Resource Usage Training/Evaluation of KERN

The same experiment was performed with an equal number of data samples from the KERN Monitor. Table 7.6 shows the results of the experiment:

Table 7.6: Training & Evaluation of KERN

Category	Task	CPU (%)	Memory (%)	Seconds (s)
Pre-Processing	Pre-processing KERN	25.19	45.58	0.90
Anomaly detection ML	Training IF	37.20	46.56	3.72
	Evaluating IF	33.90	46.6	1.05
	Training One-Class SVM	31.60	46.74	0.87
	Evaluating One-Class SVM	33.75	46.8	0.38
	Training LOF	35.29	47.09	0.77
	Evaluating LOF	63.15	47.48	0.15
Anomaly detection DL	Training Autoencoder	35.26	48.98	110.72
	Evaluating Autoencoder	4.18	49.50	42.09
Classification ML	Training Logistic Regression	29.38	53.70	104.14
	Evaluating Logistic Regression	8.13	53.73	0.04
	Training Decision Tree	27.67	53.7	1.29
	Evaluating Decision Tree	nan	nan	0.005
	Training Support Vector Machine	33.12	55.63	10.30
	Evaluating Support Vector Machine	30.17	53.80	4.91
	Training Random Forest	30.89	54.17	9.52
	Testing Random Forest	28.42	56.00	0.35

Resource Usage Training/Evaluation of SYS

Performing the same experiment with system call data was more complicated. Only twelve data samples (log files) could be assessed for anomaly detection and classification due to the sensor's limited memory and storage capacity. Furthermore, training the DL Autoencoder with extracted system call features crashed the Flask application. Table 7.7 and 7.8 present the findings:

Table 7.7: Training & Evaluation of Anomaly Detection SYS

Category	**Task**	**CPU (%)**	**Memory (%)**	**Seconds (s)**
Pre-Processing	Cleaning log files	51.63	45.30	89.27
	Creating Corpus	55.34	48.08	11.02
	Creating Count Vectorizer (Frequency)	52.13	48.21	8.58
	Applying Count Vectorizer (Frequency)	51.41	48.46	8.22
	Creating TF-IDF Vectorizer	51.29	48.11	8.20
	Applying TF-IDF Vectorizer	51.44	48.58	8.20
	Creating Hashing Vectorizer	nan	nan	0.0005
	Applying Hashing Vectorizer	52.7	50.37	9.43
Anomaly Detection ML	Training IF Frequency	53.84	47.76	2.40
	Evaluating IF Frequency	52.43	47.80	0.46
	Training IF TF-IDF	53.66	47.80	2.25
	Evaluating IF TF-IDF	51.65	47.80	0.46
	Training IF Hashing	53.28	48.08	2.80
	Evaluating IF Hashing	53.38	47.90	0.55
	Training One-Class SVM Frequency	54.13	47.8	0.01
	Evaluating One-Class SVM Frequency	nan	nan	0.003
	Training One-Class SVM TF-IDF	47.03	47.80	0.01
	Evaluating One-Class SVM TF-IDF	nan	nan	0.002
	Training One-Class SVM Hashing	45.03	47.90	0.07
	Evaluating One-Class SVM Hashing	nan	nan	0.003
	Training LOF Frequency	48.6	47.8	0.04
	Evaluating LOF Frequency	53.68	47.8	0.03
	Training LOF TF-IDF	72.03	47.8	0.02
	Evaluating LOF TF-IDF	63.73	47.8	0.03
	Training LOF Hashing	55.43	47.93	0.91
	Evaluating LOF Hashing	54.96	47.7	0.05

Table 7.8: Training & Evaluation of Classification Algorithms SYS

Category	Task	CPU (%)	Memory (%)	Seconds (s)
Classification ML	Training Logistic Regression Frequency	47.03	51.40	0.06
	Evaluating Logistic Regression Frequency	nan	nan	0.001
	Training Logistic Regression TF-IDF	24.53	51.40	0.05
	Evaluating Logistic Regression TF-IDF	nan	nan	0.001
	Training Logistic Regression Hashing	33.99	51.41	0.50
	Evaluating Logistic Regression Hashing	17.03	51.40	0.03
	Training Decision Tree Frequency	nan	nan	0.006
	Evaluating Decision Tree Frequency	nan	nan	0.001
	Training Decision Tree TF-IDF	63.73	51.40	0.005
	Evaluating Decision Tree TF-IDF	nan	nan	0.003
	Training Decision Tree Hashing	15.69	51.40	0.07
	Evaluating Decision Tree Hashing	nan	nan	0.002
	Training Support Vector Machine Frequency	nan	nan	0.006
	Evaluating Support Vector Machine Frequency	57.03	51.40	0.002
	Training Support Vector Machine TF-IDF	57.03	51.40	0.007
	Evaluating Support Vector Machine TF-IDF	nan	nan	0.002
	Training Support Vector Machine Hashing	25.03	51.40	0.06
	Evaluating Support Vector Machine Hashing	nan	nan	0.002
	Training Random Forest Frequency	26.13	51.40	0.94
	Evaluating Random Forest Frequency	28.88	51.40	0.10
	Training Random Forest TF-IDF	26.89	51.40	1.05
	Evaluating Random Forest TF-IDF	29.36	51.40	0.11
	Training Random Forest Hashing	27.21	51.40	1.10
	Evaluating Random Forest Hashing	27.53	51.40	0.12

Time Assessment of ML/DL per Sample

This subsection illustrates the time-related performance metrics per sample for each monitoring script (RES, KERN, and SYS). The experiment was carried out on the Lenovo Laptop to show the real-time performance of live/online evaluation. It considers the time required by the framework to:

1. Pre-process a single data sample.
2. Evaluate the sample with the best anomaly detection algorithm.
3. Evaluate the sample with the best classification algorithm.

Table 7.9 shows the findings. The pre-processing column refers to pre-processing the data for anomaly detection. The pre-processing for classification is omitted, as it is almost the same procedure and requires around the same amount of time. Furthermore, the pre-processing for the SYS monitor is the time needed to clean the data, create the corpus + apply the appropriate vectorizer.

Table 7.9: Time evaluation of a single data sample

Monitor	Pre-Processing (s)	Best Anomaly Algorithm	Evaluation Time (s)	Best Classification Algorithm	Evaluation Time (s)
RES	0.015	LOF	0.240	Decision Tree Classifier	0.0003
KERN	0.011	LOF	0.009	Decision Tree Classifier	0.0002
SYS Hashing	0.531	IF	0.032	Decision Tree Classifier	0.0001
SYS Frequency	0.524	Autoencoder with IQR	0.098	Decision Tree Classifier	0.0002
SYS Tf-idf	0.519	Autoencoder with IQR	0.092	Decision Tree Classifier	0.0003

7.3.4 Recommendations

The experiments demonstrated that training and evaluation are possible on the sensor for monitoring data KERN and RES. However, it is advised that the training/evaluation be performed on the server. This ensures ”normal” sensor operation and decreases the evaluation and training time.

It is not recommended to train ML/DL algorithms with extracted system-call data (SYS Monitor) on the sensor. As only storage capacity is limited and a large amount of memory is required, the training/evaluation should only take place on a high-powered machine.

Online evaluation appears to be a viable method for live anomaly detection and classification because it does not require a significant amount of time to pre-process the data and evaluate the ML/DL algorithms.

7.4 Comparison of Ransomware Detection Approaches

This work has focused on ML and DL for detecting and classifying ransomware. In addition to this approach, human-defined policies can also be used. One of the objectives of this thesis was to compare the detection performance and resource utilization of both detection techniques and identify the strength and weaknesses of each. The policy-based approach introduced in the Related Work chapter [17] only used Monitor RES for its evaluation. Therefore, only the metrics for the RES Monitor can be used for comparison. Furthermore, three different policies were created by Huertas et al.:

- Policy 1: For detecting the normal/abnormal sensor behavior
- Policy 2: For identifying a Ransomware-PoC infection
- Policy 3: For identifying a DarkRadiation infection

The monitoring RES script includes the pre-defined policies for ransomware detection. Creating these policies, however, requires an upfront investment to identify the various levels between the normal and ransomware infection behaviors. A key advantage of this approach is that it requires fewer system resources than ML/DL algorithms, which need training.

The policies correctly identified the anomalous behavior of Ransomware-PoC with a 94.91% TPR and DarkRadiation with 89.80% TPR. For classification, a 93.22% TPR (recall) was achieved in terms of classifying Ransomware-PoC and a 55.10% TPR for classifying DarkRadiation.

In terms of the overall anomaly detection accuracy, all ML/DL algorithms from the thesis experiments outperformed the policy approach (see 7.1).

$$95.88\% \leq \text{TPR DarkRadiation} \leq 100\%$$

$$96.94\% \leq \text{TPR Ransomware-PoC} \leq 100\%$$

Regarding classification, a similar observation can be made when looking at the recall values of the Decision Tree Classifier in table 7.3.

As noted above, the ML/DL results might have been impacted by the shorter duration of training data captured (15 hours for anomaly detection and 4 hours/class for classification).

Both solutions are feasible regarding anomaly detection and ransomware classification. For the given ransomware samples, Huertas policies allow for a higher level of control and tuning. However, ML/DL training is less labor intensive and can be easily applied to sensors with differing hardware configurations. Therefore, it is felt that better results could be obtained by combining both detection and classification strategies.

Chapter 8

Summary, Conclusions and Future Work

This chapter summarizes the main achievements of the thesis and outlines its key findings. A section covering possible areas for future research is also included.

8.0.1 Summary and Conclusion

In summary, this thesis designed and developed a distributed anomaly detection and classification framework. Different behavioral data sources were monitored and ML/DL algorithms were selected to analyse and compare their performance in detecting and classifying ransomware. A variety of experiments related to anomaly detection, classification, and resource utilization were conducted and analyzed to demonstrate the framework's capabilities and limitations. These results were then compared to a policy-based approach. In conclusion, the following are the thesis's most significant findings:

- ML and DL can assist in ransomware detection and classification.
- Anomaly detection algorithms yielded the best results (over all ransomware samples), when trained with RES and KERN monitoring data.
- The LOF algorithm was identified to be the most promising regarding anomaly detection when using RES and KERN monitoring data. It can, however, also be very CPU demanding during evaluation and training.
- DL outperformed ML in anomaly detection, when trained with extracted system call features (Tf-idf).
- SYS, KERN and RES monitoring data is suitable for ML classification.
- The decision tree classifier yielded the best overall classification results (performance and training time) for RES, KERN, and SYS.

- System call monitoring produces a large volume of data. Gigabytes of data can be accumulated over a short period of time, which can quickly overwhelm available storage capacities. In addition, processing of this data is very time and memory intensive. The question then arises, given the less then optimal anomaly detection results observed, is this a viable option for ransomware detection? I believe not.
- The implemented ML/DL algorithms outperformed the policy-based approach regarding anomaly detection and classification.
- Live evaluation of the sensor's health seems to be a viable and fast technique for identifying zero-day attacks and classifying different ransomware strains.
- A larger training data sample for classification and anomaly detection could improve performance of the algorithms.
- The network connection appeared to have a negative impact on monitoring behavior.

8.0.2 Future Work

As noted in the thesis, there are several areas where further investigation could prove beneficial in improving ransomware detection and classification. These include assessing the impact of network activity on the monitoring results and isolating those factors which negatively influence the process. Another future area of research could be identifying the optimal balance between training data size/monitoring duration and ML/DL ransomware detection accuracy.

Finally, testing the framework's detection and classification performance with various types of malware and devices could be an exciting research topic.

Bibliography

[1] Yahye Abukar Ahmed, Shamsul Huda, Bander Ali Saleh Al-rimy, Nouf Alharbi, Faisal Saeed, Fuad A Ghaleb, and Ismail Mohamed Ali. A weighted minimum redundancy maximum relevance technique for ransomware early detection in industrial iot. *Sustainability*, 14(3):1231, 2022.

[2] Manaar Alam, Sayan Sinha, Sarani Bhattacharya, Swastika Dutta, Debdeep Mukhopadhyay, and Anupam Chattopadhyay. Rapper: Ransomware prevention via performance counters. *arXiv preprint arXiv:2004.01712*, 2020.

[3] Ahmad O Almashhadani, Mustafa Kaiiali, Sakir Sezer, and Philip O'Kane. A multi-classifier network-based crypto ransomware detection system: A case study of locky ransomware. *Ieee Access*, 7:47053–47067, 2019.

[4] Iman Almomani, Raneeem Qaddoura, Maria Habib, Samah Alsoghyer, Alaa Al Khayer, Ibrahim Aljarah, and Hossam Faris. Android ransomware detection based on a hybrid evolutionary approach in the context of highly imbalanced data. *IEEE Access*, 9:57674–57691, 2021.

[5] May Almousa, Sai Basavaraju, and Mohd Anwar. Api-based ransomware detection using machine learning-based threat detection models. In *2021 18th International Conference on Privacy, Security and Trust (PST)*, pages 1–7. IEEE, 2021.

[6] Ömer Aslan Aslan and Refik Samet. A comprehensive review on malware detection approaches. *IEEE Access*, 8:6249–6271, 2020.

[7] Amin Azmoodeh, Ali Dehghantanha, Mauro Conti, and Kim-Kwang Raymond Choo. Detecting crypto-ransomware in iot networks based on energy consumption footprint. *Journal of Ambient Intelligence and Humanized Computing*, 9(4):1141–1152, 2018.

[8] Seong Il Bae, Gyu Bin Lee, and Eul Gyu Im. Ransomware detection using machine learning algorithms. *Concurrency and Computation: Practice and Experience*, 32(18):e5422, 2020.

[9] Pranshu Bajpai and Richard Enbody. Memory forensics against ransomware. In *2020 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, pages 1–8. IEEE, 2020.

[10] Pranshu Bajpai, Aditya K Sood, and Richard Enbody. A key-management-based taxonomy for ransomware. In *2018 APWG Symposium on Electronic Crime Research (eCrime)*, pages 1–12. IEEE, 2018.

[11] Jim Bates. Trojan horse: Aids information introductory diskette version 2.0. *Virus Bulletin*, 6:1143–1148, 1990.

[12] Eduardo Berrueta, Daniel Morato, Eduardo Magaña, and Mikel Izal. Crypto-ransomware detection using machine learning models in file-sharing network scenario with encrypted traffic. *arXiv preprint arXiv:2202.07583*, 2022.

[13] Bruhadeshwar Bezawada, Maalvika Bachani, Jordan Peterson, Hossein Shirazi, Indrakshi Ray, and Indrajit Ray. Behavioral fingerprinting of iot devices. In *Proceedings of the 2018 workshop on attacks and solutions in hardware security*, pages 41–50, 2018.

[14] Bruhadeshwar Bezawada, Maalvika Bachani, Jordan Peterson, Hossein Shirazi, Indrakshi Ray, and Indrajit Ray. Iotsense: Behavioral fingerprinting of iot devices, 2018.

[15] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. Lof: identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 93–104, 2000.

[16] Andrea Capponi, Claudio Fiandrino, Burak Kantarci, Luca Foschini, Dzmitry Kliazovich, and Pascal Bouvry. A survey on mobile crowdsensing systems: Challenges, solutions, and opportunities. *IEEE communications surveys & tutorials*, 21(3):2419–2465, 2019.

[17] Alberto Huertas Celdrán, Pedro M Sánchez Sánchez, Eder J Scheid, Timucin Besken, Geŕôme Bovet, Gregorio Martínez Pérez, and Burkhard Stiller. Policy-based and behavioral framework to detect ransomware affecting resource-constrained sensors. pages 1–7, 2022.

[18] Zhaomin Chen, Chai Kiat Yeo, Bu Sung Lee, and Chiew Tong Lau. Autoencoder-based network anomaly detection. In *2018 Wireless Telecommunications Symposium (WTS)*, pages 1–5. IEEE, 2018.

[19] Clarence Chio and David Freeman. *Machine learning and security: Protecting systems with data and algorithms.* " O'Reilly Media, Inc.", 2018.

[20] Anusha Damodaran, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H Austin, and Mark Stamp. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, 13(1):1–12, 2017.

[21] Brittany Day. Anatomy of a Linux Ransomware Attack. `URL:https://linuxsecurity.com/features/anatomy-of-a-linux-ransomware-attack`, 2021. Last accessed on 2022-08-08.

[22] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM computing surveys (CSUR)*, 44(2):1–42, 2008.

[23] Issam El Naqa and Martin J Murphy. What is machine learning? In *machine learning in radiation oncology*, pages 3–11. Springer, 2015.

[24] ElectroSense: Collaborative Spectrum Monitoring. URL:https://electrosense.org/. Last accessed on 2022-08-11.

[25] ElectroSense Github. URL:https://github.com/electrosense/es-sensor. Last accessed on 2022-08-11.

[26] Farnood Faghihi and Mohammad Zulkernine. Ransomcare: Data-centric detection and mitigation against smartphone crypto-ransomware. *Computer Networks*, 191:108011, 2021.

[27] Damien Warren Fernando, Nikos Komninos, and Thomas Chen. A study on the evolution of ransomware detection using machine learning and deep learning techniques. *IoT*, 1(2):551–604, 2020.

[28] Alexandre Gazet. Comparative analysis of various ransomware virii. *Journal in computer virology*, 6(1):77–90, 2010.

[29] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL:http://www.deeplearningbook.org.

[30] HashingVectorizer. URL:https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.HashingVectorizer.html. Last accessed on 2022-08-11.

[31] Nwokedi Idika and Aditya P Mathur. A survey of malware detection techniques. *Purdue University*, 48(2):32–46, 2007.

[32] Syed Ibrahim Imtiaz, Saif ur Rehman, Abdul Rehman Javed, Zunera Jalil, Xuan Liu, and Waleed S Alnumay. Deepamd: Detection and identification of android malware using high-efficient deep artificial neural network. *Future Generation computer systems*, 115:844–856, 2021.

[33] IoT and Ransomware: A Recipe for Disruption. URL:https://www.trendmicro.com/vinfo/us/security/news/internet-of-things/iot-and-ransomware-a-recipe-for-disruption, 2018. Last accessed on 2022-08-11.

[34] M Asha Jerlin and C Jayakumar. A dynamic malware analysis for windows platforma survey. *Indian Journal of Science and Technology*, 8(27):1, 2015.

[35] VVRPV Jyothsna, Rama Prasad, and K Munivara Prasad. A review of anomaly based intrusion detection systems. *International Journal of Computer Applications*, 28(7):26–35, 2011.

[36] Machine Learning for Malware Detection. URL:https://media.kaspersky.com/en/enterprise-security/Kaspersky-Lab-Whitepaper-Machine-Learning.pdf, 2021. Last accessed on 2022-08-11.

[37] Gary McGraw and Greg Morrisett. Attacking malicious code: A report to the infosec research council. *IEEE Software*, 17(5):33–41, 2000.

[38] Per Håkon Meland, Yara Fareed Fahmy Bayoumy, and Guttorm Sindre. The ransomware-as-a-service economy within the darknet. *Computers & Security*, 92:101762, 2020.

[39] Vinit B Mohata, Dhananjay M Dakhane, and Ravindra L Pardhi. Mobile malware detection techniques. *Int J Comput Sci Eng Technol (IJCSET)*, 4(04):2229–3345, 2013.

[40] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018.

[41] Christoph Neumann, Olivier Heen, and Stéphane Onno. An empirical study of passive 802.11 device fingerprinting. In *2012 32nd International Conference on Distributed Computing Systems Workshops*, pages 593–602. IEEE, 2012.

[42] Novelty and Outlier Detection. `URL:https://scikit-learn.org/stable/modules/outlier_detection.html`. Last accessed on 2022-08-11.

[43] Collela Paolo. Ushering in a better connected future. `URL:https://www.ericsson.com/en/about-us/company-facts/ericsson-worldwide/india/authored-articles/ushering-in-a-better-connected-future`. Last accessed on 2022-08-11.

[44] Subash Poudyal and Dipankar Dasgupta. Analysis of crypto-ransomware using ml-based multi-level profiling. *IEEE Access*, 9:122532–122547, 2021.

[45] RAASNet. `URL:https://github.com/leonv024/RAASNet`. Last accessed on 2022-08-11.

[46] Sreeraj Rajendran, Roberto Calvo-Palomino, Markus Fuchs, Bertold Van den Bergh, Héctor Cordobés, Domenico Giustiniano, Sofie Pollin, and Vincent Lenders. Electrosense: Open and big spectrum data. *IEEE Communications Magazine*, 56(1):210–217, 2017.

[47] Gowtham Ramesh and Anjali Menen. Automated dynamic approach for detecting ransomware using finite-state machine. *Decision Support Systems*, 138:113400, 2020.

[48] Ransomware-PoC. `URL:https://github.com/jimmy-ly00/Ransomware-PoC`. Last accessed on 2022-08-11.

[49] Ransomware Spotlight: Hive. `URL:https://www.trendmicro.com/vinfo/us/security/news/ransomware-spotlight/ransomware-spotlight-hive`. Last accessed on 2022-08-11.

[50] Ransomware Spotlight: LockBit. `URL:https://www.trendmicro.com/vinfo/us/security/news/ransomware-spotlight/ransomware-spotlight-lockbit`. Last accessed on 2022-08-11.

[51] Matilda Rhode, Pete Burnap, and Kevin Jones. Early-stage malware prediction using recurrent neural networks. *computers & security*, 77:578–594, 2018.

[52] Y Robiah, S Siti Rahayu, M Mohd Zaki, S Shahrin, MA Faizal, and R Marliza. A new generic taxonomy on hybrid malware detection technique. *arXiv preprint arXiv:0909.4860*, 2009.

[53] Miss Harshada U Salvi and Mr Ravindra V Kerkar. Ransomware: A cyber extortion. *Asian Journal For Convergence In Technology (AJCT) ISSN-2350-1146*, 2, 2016.

[54] Pedro Miguel Sánchez Sánchez, Jose María Jorquera Valero, Alberto Huertas Celdrán, Gérôme Bovet, Manuel Gil Pérez, and Gregorio Martínez Pérez. A survey on device behavior fingerprinting: Data sources, techniques, application scenarios, and datasets. *IEEE Communications Surveys & Tutorials*, 2021.

[55] Anuj Sehgal, Vladislav Perelman, Siarhei Kuryla, and Jurgen Schonwalder. Management of resource constrained devices in the internet of things. *IEEE Communications Magazine*, 50(12):144–149, 2012.

[56] Shweta Sharma, C Rama Krishna, and Rakesh Kumar. Ransomdroid: Forensic analysis and detection of android ransomware using unsupervised machine learning technique. *Forensic Science International: Digital Investigation*, 37:301168, 2021.

[57] William Stallings and Lawrie Brown. *Computer security: principles and practice*, volume 3. Pearson, 2014.

[58] Supervised Learning. `URL:https://www.ibm.com/cloud/learn/supervised-learning`, 2020. Last accessed on 2022-08-11.

[59] Fei Tang, Boyang Ma, Jinku Li, Fengwei Zhang, Jipeng Su, and Jianfeng Ma. Ransomspector: An introspection-based approach to detect crypto ransomware. *Computers & Security*, 97:101997, 2020.

[60] Adam Thierer and Andrea Castillo. Projecting the growth and economic impact of the internet of things. *George Mason University, Mercatus Center, June*, 15, 2015.

[61] Threat research report: The anatomy of a ransomware attack. `URL:https://www.exabeam.com/wp-content/uploads/2017/07/Exabeam_Ransomware_Threat_Report_Final.pdf`, 2016. Last accessed on 2022-08-08.

[62] Unsupervised Learning. `URL:https://www.ibm.com/cloud/learn/unsupervised-learning`, 2020. Last accessed on 2022-08-11.

[63] P Vinod, R Jaipur, V Laxmi, and M Gaur. Survey on malware detection methods. In *Proceedings of the 3rd Hackers' Workshop on computer and internet security (IITKHACK'09)*, pages 74–79, 2009.

[64] What is IoT? `URL:https://www.oracle.com/internet-of-things/what-is-iot/`. Last accessed on 2022-08-11.

[65] What Is Malware? `URL:https://www.cisco.com/c/en/us/products/security/advanced-malware-protection/what-is-malware.html`, Last accessed on 2022-08-08.

[66] Ibrar Yaqoob, Ejaz Ahmed, Muhammad Habib ur Rehman, Abdelmuttlib Ibrahim Abdalla Ahmed, Mohammed Ali Al-garadi, Muhammad Imran, and Mohsen Guizani. The rise of ransomware and emerging security challenges in the internet of things. *Computer Networks*, 129:444–458, 2017.

[67] Yanfang Ye, Tao Li, Qingshan Jiang, Zhixue Han, and Li Wan. Intelligent file scoring system for malware detection from the gray list. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1385–1394, 2009.

[68] Aliakbar Zahravi. Bash Ransomware DarkRadiation Targets Red Hat- and Debian-based Linux Distributions. `URL:https://www.trendmicro.com/en_us/research/21/f/bash-ransomware-darkradiation-targets-red-hat--and-debian-based-linux-distributions.html`. Last accessed on 2022-08-11.

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

Abbreviations

AIDS	Acquired Immunodeficiency Syndrome
APK	Android Package
API	Application Programming Interface
BN	Bayes Network
C2	Command-And-Control
CPU	Central Processing Unit
CMS	Content Management System
DL	Deep Learning
DLLs	Dynamic Link Libraries
DNN	Deep Neural Network
DT	Decision Tree
DTW	Dynamic Time Warping
FFT	Fast Fourier Transformation
FN	False Negative
FP	False Positive
FPR	False Positive Rate
FSM	Finite State Machine
GMM	Gaussian Mixture Model
GUI	Graphical User Interface
HPC	Hardware Performance Counters
IF	Isolation Forest
IoT	Internet of Things
KERN	Kernel Monitor
kNN	K-Nearest Neighbors
LOF	Local Outlier Factor
LSTM	Long-Short-Term-Memory
ML	Machine Learning
mRmR	Maximum Relevance — Minimum Redundancy
MVC	Model-View-Controller
NB	Naive Bayes
NLP	Natural Language Processing
NN	Neural Network
PSO	Particle Swarm Optimization
RaaS	Ransomware-as-a-Service
RES	Resource Consumption Monitor
RF	Random Forest

RNN	Recurrent Neural Networks
SGD	Stochastic Gradient Descent
SSD	Solid State Drive
SVM	Support Vector Machine
SYS	System Call Monitor
TEs	Three Ensembles
Tf-id	Term frequency-inverse document frequency
TN	True Negative
TNR	True Negative Rate
VMI	Virtual Machine Introspection
WEmRmR	Weighted Enhanced Maximum Relevance and Minimum Redundancy

Glossary

Zero-Day Attack Zero-day malware attacks are caused by malware that exploits software security weaknesses yet to be fixed/patched. It can also refer to an attack that deploys novel, unseen malware.

List of Figures

List of Tables

Appendix A

Installation Guidelines

This chapter provides the installation and deployment instructions for the Monitor Controller, the Flask Web framework, and the three ransomware samples. Before proceeding with these instructions, it is assumed that the user has deployed a Raspberry Pi ElectroSense sensor. Furthermore, running a Linux-based distribution on the main computer/server (i.e., Ubuntu) is recommended. Finally, the user should define a separate folder on the server/desktop to store training/evaluation data.

A.1 Monitor Controller

A.1.1 Installing the Monitor Controller on the Raspberry Pi

Prior to installing the Monitor Controller on the Raspberry Pi, SSH needs to be enabled on the server/desktop. The following code presents the installation on a server/desktop machine running Ubuntu:

```
sudo apt-get install openssh-server
sudo systemctl enable ssh
sudo systemctl start ssh
```

After SSH has been enabled, the following commands need to be run on the Raspberry Pi sensor:

```
# Update the packages on the sensor:
apt-get update

# Install git on the sensor:
apt-get install git

# Clone the repository:
git clone https://github.com/dennisshushack/BA_Thesis_PI.git
```

Installing the Monitor Controller is straightforward, as an installer script is provided which downloads all dependencies and creates a passwordless SSH connection between the Raspberry Pi sensor and the server. As shown below, the script takes the username and server/desktop IP address as a command argument.

The following commands illustrate the procedure:

```
# Change Directory into the Git Repo:
cd BA_Thesis_PI

# Give access to the installer script:
chmod +x install_source.sh

# Run installer script (User will be prompted to enter his password)
./install_source.sh -s username@serveripaddress
```

Note use the 'ipconfig' command to find the ip-address of the server if it is not known.

A.1.2 Collecting Data, Sending Data and Live Monitoring

Prior to starting the Monitor Controller, the following commands need to be run on the Raspberry Pi sensor:

```
# Change Directory to Monitor Controller:
cd BA_Thesis_PI/middleware

# Enable Virtual Environment:
source env/bin/activate
```

The Monitor Controller middleware has four commands available to control the functionality of the program. These include, Collect, Show, Send, and Live. These will be used as a command argument when starting the Monitor Controller on the Raspberry Pi sensor.

Collect Command

The Collect command, as the name suggests, is for collecting training or evaluation data. The monitored data is sent every 10 seconds to the server automatically. To start the data collection, the following command has to be run:

```
# Start collecting:
python3 cli.py collect
```

The user is then prompted to enter some additional information to start the collection process:

- **Description**: A short description of the monitoring session i.e. normal data collection
- **Ransomware Type**: Either normal (Normal Behavior), poc (Ransomware-Poc), dark (DarkRadiation) or raas (RAASNet) i.e. normal
- **Training or Evaluation**: Is it training or evaluation data? i.e. training
- **Time**: The time to monitor in seconds i.e. 3600
- **Server Path**: Path on the server/desktop, where the data will be stored i.e. username@serverip:/home/username/Desktop/data

  Note: Reuse the same path for all monitoring sessions!
- **ML/DL Type** Classification or anomaly detection i.e. anomaly
- **Monitors**: Which monitors to run in parallel (If more than one separate by commas) i.e. RES,KERN,SYS

Show Command

The show command displays all past monitoring sessions (from the collect command) in a table. It can easily be executed by running:

```
python3 cli.py show
```

The following shows a sample output of the command. The # column is the index of the monitoring session.

```
root@sensor(rw):~/DA_Thesis_PI/middleware# python3 cli.py show
Table for the device with the following cpu-id: 000000005426d851
```

#	Description	Task	Category	Type	Monitors	Seconds	Done	Sent	Added	Completed

A.1.3 Send Command

The send command sends the metadata (from the collect command) to the server to initiate the **training/evaluation** procedure. Before executing this command, it is necessary to ensure the Flask application is running on the server (next section).

Note, if the user wishes to train classification algorithms with multiple classes/datasets (i.e., normal, poc, dark, and raas), it is sufficient to send only the metadata of one monitoring session (i.e., normal). After this, the Flask application automatically combines the datasets (normal, raas, poc, and dark). Furthermore, the monitoring configurations for training and evaluation must be the same. I.e., when collecting training data with monitors KERN and RES, the evaluation data should also be collected with KERN and RES. Lastly, the ML/DL algorithms must be trained before starting the evaluation. A

script emulating the send command is also provided in the middleware folder called testrequest.py.

The following code executes the send command on the Monitor Controller:

```
python3 cli.py send
```

The user will then be prompted to add the following information:

- **Flask Application**: Where the Flask application is running i.e. 127.0.0.1:5000
- **Task Index**: Index of the monitoring session (# from the show command) i.e. 1

If there are issues with the Monitor Controller-Flask application communication over the Local Area Network, the following debugging steps may help:

1. Ensure that the firewall is not blocking the requests (port).
2. Check the IP address of the Flask application (it should be the same as the server IP).

A.1.4 Command Live

The live command starts a 60-minute **live evaluation** monitoring session. Before executing this command, the anomaly detection and classification algorithms must be trained. This command combines the aforementioned collect and send command for evaluation on a per sample basis. To start a live monitoring session, the following command must be run:

```
python3 cli.py live
```

The required user inputs are similar to those of the collect and send command.

A.2 Flask Web-framework

The Flask application does the data pre-processing, training, and evaluation of the ML/DL algorithms. This installation instruction assumes that the framework will be deployed on the server/desktop.

A.2.1 Installing the Flask Application

A modern Python version is required to run the application (min. Python 3.6). The following commands install the framework together with its dependencies:

```
# Get Virtual Environment:
apt install python3-venv

# Get Git:
apt-get install git

# Clone Repository:
git clone https://github.com/dennisshushack/BA_Thesis_Flask.git

# Change Directory:
cd BA_Thesis_Flask

# Create Virtual Environment:
python3 -m venv env

# Activate the Environment:
source env/bin/activate

# Install python packages
pip install flask numpy pandas pyod tensorflow
```

A.2.2 Running the Flask Application

The following commands start the Flask Web application:

```
# Activate Virtual Environment:
source env/bin/activate

# Set App Folder Location:
export FLASK_APP=app

# Set development mode:
export FLASK_ENV=development

# Initalize the database (Has to be run only once)
# Deletes all entries in db:
flask init-db

# Start the server
flask run --host=serverip
```

Offline training and evaluation of ML/DL algorithms does not require any user interaction. The Flask application uses an SQLite database for data persistence. It contains the anomaly detection evaluation metrics. These can be accessed via the command line or graphically using a tool such as DB Browser for SQLite.

The following are the tables with the evaluation metrics:

- ML_Training_Anomaly: Contains the ML training evaluation metrics (TNR).
- ML_Testing_Anomaly: Contains the ML evaluation metrics produced from testing data (TPR).
- DL_Training_Anomaly: Contains the DL training evaluation metrics (TNR).
- DL_Testing_Anomaly: Contains the DL evaluation metrics produced from testing data (TPR).

Classification reports are saved as text files in the defined monitoring data directory on the server/desktop.

The metrics from a live monitoring session are graphically displayed through the Flask GUI. It can be accessed through a browser with the url: http://flaskip:port/. It is password protected with the default username: admin and password:admin

A.3 Ransomware Samples

The ransomware samples are not accessible through GitHub and must be provided by the university project sponsors. The following subsections illustrate the deployment procedure.

A.3.1 DarkRadiation

The DarkRadiation ransomware source folder contains three files, namely bash.sh, bot.sh and supermicro.sh.

This ransomware requires a Telegram Bot for communication. To create a Bot, the user can follow the official instructions (https://core.telegram.org/bots) section 6: BotFather. This should generate a unique Telegram Token. The Chat ID can be retrieved with the following tool (https://sean-bradley.medium.com/get-telegram-chat-id-80b575520659)

In the next step, the user will have to replace all instances of <TELEGRAM_TOKEN> and <CHAT_ID> appearing in the ransomware source files with his newly generated values. Furthermore, lines 59 and 53 of supermicro.sh have to modified with the correct path (location where the ransomware will be saved on the sensor).

The DarkRadiation ransomware can be executed using the following command:

```
cd DarkRadiation
./supermicro_cr.sh testpassword
```

A.3.2 RAASNet

RAASNet is a ransomware-as-a-service Python program that allows users to create customized ransomware payloads. This thesis uses a specific payload configuration (see Chapter Scenario).

If needed, a new payload can be created by following the installation instructions here (https://github.com/leonv024/RAASNet)

The RAASNet source folder contains two files, payload.py, and requirements.txt. Before deploying the ransomware, the file payload.py has to be modified (lines 57 & 64) to the proper path for file encryption. The provided configuration would encrypt the /root/home directory and its sub-directories.

Two options for deploying the ransomware on the Raspberry Pi (source or binary) are available. The following commands executed on the Raspberry Pi Illustrate the procedure:

```
# Change Directory into RAASNet folder:
cd RAASNet

# Create Virtual Environment:
python3 -m venv env

# Install dependencies:
pip install -r requirements.txt

# Compile as binary:
pyinstaller payload.py --onefile

# Execute from source:
python3 payload.py

# Execute as binary:
cd dist
./payload
```

A.3.3 Ransomware-PoC

Ransomware-PoC is written in Python and allows users to encrypt a specific directory and its sub-directories. It contains the three source files: discover.py, main.py, and modify.py,

and can be downloaded from (https://github.com/jimmy-ly00/Ransomware-PoC). In addition, a compiled binary is provided, which can directly be executed on the Raspberry Pi Sensor. Assuming the user uses the binary version, the following commands execute the ransomware payload:

```
# Change Directory into RAASNet folder:
cd Ransomware-PoC/binary/dist

# Execute payload (Info: -e = encrypt, -p = path)
# Encrypt the folder /root/home/electrosense
./main -e -p /root/home/electrosense
```

Appendix B

Contents of the CD

The CD contains the following:

1. The Midterm presentation of this thesis.
2. The LaTeXsource code of this thesis and the PDF.
3. Links to the Flask-Application's and Monitor Controller's GitHub repositories.
4. The three Ransomware samples (RAASNet, DarkRadiation, and Ransomware-PoC).
5. The pre-processed data-sets for training/evaluating anomaly detection and classification algorithms.
6. Evaluation Metrics (Classification, Anomaly Detection and Resource Utilization)