Battle for Data Integrity: A Machine Learning Approach to Ransomware Detection

Manou Keizer
manou.keizer@gmail.com

August 18th, 2025, 53 pages

Academic supervisor: Kostas Papagiannopoulos, k.papagiannopoulos@uva.nl
Daily supervisor: Jaap Stelma, Jaap.Stelma@infosupport.com
Host organisation/Research group: Info Support B.V., https://www.infosupport.com/

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

Ransomware is a detrimental threat to cybersecurity. In recent years it has gained the ability to evade traditional detection systems by manipulating user-level signals. This thesis looks at how effective machine learning models are for early and robust ransomware detection, by analyzing low-level memory and storage I/O patterns captured at the hypervisor, as this data source is more resistant to tampering within the guest system.

The research systematically compared three different Machine Learning architectures: non-sequential, recurrent, and attention-based. This comparison used a public dataset and a cross-validation framework to evaluate performance, efficiency, and generalization. The main finding is that the non-sequential model performed better across all evaluation metrics, suggesting that the overall statistical properties of a trace are more important than their temporal order.

The research measured the direct trade-off between detection speed and accuracy. It showed inconsistent generalization against ransomware variants with unfamiliar behaviors. The experiments conducted in this study confirm that hypervisor-level monitoring is a viable approach and that a non-sequential model provides the most practical and effective solution for this specific detection context. The findings highlight the important role of feature representation in choosing models and support a tiered strategy for effective ransomware defense.

Contents

Chapter 1

Introduction

Ransomware attacks are a substantial threat to both private and public organizations. They can disrupt operations and cause significant financial losses. Current methods for detecting ransomware often depend on recognizing known harmful patterns or obvious behaviors. However, attackers frequently change their tactics to avoid being detected. This thesis looks into whether monitoring low-level memory and storage activities at the hypervisor level can help detect ransomware earlier in the attack process. By using data sources that are harder for attackers to mimic or change, the goal is to improve detection speed and reliability, especially against new ransomware variants.

1.1 Motivation

Ransomware is a damaging cybersecurity threat because it can quickly encrypt files. If not detected early, this often leads to permanent data loss [1, 2]. High-profile incidents like WannaCry [3] and NotPetya [4, 5] showcased the weaknesses of current defenses. They showed how fast ransomware can inflict damage once it starts encrypting files [6]. Most detection systems today, like antivirus software and endpoint detection tools, rely heavily on recognizing known harmful patterns or specific behaviors that typically appear after significant harm has been done. Moreover, these systems often depend on user-level signals like system calls or API activity, which attackers have learned to manipulate or hide effectively [7, 8].

To tackle these weaknesses, this thesis studies ransomware detection by observing low-level memory and storage access patterns from the hypervisor. The idea is that these patterns, connected to basic system operations, are harder for ransomware to disguise. Detecting ransomware through these patterns could allow for quick action before extensive damage happens.

1.2 Problem Statement and Research Gap

Most existing ransomware detection methods match known malicious signatures or identify suspicious runtime behavior based on heuristics or predefined rules [1]. Signature-based methods often fail to detect new unknown ransomware variants, while behavior-based methods typically wait for clear malicious actions to be able to detect an attack [2, 9]. Researchers have continuously been exploring machine learning methods using user-level features like API calls and registry entries. Even though these methods show promise, they are still vulnerable to manipulation tactics used by sophisticated ransomware [7].

Another approach that is gaining attention involves hypervisor-level monitoring. This method observes memory and storage activity from an external perspective, without using signals from within the compromised system [10, 11]. While this offers potential benefits, current research has mainly focused on static or simplified models without thoroughly testing methods that consider the sequential and time-dependent nature of ransomware behavior. There is limited exploration of sequential machine learning models, such as recurrent neural networks, applied to hypervisor-level data. Additionally, there has not been much focus on how well these models perform under realistic conditions, like short observation windows or when faced with unknown ransomware variants.

This thesis addresses the following specific research gaps:

- A lack of systematic evaluation of sequential machine learning models using hypervisor-level memory and storage access data for early ransomware detection.

- Limited knowledge about how well these sequential models can detect ransomware variants that differ significantly from those used in training.

By exploring these gaps, this thesis aims to evaluate the practical effectiveness of using sequential modeling and hypervisor-level monitoring as viable methods for timely ransomware detection.

1.3 Research Questions

This thesis aims to answer the following main research question:

- **RQ1:** *How well can machine learning models, trained on hypervisor-level behavioral data, detect known and unknown ransomware with minimal latency?*

To explore this main question, the research focuses on the following three sub-questions:

- **RQ1.1:** *How do non-sequential, recurrent, and attention-based models differ in their trade-offs between predictive performance, computational efficiency, and tuning complexity when detecting ransomware from low-level traces?*
- **RQ1.2:** *How does detection latency affect model accuracy, and what is the shortest observation window needed for reliable ransomware detection?*
- **RQ1.3:** *How well can the models apply their learned behavioral patterns to identify ransomware variants they have not seen during training?*

1.4 Proposed Solution

To overcome the limitations of traditional ransomware detection, this thesis proposes and evaluates a detection framework based on low-level, hypervisor-based system traces by utilizing a method that uses a tamper-resistant data source to clearly separate malicious from benign system behavior.

The framework begins by gathering memory and storage access patterns from the RanSMAP dataset [10]. Common evasion tactics, such as API hooking or kernel-level tampering, are not able to tamper this data as it is collected at the hypervisor level. Data from this source has large volume, which is why it is prepared with a feature engineering pipeline that changes the raw event streams into a uniform, sequential view of system activity.

The final and most important part of the proposed solution is finding the best machine learning architecture for this specific task. The framework evaluates three different types of models: a non-sequential gradient boosting model (XGBoost), a recurrent neural network (LSTM), and an attention-based model (Transformer). This evaluation aims to find which model offers the best balance of predictive accuracy, computational efficiency, and generalization, thereby completing the proposed detection solution.

1.5 Contributions

This thesis contributes to the field of behavior-based ransomware detection in the following ways:

1. It shows that a non-sequential model like XGBoost outperforms sequential models in classifying engineered hypervisor traces, achieving a mean F1-score of 0.9794 compared to the LSTM with 0.9680 and the Transformer with 0.9565. This challenges the common belief that sequence-aware architectures are often the best choice for behavioral data.
2. It delivers a quantitative analysis of key operational trade-offs related to real-world use. This includes a measurement of how detection latency affects model performance and an evaluation of how well each architecture generalizes to unseen ransomware families.
3. It presents the design and implementation of a methodologically robust and reproducible evaluation framework. This framework can serve as a basis for future research.

1.6 Scope and limitations

This research provides an analysis of machine learning models designed for ransomware detection using a specific type of low-level data. To keep the study manageable and methodologically sound, the following

scope and limitations are defined.

Scope

The research is bounded by the following components:

- **Data Source:** The analysis relies exclusively on the RanSMAP (Ransomware Storage and Memory Access Patterns) dataset [10]. This study does not include the collection of new data or the use of other datasets.
- **Feature Set:** The models are trained and tested only on pre-engineered 23-dimensional feature vectors, which represent statistical summaries of memory and storage activity as defined in the original RanSMAP study [10]. The study will not consider alternative feature engineering methods or the direct use of raw hypervisor data.
- **Model Architectures:** The comparison focuses on three specific models: XGBoost, LSTM and Transformer. Other machine learning or deep learning models are theoretically evaluated, but not included in the practical analysis.
- **Research Focus:** The main goal is to assess predictive accuracy and computational performance. The study does not cover model interpretability.

Methodological Limitations

The following limitations are part of the study's design and methodology:

- **Limited Generalization:** The findings are based on a relatively small set of six ransomware families and six benign applications. Therefore, the conclusions regarding model performance and zero-day detection capabilities cannot be generalized to the entire range of real-world malware.
- **Exclusion of Network and Process Data:** The selected data set is limited to memory and storage access patterns. Other indicators, like network C2 communication, process creation events, or API call sequences, are not included and are outside the scope of this thesis.
- **Abstracted Early Detection:** The early detection analysis evaluates performance based on the length of the available data trace, as the dataset does not provide true labels for specific attack stages, such as the exact moment encryption occurs.

Technical and Practical Limitations

The following practical constraints affected the experimental design and execution:

- **Hardware and Model Selection:** The choice of model architectures was limited to those that could be trained without needing special hardware acceleration. As a result, newer architectures that depend on GPUs were not implemented.
- **Offline Analysis:** This study is an offline and academic effort that is based on a static dataset. It does not address real-world challenges in a live setting, such as the performance impacts of hypervisor monitoring, data processing delays or the risk of concept drift as attacker tactics evolve.
- **Bounded Model Optimization:** The models were tuned with a systematic grid search over a set range of hyperparameters. This method aimed to find the best configuration for each architecture, but it did not explore all possible parameter combinations, implying there could still be a better configuration.

1.7 Thesis overview and structure

This thesis is structured into the following chapters:

- **Chapter 2: Background**
  This chapter provides an overview of ransomware, its tactics and its effects. It details how traditional detection methods work and what their limitations are. It also introduces the potential of a hypervisor for security monitoring.
- **Chapter 3: Methodology**
  This chapter details the research framework. It introduces the RanSMAP dataset, outlines the feature engineering process and goes through a model selection process. Finally, it details the cross-validation method that is used in all experiments.

- **Chapter 4: Experimental Setup**
  This chapter details the entire workflow of XGBoost, LSTM and Transformer, including the data transformation, model design and training methods.
- **Chapter 5: Experimental Design**
  This chapter describes the design of the experiments that will be conducted in this study. These experiments consist of a hyperparameter optimization process, a baseline model performance comparison, an early detection analysis, and a zero-day generalization test.
- **Chapter 6: Results and Analysis**
  This chapter presents the findings from the experiments described in Chapter 5. It shows a quantitative comparison of the models across a range of evaluation metrics.
- **Chapter 7: Discussion**
  This chapter interprets the findings from the previous chapter. It combines the results to answer the research questions, review the key trade-offs and implications, reflects on the study's limitations and suggests directions for future research.
- **Chapter 8: Conclusion**
  This chapter provides a summary of the thesis. It restates the main findings, reflects back on the work's contributions and provides a concluding statement.

Chapter 2

Background

This chapter provides the background knowledge that is needed to understand the context of this research. It starts by defining ransomware and showing its impact, highlighting the significance of the problem. Next, it looks at traditional detection methods and their limitations, especially against modern evasion techniques. The chapter then introduces the hypervisor as an alternative point for security monitoring, discussing its potential and challenges. Finally, it reviews current academic and commercial solutions in the field.

2.1 Ransomware

Malware, short for malicious software, refers to "any code added, changed or removed from a software system in order to intentionally cause harm or subvert the intended function of the system" [12]. A paper by Namanya et al. in 2018 [13] classifies malware into categories such as viruses, Trojan horses, and worms. However, the most significant type for this research is ransomware. Ransomware is a kind of malware that encrypts a victim's data and asks for payment to restore it.

The effects of malware attacks can reach far beyond individual businesses. They can spill over into critical infrastructure and public services, affecting national security and public health. These attacks can lead to major financial setbacks. For example, the global shipping company Maersk reported a loss of about $300 million from the NotPetya attacks [4, 5]. They can also disrupt essential services, as seen in the Colonial Pipeline ransomware attack in April 2021, which caused the shutdown of Colonial Pipeline for 5 days [14]. Moreover, the influence of malware attacks can linger long after the initial incident, resulting in significant long-term effects. A notable case is the OPM data breach, which involved sophisticated malware and became one of the largest breaches in U.S. history. It exposed personally identifiable information (PII) for 22.1 million individuals [15].

As malware strategies have varied and grown more complex, the ongoing emergence of new ransomware variants presents a major hurdle for today's defense systems. This evolution challenges traditional detection methods because it highlights the need for techniques that work across different types of ransomware families and behaviors.

2.2 Traditional Detection and Its Evasion

Traditional malware detection techniques can fall into a number of different categories, including signature-based, heuristic-based, and behavior-based methods. Each approach has limitations when it comes to detecting ransomware, especially new variants that aim to evade detection.

Signature-based detection is the most common method used in commercial antivirus tools. It looks for known patterns of malicious code but struggles with new samples or polymorphic malware that change their form [12, 16]. Heuristic methods try to detect suspicious behavior based on rule sets or code analysis, but they often depend on static features. This reliance makes these methods open to various types of false positives and false negatives [13]. Behavior-based detection offers a more dynamic solution by monitoring actions during runtime, such as file encryption or registry changes, but these signals often appear late in the attack timeline, which limits the window for early intervention [9].

Machine learning-based detection has recently gained popularity because it can learn patterns from training data and adjust to new threats [16]. However, many of these models rely on features visible

from inside the guest operating system, making them vulnerable to evasion. Modern ransomware can imitate harmless process behavior, delay execution, or use clever input changes to trick the classifier. Specifically, manipulating system calls and timing has been shown to weaken the effectiveness of these techniques [7, 8, 17]. Fileless ransomware adds another layer of challenge. These versions often use reflective loading and direct syscalls to bypass user-mode hooks and behavioral logging [8, 17]. This forces many traditional methods to depend on indirect indicators of compromise, which may not appear until after the payload has executed.

These limitations stem not from poor model design but from fundamental issues regarding where and how data is collected. Models that observe from within the guest OS are easier to mislead or evade.

2.3 The Hypervisor

Instead of monitoring within the guest operating system, a promising point of visibility is outside it at the hypervisor level. This section will introduce the hypervisor and discuss its role in security monitoring.

Hypervisors manage virtual machines and allow one to observe low-level activity, including memory and I/O operations. This creates an opportunity to capture behaviors that malware finds hard to manipulate or hide from.

Hypervisor-based introspection (HVI) lets security tools inspect memory pages, track I/O operations, and monitor control flow without depending on the compromised guest system [18]. This limits the attacker's ability to alter detection logic. This gives defenders access to raw system activity that is difficult to mimic for attackers.

However, HVI brings along challenges. There is a gap between the low-level events a hypervisor detects, such as a page table change, and the high-level behavior that analysts focus on, such as a file being encrypted [19]. Bridging this gap requires one to accurately reconstruct the context of the operating system, which can be prone to errors and costly.

Some early efforts with systems like Ether used hardware virtualization extensions to study malware behavior in isolation [20], but the performance overhead was significant. More recent designs aim to reduce latency by targeting specific low-level signals, though this often sacrifices interpretability.

While HVI has its downsides, it offers an important defense layer by moving observation beyond the attacker's control. For ransomware detection, it allows for modeling the access patterns of memory and storage subsystems, where encryption workloads often leave recognizable traces.

2.4 Current Solutions

Recent research has looked into malware detection by moving closer to the execution root. An example of this is Pulse [21], which uses a Transformer model trained on dynamically collected assembly instructions through binary instrumentation. It identifies malicious behavior by learning structural patterns in instruction-level traces. This introspection enables detailed analysis and has shown promise for zero-day detection, especially when traditional indicators are absent.

In practice, endpoint detection and response (EDR) tools like CrowdStrike Falcon and SentinelOne are commonly used. These systems monitor process behavior, file access, and network connections in near real time. While they work well in many situations, they run within the guest OS, making them vulnerable to tampering. Some malware families now employ techniques like direct system calls or malicious drivers to bypass common EDR hooks [8, 22].

Other systems, such as HOLMES [23], create causal graphs from system events to spot suspicious flows. This graph-based approach provides better situational awareness and can detect attacks that develop over time. However, it often requires careful tuning and may struggle to catch fast-acting or stealthy ransomware variants.

Bitdefender's Hypervisor Introspection (HVI) platform is one of the few commercial tools that takes a different path by analyzing raw memory from outside the guest OS [24]. By placing detection logic at the hypervisor level, HVI aims to provide better isolation from attacks inside the VM. This concept reflects a wider trend toward tamper-resistant monitoring in virtualized environments.

This approach follows a similar idea but shifts the focus to combining low-level memory and storage access patterns by extracting them externally from the hypervisor. They do not rely on file names, process trees, or system APIs, allowing the behavior to be modeled from a neutral perspective. The method becomes less dependent on predefined indicators, which in theory will make it more resilient to new and unknown variants [10, 18, 20].

Chapter 3

Methodology

This chapter explains the framework set up to study how well machine learning models detect ransomware by analyzing low-level system behavior. It starts by discussing the main strategy of using hypervisor-based monitoring to address the weaknesses of traditional detection methods. Next, the RanSMAP dataset is presented, along with its benefits and challenges. Then, the process of feature engineering will be described, which converts raw trace data into behavioral patterns. Finally, the selection process for models is outlined, making the case for choosing a non-sequential baseline and two different advanced sequential architectures.

3.1 Detection at the Hypervisor Level

Traditional ransomware detection methods often depend on user-space and OS-level features, like system call sequences or API logs [16]. While this can be effective, these methods are situated within the guest operating system, making them vulnerable to the very threats they attempt to monitor. Modern ransomware increasingly uses evasion techniques, such as making direct system calls to avoid high-level APIs [8] or using kernel-mode drivers to disable security tools [22], thus bypassing these in-guest monitors.

To address these challenges, this research shifts the focus from within the guest layer to the hypervisor layer, which operates below the guest operating system. The hypervisor, which manages the virtual machine, can monitor low-level hardware activity, including memory and storage I/O operations, without interference from the guest OS [18]. This type of external monitoring is harder to tamper with, providing a more dependable stream of data even if an attacker takes full control of the operating system [11].

By changing this perspective, the task becomes one of detecting behavioral anomalies at the virtualization layer [11], rather than focusing on OS-specific events. The main hypothesis is that even if ransomware mimics benign applications, it will have a harder time masking its core behavior, such as reading files, encrypting data in memory, and writing the encrypted results back to disk, though truly sophisticated variants may intermix or throttle their encryption action to resemble benign workloads. The following sections describe the methodology used to capture and learn from these low-level behavioral patterns.

3.2 The RanSMAP Dataset

The data used in this thesis comes from RanSMAP (Ransomware Storage and Memory Access Patterns), an open dataset of low-level traces collected from both ransomware and benign applications running in a virtualized environment [10]. The dataset provides raw memory and storage activity logs that are collected externally from the hypervisor layer, making it suitable for the detection approach described in Section 3.1. The traces include six different real-world ransomware samples, including WannaCry [25], Ryuk [26], and Conti [27], combined with six benign applications that include applications specifically selected for having similar behavior to ransomware, such as AESCrypt for file encryption and Zip for file compression [10].

3.2.1 Data Robustness

One main benefit of using hypervisor-level data is its strength against common evasion tactics that target security monitors running within the guest environment, such as:

- **API Call Obfuscation.** Ransomware variants often use direct system calls to avoid high-level API hooks used by Endpoint Detection and Response (EDR) tools [8]. Nevertheless, the underlying behavior, such as file encryption, still leads to disk writes and memory activity that the hypervisor captures.
- **Fileless Execution.** "Fileless" ransomware does not write a harmful binary to the disk. Instead, it runs directly in memory [17]. While this activity may escape detection from many file-based scanners, it creates unique memory access patterns that can be seen from the hypervisor layer.
- **OS Kernel Tampering.** Some advanced ransomware might use malicious kernel-mode drivers to disrupt or blind security tools in the operating system [22]. Since the hypervisor operates separately from the guest OS kernel, it remains unaffected by this tampering, which helps keep the data collection process intact.

3.2.2 Challenges and Trade-offs

Even though hypervisor-level monitoring has a lot of potential, it brings along a set of challenges and trade-offs that can greatly influence the design of a detection system. Addressing these challenges requires careful selection of features and models.

- **Semantic Ambiguity.** One of the biggest challenges is the semantic gap between what the hypervisor can observe and what is meaningful [18]. A hypervisor detects memory page accesses and disk block I/O, but does not have an understanding of processes, files, or user actions [19]. For instance, a series of high-entropy disk writes could be part of a ransomware encryption burst or a benign file compression task. This gap drives the methodology that uses feature engineering to convert raw events into behavioral metrics in Section 3.3 and selects sequential models that have the ability to learn the temporal context between these metrics in Section 3.4.
- **Performance Overhead and Data Volume.** As one would expect, tracking low-level activity with high accuracy consumes resources, as for example, every intercepted memory access can trigger a VM-exit, using CPU cycles and stalling the guest VM. This can lead to significant performance problems if not properly managed [20]. The process also results in a large volume of event data, which needs efficient processing and aggregation. As will be discussed in Section 3.3, this issue is dealt with by grouping raw events into fixed-size feature vectors over a sliding time window.
- **Detection Latency vs. Accuracy.** There is a natural trade-off between detection speed and accuracy [9]. A longer temporal window of trace data can provide more context and enhance classification accuracy. However, this also delays detection, giving attackers more time to encrypt files. This trade-off is one of the most critical themes in this research and its experimental design is described in Chapter 5, where the relationship is measured and a minimal viable detection window is sought.
- **Evasion through Mimicry.** A final concern is that attackers may find ways to evade detection by mimicking the I/O patterns of benign software. Research has shown that adversarial workloads can be designed to exploit storage-level features and bypass learning-based detectors [7].

  This supports the choice of advanced sequential models for this research, as they are better at learning complex and long-range dependencies. This resilience is assessed in the zero-day generalization experiment described in Section 5.5.

  This inherent semantic ambiguity requires models that can understand context from the order of events, not just from their overall characteristics. This motivates the investigation of recurrent and attention-based architectures that can learn temporal relationships, as explained in Section 3.4.

3.3 Feature Engineering

To address the challenges of semantic ambiguity and large data volume in hypervisor-level traces, a feature engineering process is used. This process turns the raw data, containing low-level memory and storage patterns, into a fixed-size format that is suitable for machine learning. The features are calculated as described in the original RanSMAP study to provide a reproducible baseline [10]. For more detail on

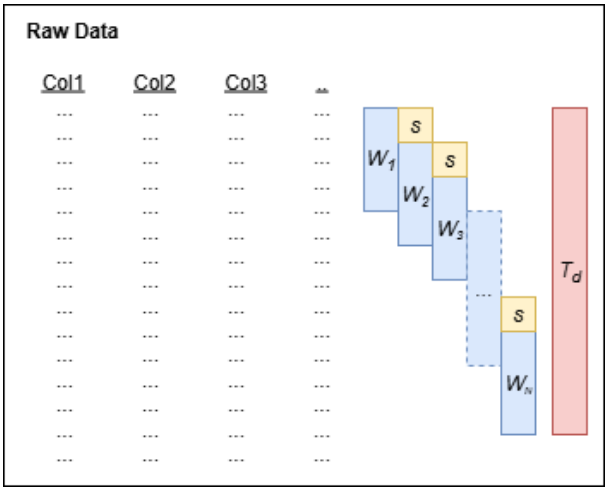


Figure 3.1: The Sliding Window Approach

the exact feature calculation and method, the reader is referred to the original paper.

The main part of this process consists of a sliding window approach, where a time window of size T_d is taken, divided using a fixed-size window size (W) with a step size (s) that was consistently kept at value 1. A visual representation of this process can be seen in Figure 3.1, where Col1 and Col2 are UNIX timestamps in seconds and nanoseconds in the true dataset.

All events in a given window are combined to create a single feature vector. This method changes a continuous, variable-length stream of events into a consistent sequence of behavioral snapshots, which significantly reduces the data volume while maintaining a record of activity over time. While this windowed feature engineering approach gives a reproducible baseline and controls data volume, it assumes that the aggregated features have enough discriminatory power.

The resulting 23-dimensional feature vector consists of 5 storage features and 18 memory features, as described below.

3.3.1 Storage Access Patterns

Storage features come from the disk read and write commands. For each time window, the following five-dimensional vector is calculated, capturing three aspects of storage behavior:

- **Throughput (T_s):** The average read ($T_{s,r}(t)$) and write ($T_{s,w}(t)$) throughput, measured in bytes per second. These reflect the intensity of disk I/O activity.
- **Address Variance (V_s):** The variance of Logical Block Addresses (LBAs) for read ($V_{s,r}(t)$) and write ($V_{s,w}(t)$) operations, calculated as

$$V_s(t) = \frac{1}{N-1}\sum_{i=1}^{N}(LBA_i - \overline{LBA})^2 \tag{3.1}$$

  , where N is the number of accesses between a certain time interval and i the access number. High variance may suggest that disk access is scattered and non-sequential, a common trait of ransomware encrypting different files.
- **Shannon Entropy (H_s):** The average Shannon entropy ($H_{s,w}(t)$) of data from write operations. The entropy is calculated per byte and is normalized to a value between 0 and 1 that has no unit. Encrypted or compressed data shows high entropy, making this an important feature for spotting ransomware activity.

3.3.2 Memory Access Patterns

Memory features are obtained from the operation types read, write, read/write, and execute memory access events. Within each time window, an 18-dimensional vector is created, capturing three aspects of memory behavior:

- **Shannon Entropy (H_m):** The average entropy is calculated for pages accessed during write ($H_{m,w}(t)$) and read/write ($H_{m,rw}(t)$) operations, which can indicate in-memory data obfuscation or encryption. Similar to the storage entropy, this is a normalized, unitless value between 0 and 1.
- **Page Access Counts (C):** The count of memory pages accessed for each operation type and for each page size (4 KiB, 2 MiB, and MMIO), resulting in 12 count-based features. These features measure the quantity and type of memory activity.
- **Address Variance (V_m):** The variance of Guest Physical Addresses (GPAs) is calculated for each of the four operation types. It is calculated similarly to V_s, as can be seen in equation (3.1), but uses GPA instead of LBA. A high GPA variance shows that the process accesses scattered locations in memory.

3.4 Model selection

For this research, determining the most suitable AI model requires comparison between various well-established and state-of-the-art models. Due to the selection of the RanSMAP dataset, containing sequences of low-level storage and memory access feature vectors, there are important attributes that the final selected model is required to have.

3.4.1 Selection method

The following steps were taken in order to filter out models that do not fit the scope of this thesis or the context of the RanSMAP dataset:

1. **Potential Candidates:** A survey of established and state-of-the-art models suitable for classification on system-level data was conducted.
2. **Essential Requirements:** The candidates were evaluated against a set of essential requirements. The models that did not pass these requirements were filtered out.
3. **Model Comparison:** The remaining models were compared based on their strengths and weaknesses.
4. **Final Selection:** A final set of three models was chosen to serve specific roles: a high-performance non-sequential baseline, a classic recurrent sequential model, and a state-of-the-art attention-based model.

3.4.2 Potential candidates

The potential candidates consist of machine learning (ML) algorithms, artificial neural networks, deep learning models, hybrid models, and state-of-the-art models. This selection was based on existing literature and expert opinion.

1. **Traditional Machine Learning Models:**
These algorithms are well-established techniques that are designed to operate on fixed-length static feature vectors instead of sequences. Therefore, if these algorithms are applied to the time-ordered RanSMAP data, a transformation will be required to convert the sequences into a single input vector. This is most commonly done by "flattening" the sequence into one large vector. Note that this necessary adaptation will discard the temporal ordering and sequential dependencies offered by the dataset, which ideally would be preserved. With this understood, the ML models that are considered are:

- *Random Forests (RF):* Random Forests are ensemble classifiers that create multiple decision trees. Each tree is trained on different bootstrap samples and uses a random subset of features for each split [28]. Predictions come from a majority vote among the trees. RF models are robust against overfitting, and they are easy to interpret. In the RanSAP study [29], Random Forests achieved relatively high average F1 scores on storage-based ransomware traces. However, when applied to flattened RanSMAP data, the loss of sequential order might limit their ability to recognize behavior changes over time.
- *k-Nearest Neighbors (kNN):* kNN is a non-parametric, instance-based learning algorithm. It classifies input vectors based on the majority class among their k nearest neighbors, using a distance metric like Euclidean distance. It is simple to implement and can produce good results when relevant patterns are distinct in the feature space. The RanSAP study also reported good performance for kNN [29]. However, the performance of kNN is known to degrade with high-dimensional data, which

is a potential outcome as the sequence will need to be flattened here as well, and its prediction time complexity does not scale well.

- *Support vector machine (SVM):* SVMs create a maximum-margin hyperplane to separate data in high-dimensional feature space, often using kernel functions. They are memory-efficient and can manage both linearly and non-linearly separable data. The RanSAP study observed that SVMs did not perform as well as RF and kNN on their dataset [29]. It's unclear how additional memory features in RanSMAP might change this ranking. When working with flattened sequences, SVMs might benefit from the high-dimensional feature representations.
- *Decision Tree (DT):* Decision Trees make hierarchical splits on the input space based on feature thresholds, creating a tree-like structure of decisions. Their interpretability and fast inference make them appealing in real-time systems. However, DTs are prone to overfit, especially in high-dimensional spaces and when the training data lacks diverse behavior patterns. The study by Wang et al. [30] found that Decision Trees achieved a relatively high F1 score of 91.5% as a first-layer filter, but their performance in zero-day settings wasn't specifically evaluated.
- *Extreme Gradient Boosting (XGBoost):* XGBoost is a scalable and regularized gradient boosting framework that builds an ensemble of trees in a sequential, additive way [31]. It has built-in mechanisms for handling missing values and ranking feature importance. This model showed relatively promising results in the study by Wang et al. [30], where it achieved an F1 score of 96.7%, but had a high computation time of 56 seconds, which they solved by enforcing a two-layered model. This study does make the XGBoost method look promising with its high precision and recall for similar data.

2. **Recurrent Neural Networks (RNNs):**
These deep learning models are designed to process sequential data by incorporating loops in their architecture. This enables them to have an internal state or "memory" that stores information from past time steps, which influences processing of future steps. This property makes RNNs strong candidates to process the time-ordered sequences of feature vectors from the RanSMAP dataset. The DL models that are considered are:

- *Simple Recurrent Neural Networks (RNN):* This model processes sequences one at a time and uses the output of the previous step in combination with the input to compute the current hidden state and output. The downside of a simple RNN is that they are prone to suffer from the vanishing or exploding gradient problem, making it challenging to learn long-range temporal dependencies [32]. Their practical performance is often surpassed by more advanced variants like LSTMs.
- *Long Short-Term Memory (LSTM):* LSTM networks build on RNNs by adding memory cells and gating mechanisms that help manage the flow of information [33]. This structure reduces the vanishing gradient problem and allows for modeling longer-range dependencies. LSTMs are commonly used for sequence classification tasks and were part of the original RanSMAP implementation where they used two layers [10]. However, their large number of parameters and the need for extensive hyperparameter tuning can lengthen training time and add complexity. Greff et al. [34] highlight the challenges of effectively tuning LSTM variants across different tasks.
- *Gated Recurrent Unit (GRU):* This RNN is similar to a LSTM model, but lacks a context vector or output gate, which results in fewer parameters than LSTM. This can lead to GRUs being computationally more efficient and faster to train in certain cases, while often being as effective as LSTMs on sequence modeling tasks. This effectiveness is demonstrated in a paper by Zhao et al., in which they modeled sequential data derived from sensor inputs for system state analysis in the related field of machine health monitoring [35]. Although generally competitive and more efficient than LSTMs, the reduced gating structure may offer a lower modeling capacity for certain complex sequence modeling problems than the more parameterized LSTM framework. However, as with LSTMs, they in general are still considerably more complex and can handle data requirements better than traditional ML models.

3. **Convolutional Neural Networks (CNNs):**
Though most frequently utilized for processing spatial data such as images, the fundamental concepts of CNNs can be applied to sequential data analysis via 1D convolutions. This is relevant to the RanSMAP dataset, as it is a time-ordered sequence of multivariate feature vectors. The models that are considered are:

- *1D-CNN:* This model uses convolutional filters that slide along the time dimension of the input sequence, while processing all the features at each step. They often use pooling layers to reduce

dimensionality afterwards. Its main benefit is its ability to learn local, short-term patterns or motifs that are present in the sequence data [36]. In addition, it is computationally efficient because of parameter sharing and parallel computation. However, its default structure has a limited receptive field, which means that it only sees a small part of the sequence at a time. This makes it challenging for 1D-CNNS alone to grasp relationships between events that do not happen consecutively in the detection window, which may be required to capture complex ransomware behavior.

- *Temporal Convolutional Network (TCN):* TCNs build on CNNs by using dilated convolutions to expand the receptive field exponentially. This makes it possible to model long-range temporal dependencies without recursion [37]. The causal structure ensures that outputs at any time depend only on past inputs. Liu et al. [38] show that TCNs perform better than LSTMs and GRUs on various sequence prediction tasks, often converging faster. However, adjusting dilation rates, kernel sizes, and residual connections may need more expertise than with traditional RNN models.

4. **State-of-the-Art Models:**
Aside from the standard recurrent and convolutional model, recent research from security conferences like USENIX Security explores more sophisticated architectures that are capable of capturing complex relationships and address advancing problems in system behavior analysis. These are the state-of-the-art approaches that are considered:

- *Transformers:* Transformers replace recurrence with multi-head self-attention mechanisms. This change allows the model to relate all positions in the input sequence directly, no matter the distance [39]. Positional encodings are added to keep the sequence order. This architecture can model long-range dependencies and has been successful in areas beyond natural language processing, including cybersecurity. FlowTransformer [40] applies Transformers to network flow sequences for intrusion detection. Pulse [21] uses them for zero-day ransomware detection on runtime instruction sequences. However, the self-attention mechanism that Transformers use has a quadratic memory cost with respect to sequence length ($O(T^2 \cdot d)$) and training Transformers oftentimes requires extensive hyperparameter tuning.
- *Graph Neural Networks (GNNs):*
GNNs work on graph-structured data by sending messages between connected nodes over several layers [41]. They can model complex relationships and have shown success in analyzing system provenance graphs. However, using GNNs for RanSMAP would require a preprocessing step to turn sequences into graphs. This step introduces design decisions that can impact performance. Moreover, training on large or dense graphs can be costly in terms of computation.
- *State Space Models (SSMs):* SSMs model temporal data with a latent state that changes over time through learned update functions. Recent developments like Mamba [42] include high-order polynomial operators in linear state-space models to scale sequence modeling over long time periods. Mamba offers linear scaling ($O(T)$) compared to the quadratic cost of Transformers and shows competitive results on long-sequence benchmarks. However, its recent arrival means that tools and libraries are still being developed, which may pose challenges for practical use in limited environments. Additionally, it requires to be trained on a GPU.

3.4.3 Essential Requirements and Model Filtering

To narrow down the candidates, a set of technical and practical requirements was established. Models that did not meet these criteria were filtered out.

- **Native Data Structure Handling:** The model must be able to process sequences of multivariate vectors. Otherwise, it should provide a well-justified baseline if it is unable. This is the most critical requirement since the objective is to evaluate the usefulness of temporal patterns.
- **Implementation Feasibility:** The model must be trainable with available computational resources, meaning it should not require GPU acceleration or complex data preprocessing steps. This practical constraint is vital to ensure the research can be completed within its time frame.
- **Architectural Maturity:** Architectures that are well-established and stable are preferred. Models with known performance issues, such as Simple RNNs and vanishing gradients, or those that are functionally replaced by better options are deprioritized.

Applying these requirements leads to the following filtering decisions:

- **Filtered Out:** Most traditional ML models (RF, kNN, SVM, DT) are excluded because they do not have the ability to natively handle sequential data. XGBoost is kept as it will function as a

non-sequential baseline, allowing analysis of the added value of a model taking temporal ordering into account. Simple RNNs are filtered out because of their architectural immaturity compared to modern recurrent networks. GNNs and SSMs are filtered for not meeting the feasibility requirement due to complex preprocessing and GPU dependency.

- **Remaining Models:** The models that meet the requirements and move on to the final comparison are XGBoost as a non-sequential baseline, LSTM, GRU, TCN, and the Transformer.

3.4.4 Comparison of Remaining Models

To get a clearer view of the practical trade-offs, the five remaining models are put into a comparison matrix that assesses them across several architectural and computational factors, as shown in Table 3.1. The aim is to understand not just what each model does but also how its design makes it suitable for specific aspects of the detection problem.

Table 3.1: Comparison of Final Model Candidates

Factor	XGBoost	LSTM	GRU	TCN	Transformer
Core Mechanism	Gradient Boosted Trees	Gated Recurrence	Gated Recurrence	Dilated Causal Convolutions	Multi-Head Self-Attention
Temporal Modeling	None (Static features on flattened data)	Inherently Sequential (processes step-by-step)	Inherently Sequential (processes step-by-step)	Hierarchical and Parallel (captures local patterns)	Global and Parallel (relates all steps at once)
Complexity	Low (highly optimized)	$O(T \cdot d^2)$ (Sequential execution is a bottleneck)	$O(T \cdot d^2)$ (Slightly faster than LSTM)	$O(T \cdot d \cdot k)$ (Highly parallelizable)	$O(T^2 \cdot d)$ (Quadratic in sequence length)
Ideal Use-Case	High-performance non-sequential baseline.	Modeling complex sequences where memory is critical.	Efficiently modeling less complex time dependencies.	Capturing local motifs and hierarchical patterns in long sequences.	Detecting complex, long-range dependencies between non-contiguous events.
Key Weaknesses / Trade-Offs	Completely ignores all temporal information by design.	Can struggle with very long-range dependencies; sequential nature is slow.	Slightly less expressive than LSTM for highly complex sequences.	Requires careful tuning of kernel size and dilation factors to be effective.	Quadratic complexity makes it computationally expensive for very long traces.

The comparison shows several key trade-offs. XGBoost gives a strong and efficient baseline, but ignores temporal information. Among the sequential models, a choice needs to be made between recurrent (LSTM, GRU) and parallel (TCN, Transformer) architectures. While GRU and TCN offer high efficiency, LSTM and the Transformer are the most established and well researched, making them good candidates for a thorough comparison of different modeling methods.

3.4.5 Final Selection and Conclusion

Based on the comparison the following three models are selected for implementation and evaluation:

1. **XGBoost:** It serves as the non-sequential benchmark. This choice offers a reference point to measure any performance gain from modeling the temporal order of events.
2. **LSTM:** It represents the recurrent approach to sequence modeling. Even though GRU is more efficient, LSTM's expressive architecture makes it the standard for testing how well a model works with an evolving memory.
3. **Transformer:** It represents the attention-based approach. Although TCNs are also strong, the Transformer's global self-attention mechanism is fundamentally different from recurrence. This makes the comparison between it and LSTM more meaningful. This choice allows for testing if a global-context model performs better than a local-memory model for this task.

This selection allows for a thorough comparison of non-sequential, local-sequential, and global-sequential models. The main hypothesis tested in the following chapters is that the sequential models, especially the Transformer with its global context awareness, will perform better than the non-sequential baseline in detection accuracy. This improvement is expected to be most significant in early detection and zero-day scenarios, where capturing the timeline of an attack is believed to be important.

3.4.6 Discussion

Even though the model selection process is justifiable, it has to be acknowledged that it relies on several key assumptions and trade-offs that need to be discussed. These factors affect how the experimental results in upcoming sections are interpreted.

The first and most important assumption is the reliance on the feature engineering pipeline. The performance of all three selected models, XGBoost, LSTM, and the Transformer, is closely linked to the specific 23-dimensional sequential representation of the data created in Section 3.3. The windowed aggregation of raw hypervisor events represents a type of compression that, while necessary for manageability and useful for comparison with RanSMAP's results [10], loses context within the windows. Therefore, the later experiments will not identify which model is universally the best for ransomware detection. Instead, they will show which architecture is most effective at extracting malicious patterns from this specific representation.

Secondly, the selection process involves a careful trade-off between practicality and the latest advancements. The choice to leave out emerging architectures like State Space Models (SSMs), particularly Mamba, is a notable example. While the Transformer is the established leader in attention-based modeling, its quadratic complexity ($O(T^2 \cdot d)$) regarding sequence length is a known limitation. Architectures like Mamba aim to overcome this issue by achieving linear-time scaling ($O(T)$), making them theoretically better suited for the long behavioral traces often seen in cybersecurity. However, Mamba's reliance on specialized algorithms for modern GPUs and its relatively recent introduction made it impractical for this project. Although it is excluded for practical reasons, SSMs show promise for future work in this area.

Finally, model interpretability continues to be a challenge for all chosen architectures in this context. Tree-based models like XGBoost are often seen as more transparent than deep neural networks, but this benefit is less significant here. When a sequence of, for example, 30 time windows is combined into a single vector of $30 \times 23 = 690$ features, interpreting feature importance or individual decision paths becomes complex. For the LSTM and Transformer models, the challenge is even greater. As a result, while the models may achieve high predictive accuracy, explaining why a specific trace was marked as malicious remains challenging outside the scope of this thesis.

3.5 Evaluation Framework

To ensure a reproducible and fair evaluation of the chosen models, all experiments take place within a Stratified 10-Fold Cross-Validation framework [43]. This method is more reliable than a single train/test split because it reduces the risk of a biased or "lucky" data split. It ensures that the reported performance reflects the model's true ability to generalize.

Additionally, a strict three-way data split is done that prevents data leakage, where a model's performance may be unfairly improved by exposure to test data during training.

3.5.1 Stratified 10-Fold Cross-Validation

The feature set, referred to as $\mathcal{D}$, is randomly shuffled to make sure the order of traces does not affect the results. This shuffling is important to avoid systematic bias, such as similar traces appearing in the same partitions. After shuffling, the dataset is divided into ten separate, stratified folds of equal size, $\{F_1, F_2, \ldots, F_{10}\}$, so that $\mathcal{D} = \bigcup_{i=1}^{10} F_i$. The stratification process ensures that each fold keeps the same percentage of benign and malicious samples as the full dataset. The experiment runs 10 times, using each fold as the test set once, while the other nine folds are used for training. The final performance metrics reported include the average and standard deviation across these 10 runs.

3.5.2 Three-Way Data Split

In each of the 10 folds, a three-way data split is applied. For each iteration $i \in \{1, \ldots, 10\}$, the data is divided as follows:

1. **Test Set ($\mathcal{D}_{test}^{(i)}$)**: The test set is the current fold, accounting for 10% of the total data.

$$\mathcal{D}_{test}^{(i)} = F_i$$

2. **Training Pool ($\mathcal{D}_{train\_pool}^{(i)}$)**: The training pool consists of all other folds, making up the remaining 90% of the total data.

$$\mathcal{D}_{train\_pool}^{(i)} = \mathcal{D} \setminus F_i$$

3. **Training and Validation Sets ($\mathcal{D}_{train}^{(i)}, \mathcal{D}_{val}^{(i)}$)**: The training pool is further divided into the final training and validation sets using an 85/15 split. This results in the validation set being 13.5% of the total data (0.15×0.90) and the training set being 76.5% of the total data (0.85×0.90).

$$(\mathcal{D}_{train}^{(i)}, \mathcal{D}_{val}^{(i)}) = \text{split}(\mathcal{D}_{train\_pool}^{(i)}, \text{ratio} = 0.15)$$

This split makes sure that the final evaluation is always done on data that the model has not encountered during any training or tuning phases.

3.5.3 Data Filtering and Preparation

To develop a stronger detection model that works in various environments, the dataset for all experiments was created by combining traces from three different collections from the RanSMAP dataset [10], consisting of:

- **Original:** This collection contains the traces collected across twelve different baseline hardware configurations.
- **Variants:** This collection contains traces from seven different variants of the ransomware variant Conti.
- **Extra:** This collection contains traces from hardware that is not in the Original set, such as newer CPUs and varying RAM types.

By combining these traces, the models are encouraged to learn the basic, hardware-independent behavior patterns of ransomware, improving their chances for real-world application. There is an additional collection of 'mixed' traces, which include simultaneous benign and malicious activity, that was intentionally excluded from all experiments to avoid contaminating the training data with unclear signals.

Chapter 4

Experimental Setup

This chapter details the workflow architecture and implementation of the chosen models for the upcoming experiments, following the selection process from Section 3.4. Each model works with the same windowed storage and memory features but needs different preprocessing steps and architectural elements to optimize their performance.

For each of the models, it is shown how the data is transformed, how the structure is designed, how training is conducted, and how predictions are generated. These design choices, like flattening input versus keeping the time order, directly impact latency, generalizability, and interpretability, which will be evaluated in Chapter 6.

4.1 Extreme Gradient Boosting (XGBoost)

To establish a strong non-sequential baseline, an Extreme Gradient Boosting (XGBoost) model was implemented. XGBoost is a tree-based ensemble algorithm that performs well on tabular data. It has shown promising results in previous work on similar storage-based ransomware datasets, achieving an F1 score of 0.971 as the second layer in a two-tiered model in the study by Zhongyu Wang et al. [30]. While it does not account for temporal dependencies, it provides a useful comparison for assessing the advantages of sequence-aware models.

This section explains the implementation of the model in this thesis. First, it details how the feature data was reshaped to fit XGBoost's input requirements. Next, it describes the training process, including regularization techniques and early stopping. Finally, it explains how the trained model generates predictions and the classification thresholds applied. The entire workflow of the XGBoost model is illustrated in Figure 4.1, but will be discussed in detail in the upcoming sections where the stages will be referred to as the numbers in the top left of the blocks.

4.1.1 Data Transformation for XGBoost

This model uses the same engineered features as the upcoming models but prepares its data differently. XGBoost requires one-dimensional feature vectors for each input trace. Therefore, each trace is reshaped from a two-dimensional matrix of shape $(n_W, 23)$, which represents n_W time windows and 23 features, into a single vector of size $23 \cdot n_W$ **(1)**. To keep the input length consistent, all traces are padded or truncated to a fixed number of windows before flattening. As a result, short traces will have zero-padded windows, which may lessen the amount of informative signal available to the model.

This transformation is needed for the model to function, but it inherently removes any temporal structure that the model could use. Consequently, the model cannot tell the difference between early and late-stage behavior in a trace. Section 5.4 analyzes how this limitation impacts early detection performance.

After flattening, the model splits the data into a training set, a validation set, and a test set for each fold of the cross-validation **(2)**. This ensures the model is trained, optimized, and evaluated on completely separate data. While this allows for controlled evaluation within the distribution, it does not test the model's ability to generalize to new traces from unseen ransomware families. Section 5.5 defines an experiment that addresses this limitation through zero-day evaluation.

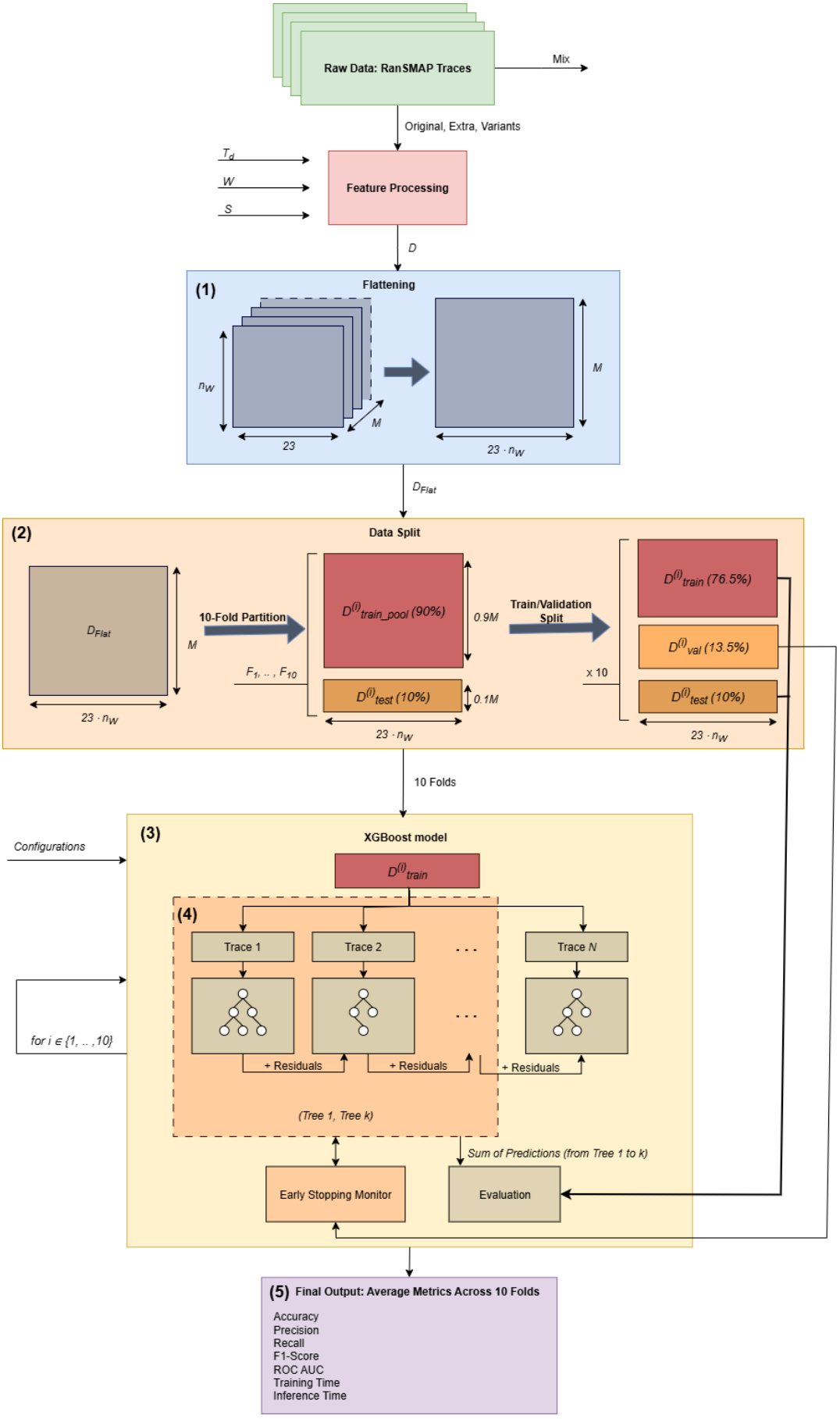


Figure 4.1: Full process for XGBoost

4.1.2 Model Architecture and Training

The XGBoost model was implemented in Python using the existing xgboost library and trained using the xgb.train function. The model employs a gradient boosting algorithm, where decision trees are built sequentially **(3)**. Each new tree is trained to correct the errors, also called residuals, of the previous trees. In order to prevent overfitting, an early stopping mechanism **(4)** was implemented. The model's performance is checked on the separate validation set. Training stops if the log-loss metric does not improve for a specific number of boosting rounds. The stopping point is indicated by a k in figure 4.1. The final evaluation is then carried out over the untouched test set.

This training method is controlled by a set of hyperparameters, which are discussed and tuned further in Chapter 5 and Chapter 6. The relevant XGBoost-specific parameters are listed below. In the implementation, the learning rate is designated as `lr` but is passed to XGBoost internally as the parameter `eta`:

- `num_boost_round`: The maximum number of trees that can be built
- `lr / eta`: The learning rate controls the contribution of each tree to the final ensemble
- `max_depth`: This is the maximum depth a tree can have, controlling its complexity
- `row_sample`: The fraction of training data that is randomly sampled for growing each tree
- `col_sample`: The fraction of features, i.e. columns, to be randomly sampled when constructing each tree
- `reg_alpha`: L1 regularization
- `reg_lambda`: L2 regularization

4.1.3 Inference and Classification

This inference and classification procedure is carried out for each of the 10 folds in cross-validation. For each fold, after training and selecting the best group of k trees using the validation set, the final model is used to predict the test set that was not part of the training. The final prediction score for each trace is calculated by summing the outputs of all k trees in the group. This total is then mapped to a binary classification, with 1 for ransomware and 0 for benign, using a probability threshold of 0.5. In real life deployment, this threshold should be adjusted based on the environment's base rate and action costs in order to control false positives. The performance metrics are then calculated for each fold, and the final results reported in this thesis reflect the average of these 10 outcomes. If applied in real-world situations, lowering this threshold could help decrease false negatives, which is important because ransomware is a low-frequency, high-impact event.

4.2 Long Short-Term Memory (LSTM)

To explore the value of sequence-aware modeling, an LSTM-based neural network was implemented.Unlike XGBoost, which processes a flattened vector of features, the LSTM is a type of recurrent neural network (RNN) built to learn from time-ordered data [33]. This difference in structure lets the model capture temporal dependencies in the low-level traces. It can potentially identify behavioral patterns that change during the life cycle of an attack.

This section outlines the complete LSTM implementation process. It starts with how the input data is prepared and fed into the model. Next, it covers the LSTM's design, which includes its recurrent structure, bidirectional flow, and dropout layers. Finally, it discusses the training setup, including choice of optimizer, early stopping, and gradient clipping. The complete data flow and model design for the LSTM are depicted in Figure 4.2.

4.2.1 Data Transformation for LSTM

Unlike the data preparation for XGBoost, the LSTM model does not need to flatten feature vectors. Each trace retains its 2D shape of (n_W, 23), where n_W is the number of windows and 23 is the number of features per window. When the model processes these traces, they are grouped into batches, forming a 3D input tensor with the shape (batch_size, n_W, 23).

This preservation of the temporal structure is a key benefit of using a recurrent model. The LSTM processes the data window by window in the original order, allowing it to learn patterns based on the sequence of events. For each of the 10 cross-validation folds, the feature set is split into training,

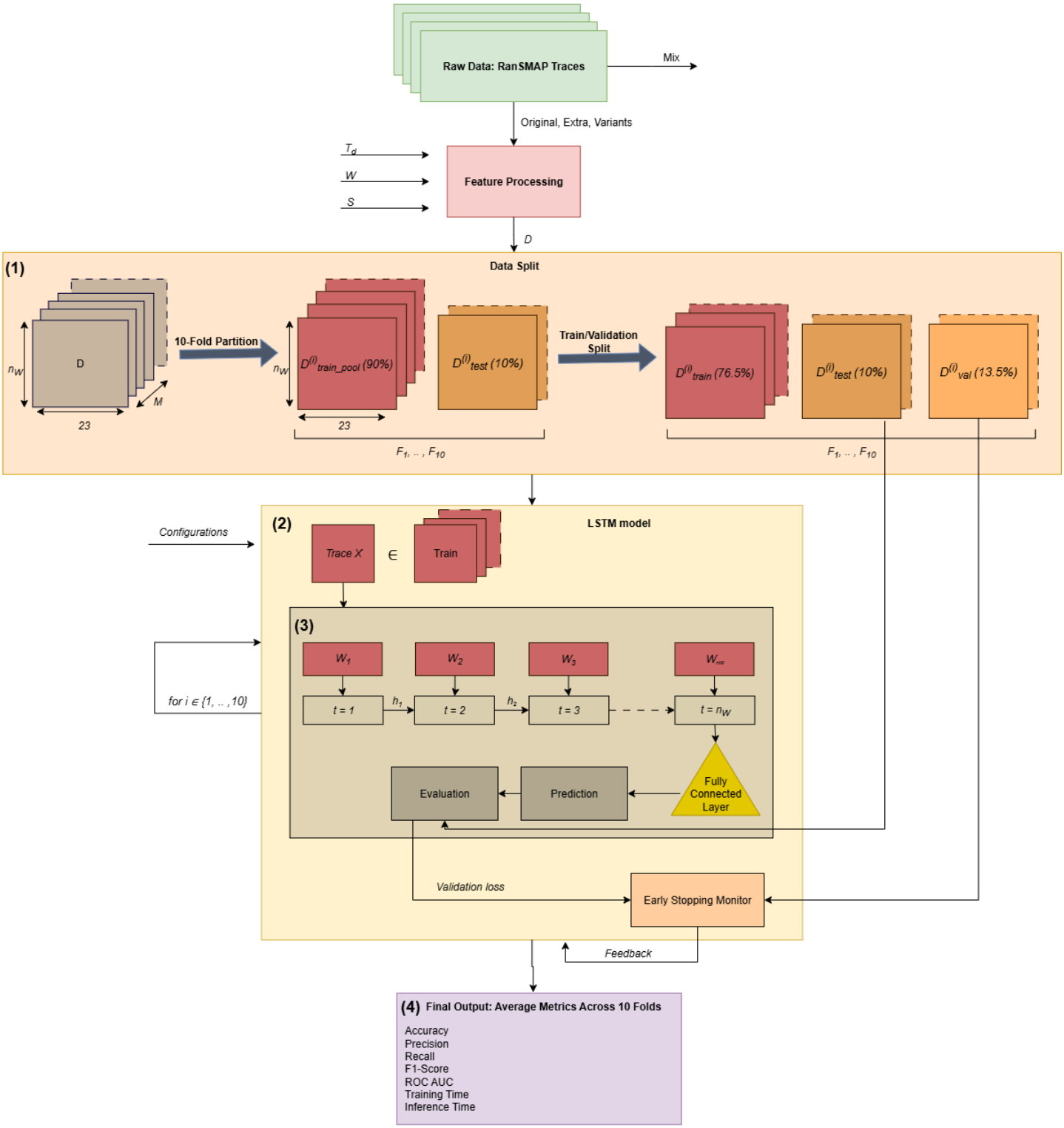


Figure 4.2: Full process for the LSTM model

validation, and test sets **(1)**. Before the data reaches the model, it is scaled using a `StandardScaler` that is fit only on the training data of the current fold, which is then used to transform the training, validation, and test sets.

4.2.2 Model Architecture and Training

The model is a multi-layered, optionally bidirectional LSTM classifier implemented using the PyTorch library **(2)**. The model was designed as sequence-to-one **(3)**, as it is the goal to classify the entire trace based on its behavior, instead of classifying each individual window. Making the model sequence-to-sequence would be an interesting approach for future research, but for this research there will be experimented with shorter detection times by varying T_d.

The LSTM processes the sequence of 23-dimensional feature vectors for each trace one time step at a time. At each step, the LSTM cell updates its internal hidden state, which acts as memory, gathering information from all previous steps. When set as bidirectional, the model processes the sequence both forwards and backwards, allowing its predictions to be informed by both past and future context within the trace. While using future context can improve classification accuracy on a finished trace, it is important to recognize that in a real-world, real-time detection scenario, this method introduces a latency trade-off. The system must wait for the full observation window to be collected before it can make a prediction. This design decision will be based on the results of the hyperparameter tuning optimization in Section 5.2.

The final output from the LSTM layer at the last time step, which represents a summary of the entire sequence, is passed through a dropout layer for regularization before being fed into a fully connected linear layer. This final layer maps the learned features to a two-dimensional output that provides the raw scores for the benign and ransomware classes.

The model is trained using the Adam optimizer and the CrossEntropyLoss function, which is suitable for binary classification problems. To prevent overfitting, an early stopping mechanism is employed. The model's performance is monitored on the validation set after each epoch, and if the validation loss does not improve for a specified number of consecutive epochs (patience), training is halted and the model state with the best validation performance is saved for final evaluation.

The training process is guided by a set of hyperparameters, which are established through an optimization process that is described in Section 5.2. The key hyperparameters for the LSTM model are:

- `hidden_dim`: The number of features in the LSTM's hidden state.
- `num_layers`: The number of stacked LSTM layers.
- `dropout`: The probability of dropping neurons to prevent overfitting.
- `bidirectional`: Indicates if the LSTM processes data in both directions.
- `learning_rate`: The step size used by the Adam optimizer.
- `weight_decay`: L2 regularization term for the optimizer.
- `epochs`: The maximum number of training epochs.
- `batch_size`: The number of traces processed in each training iteration.
- `patience`: The number of epochs to wait for improvement before early stopping.
- `grad_clip`: A maximum value to clip gradients to prevent exploding gradients.

4.2.3 Inference and Classification

This classification procedure takes place for each of the 10 folds in the cross-validation. After identifying the best-performing model with the validation set, it is used once to generate predictions on the untouched test set for that fold. The model processes the entire sequence and the output from the last time step is transformed by the fully connected layer. These resulting scores are then passed through a softmax function to turn them into probabilities for each class. The class with the higher probability is selected as the final prediction, similar to applying a 0.5 probability threshold. Similar to XGBoost, this threshold should be based on the environment's base rate and action costs in deployment. Performance metrics are computed for each fold, and the final results reported in this thesis reflect the average of these 10 outcomes **(4)**.

4.3 Transformer

A Transformer model [39] was implemented to serve as a state-of-the-art sequence model. While it was originally developed for natural language processing, they have also been proven effective for system behavior analysis, as discussed in Section 3.4.

The self-attention mechanism is a main architectural feature for the Transformer. The LSTM processes data one step at a time, but the self-attention mechanism in the Transformer looks at all elements in a trace at the same time. This design is in theory powerful for this dataset, as it can identify connections between different attack stages across a long trace, like linking early memory operations to later mass disk encryption. The complete data flow and model design for the Transformer are depicted in Figure 4.3.

4.3.1 Data Transformation for Transformer

Similar to the LSTM, the Transformer model is able to process sequential data. Each trace maintains its 2D shape of (n_W, 23). When processed, these traces are collected into batches, creating a 3D input tensor with the shape (batch_size, n_W, 23). For each of the 10 cross-validation folds the feature sets are split into training, validation, and test sets. Before the data enters the model, it is scaled using a `StandardScaler` that is fit only on the training data of the current fold.

However, unlike a Recurrent Neural Network, the Transformer does not have an inherent understanding of sequence order. To solve this, Positional Encoding is added to the input features. This step adds information about the position of each time window by incorporating a unique, mathematically derived vector into each feature vector in the sequence. This enables the model to distinguish between an event occurring at the start versus the end of a trace. This method follows the original Transformer paper [39].

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}}),$$

where pos is the position of the time window, i the dimension index and d_{model} the dimensionality of the model's embedding. This embedding practice follows that of the original Transformer paper [39].

4.3.2 Model Architecture and Training

The model is an encoder-only Transformer, implemented in Python using the PyTorch library. The model consists of the following components **(2)**:

1. An initial linear layer embeds the 23-dimensional feature vector for each time step into a higher-dimensional space (d_{model}).
2. The Positional Encoding is added to this embedded representation.
3. This sequence is processed by a stack of transformer encoder layers. Each layer has a multi-head self-attention mechanism followed by a feed-forward neural network. The self-attention mechanism allows the model, at each time step, to evaluate the significance of all other time steps in the trace and build a context-aware representation.
4. Global average pooling is applied to the output of the final encoder layer to produce a single vector for the entire trace. This means averaging the feature vectors for all time steps to create one fixed-size representation that summarizes the entire trace.
5. This pooled representation is sent to a final fully connected layer for classification into benign or ransomware classes.

The training process is identical to that of the LSTM, utilizing the Adam optimizer and CrossEntropyLoss function for binary classification. An early stopping mechanism attempts to mitigate overfitting. The model's performance is tracked on a separate validation set after each epoch. If the validation loss does not improve, training stops, and the model state with the best validation performance is saved for final evaluation.

The training process is guided by a set of hyperparameters, which are established through the optimization process described in Section 5.2. The key hyperparameters for the Transformer model are:

- `d_model:` The dimensionality of the model's internal representations.
- `nhead:` The number of heads in the multi-head self-attention mechanism.
- `num_encoder_layers:` The number of stacked Transformer encoder layers.

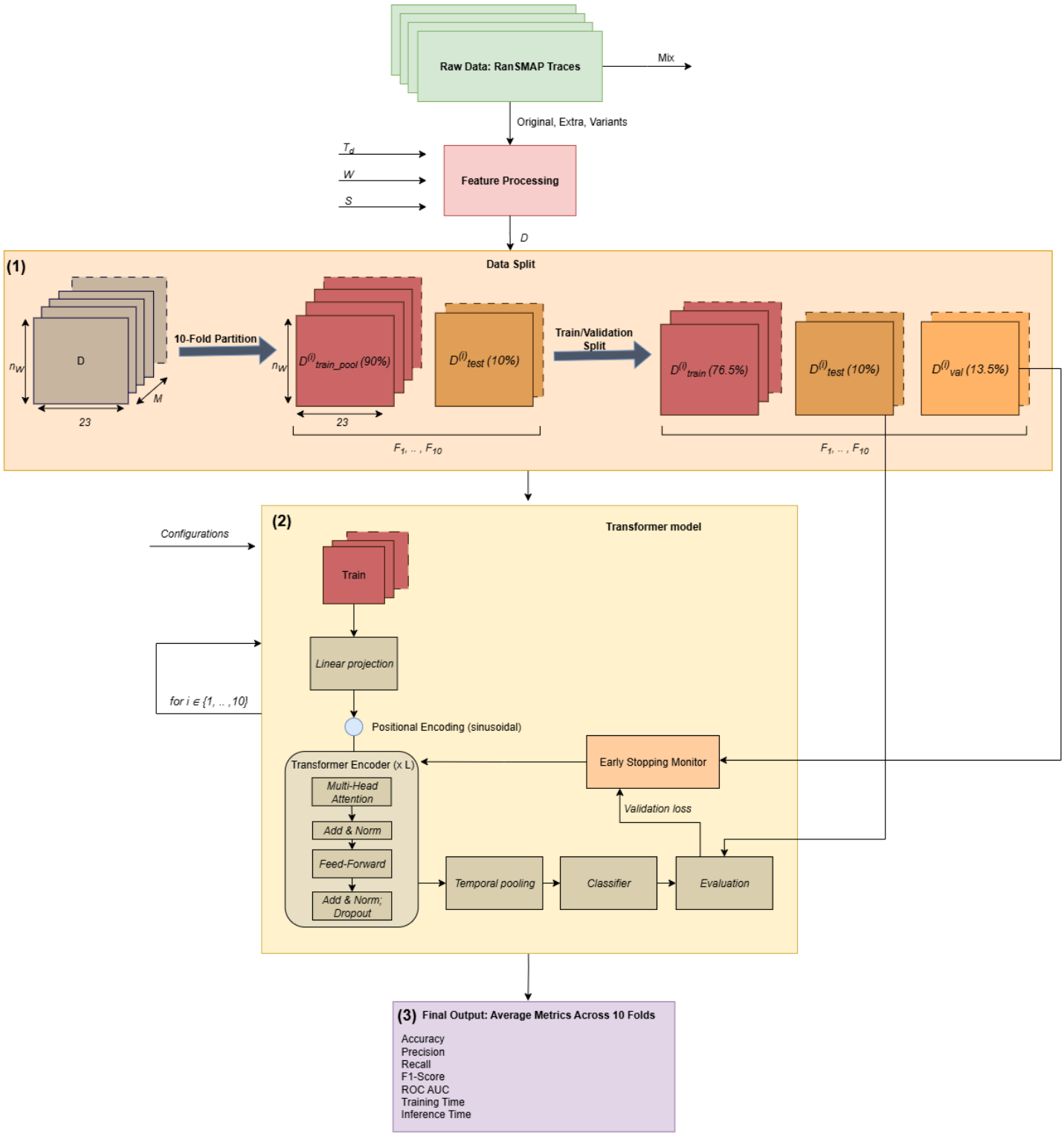


Figure 4.3: Full process for the Transformer model

- `dim_feedforward`: The dimension of the feed-forward network inside each encoder.
- `dropout`: The dropout rate used for regularization.
- `learning_rate`: The step size used by the Adam optimizer.
- `batch_size`: The number of traces processed in each training iteration.
- `epochs`: The maximum number of training epochs.
- `patience`: The number of epochs to wait for improvement before early stopping.

4.3.3 Inference and Classification

The classification procedure occurs for each of the 10 folds in the cross-validation. After finding the best-performing model using the validation set, it is used once to generate predictions on the untouched test set for that fold. During prediction, the self-attention mechanism computes a representation for each time step. The resulting sequence of vectors is averaged to create a single vector that captures the overall behavior of the trace. This vector is classified by the final linear layer, and a softmax function converts the output scores into probabilities to determine the final prediction. Performance metrics are computed for each fold

Chapter 5

Experimental Design

This chapter describes the design and steps taken in the series of experiments aimed at answering the research questions. It starts by defining the set of evaluation metrics that will be used consistently across all experiments to make a fair comparison. Then, the chapter explains the specific procedures of each experiment, including a hyperparameter optimization sweep, a baseline performance comparison, an early detection analyses and zero-day analyses.

5.1 Evaluation Metrics

To ensure an objective comparison of the models, a consistent set of metrics is used across all experiments. These metrics are selected to assess both the predictive accuracy and the computational efficiency of each model, directly addressing the main research questions.

- **F1-Score:** This is the main metric for evaluating predictive performance. The F1-score represents the harmonic mean of precision and recall.
- **Accuracy:** This measures the proportion of all predictions that were correct.
- **Precision:** This indicates the proportion of positive predictions that were correct, calculated as
$$\frac{\text{true positives}}{\text{true positives} + \text{false positives}}.$$
- **Recall:** This shows the proportion of actual positive cases that were correctly identified, calculated as
$$\frac{\text{true positives}}{\text{true positives} + \text{false negatives}}.$$
- **ROC AUC:** This stands for the Area Under the Receiver Operating Characteristic Curve, measuring the model's ability to distinguish between classes.
- **Training Time:** This is the total time in seconds needed to train a model.
- **Inference Time:** This indicates the time in seconds required for a trained model to make a prediction on a single trace.

5.2 Hyperparameter Optimization

The primary objective of this experiment is to address research question 1.1. To determine the performance of a model, the focus is not only on predictive accuracy but also a model's characteristics, including its tuning complexity, sensitivity to configuration changes, and its trade-off between performance and computational cost.

To ensure a fair comparison, each model must be evaluated at its best potential. Therefore, the first step of the experimental workflow is a systematic hyperparameter optimization sweep. This process identifies the most effective configuration for each model before the final evaluations.

5.2.1 Experimental Procedure

The hyperparameter sweep is an automated process that uses a grid search method, where a predefined set of hyperparameters for each model is systematically explored. In order to improve efficiency, this

sweep is conducted on the training pool from the first and second fold of the 10-fold cross-validation. Note that this can introduce mild optimism if those folds are not fully representative, which is acknowledged as a potential limitation. The data is then split into a training set and a validation set.

For each combination of hyperparameters, a model is trained on the training set, and its performance is assessed against the validation set. The F1-score was chosen to be the primary metric for this experiment, as it offers a balanced measure of precision and recall. The secondary metric was Inference Time, as this metric reflects the speed of detection. The hyperparameters that produce the highest F1-score on the validation set, and the lowest inference time if multiple configurations offer the highest F1 score, are automatically identified and saved as the best configuration for this model. These configurations are used as input for all subsequent experiments.

5.2.2 Feature Set Selection for Optimization

In order to make this tuning process computationally feasible and consistent, the tuning process for all 3 models was conducted on a single feature set. The set that was chosen has trace duration (T_d) 30, window size (W) 5 and step size (s) 1 in seconds. This representation was chosen because it represents a balanced trade-off between detection latency and providing enough contextual information to perform well. While feature sets with longer trace durations often yield higher accuracy, the 30-second detection window is more practical for real-world defense scenarios. The experiment found in Section 5.4 will focus on the optimization and comparison of varying this detection window.

5.2.3 Search Space

A grid search approach was used to search through the hyperparameter space for every model. The parameters that were selected and the ranges of their values were picked to span the most impactful dimensions of each model's architecture and training process.

XGBoost

For the tuning the XGBoost model, the focus is on the trade-off between the number of trees, their complexity and the learning rate. To maintain a focused and computationally feasible search space, two parameters were held at fixed values, namely `reg_alpha`=0, and `reg_lambda`=1. The sweep then focused on the combination of the following hyperparameters:

- `n_boost_round` $\in [200, 500]$**:** This tests a moderate versus larger amount of ensemble trees, influencing complexity and training time.
- `lr` $\in [0.05, 0.1, 0.3]$**:** This tests the rates that are most commonly used. A learning rate of 0.3 is often default, the 0.1 is more robust and the 0.05 may allow for finer and more stable learning.
- `max_depth` $\in [3, 5, 7]$**:** These values control the complexity of each individual tree. The value 3 represents a shallow tree, often simple and fast, up until 7, which is a deeper tree that is better at handling intricate interactions but prone to overfitting. The 5 was included to provide a middle ground.
- `row_sample` $\in [0.8, 1.0]$**:** These values represent the proportion of training data used for every tree to add stochasticity and improve generalization. The 1.0 uses all training data for each tree and 0.8 builds each tree on a random 80% sample.
- `col_sample` $\in [0.8, 1.0]$:

LSTM

For the LSTM model, the focus is on finding balance between complexity, learning dynamics and regularization. To maintain a focused search, a selection of training parameters were held at fixed values, including `grad_clip`=1.0, `scheduler_step_size`=15, `epochs`=50 and `patience`=10. The sweep then focused on the combination of the following hyperparameters:

- `hidden_dim` $\in$[64,128,256]: This parameter controls the capacity of the model. The sweep tests a small (64), medium (128), and large (256) hidden state size to find the optimal complexity for our data.
- `num_layers` $\in$[1,2]: This explores the trade-off between a simpler network (1 layer) and a deeper network (2 layers).

- `lr` ∈[0.001,0.0005]: These values represent common and effective learning rates for training LSTMs with the Adam optimizer, testing a standard rate versus a slightly more conservative one.
- `dropout` ∈[0.1,0.3]: This tests a lower and a medium level of dropout, which is an important regularization technique that prevents overfitting by randomly zeroing connections between LSTM layers.
- `weight_decay` ∈[0.0,0.0001]: This tests the model's performance with and without L2 regularization, which is a method for penalizing large weights in order to improve generalization.
- `bidirectional` ∈[True,False]: This evaluates whether the model should process the sequence only forwards or in both directions to incorporate future context.
- `batch_size` ∈[32,64]: This shows the impact of using a smaller versus a larger batch size, affecting the stability of the gradient updates and the training time per epoch.

Transformer

For the tuning of the Transformer model, the focus is on the core components of the self-attention mechanism and the model's overall capacity. As this model is known for being sensitive to tuning, only the number of epochs and patience were fixed to the same values as the LSTM (50 and 10), resulting in 256 different hyperparameter combinations. These combinations consist of the following hyperparameters:

- `d_model` $\in [64, 128]$: This parameter controls the model's capacity. The 64 represents a standard value, while the larger 128 checks if a larger capacity is needed for this task.
- `nhead` $\in [4, 8]$: This sets the number of parallel attention heads. The variable `d_model` is split across these heads, which means `d_model` needs to be divisible by `nhead`, which is how the values 4 and 8 were selected.
- `num_encoder_layers` $\in [2, 3]$: This determines the model's depth. A one-layer Transformer is often too shallow, which is why it was not included, 2 layers is a typical minimum for capturing complex relationships. Additionally, 3 layers are added to see if increasing depth gives a performance boost without overfitting.
- `dim_feedforward` $\in [128, 256]$:This dictates the size of the hidden layer of each encoder's feed-forward network. Common practice is for this to be 2-4x the d_model, which is why 128 and 256 are selected.
- `lr` $\in [0.0001, 0.00005]$:The value 0.0001 represents a commonly used and robust learning rate, while the 0.00005 explores a more conservative choice often used for fine-tuning.
- `dropout` $\in [0.1, 0.2]$:The value 0.1 is a default value used in many seminal Transformer papers. The 0.2 is included to test if a slightly more aggressive regularization is needed to prevent the model from overfitting on this specific dataset.
- `weight_decay` $\in [0.01, 0.0001]$:This tests the model sensitivity to L2 regularization. 0.01 is a fairly strong penalty, whereas 0.0001 is a much milder one.
- `batch_size` $\in [16, 32]$:This checks the balance between training stability and memory consumption. A batch size of 32 is a commonly used default value, which gives stable gradient estimates, whereas 16 can provide better generalization at the expense of noisier gradient updates.

5.3 Model Performance Comparison

After tuning the hyperparameters, this experiment establishes the baseline performance of the non-sequential XGBoost, the recurrent LSTM, and the attention-based Transformer. The goal is to compare their effectiveness and efficiency directly under the evaluation framework described in Section 3.5.

5.3.1 Experimental Procedure

The experiment uses the best hyperparameter settings for each model that will be identified by the results of the experiment described in Section 5.2, which are in Section 6.1. The baseline evaluation was conducted using Stratified 10-Fold Cross-Validation. The train, validation and test split follows the protocol described in Section 3.5.

5.4 Early Detection Analysis

This experiment focuses on the aspect of RQ1 that focuses on how well a model can detect threats quickly. It tests the performance of each model with incomplete data traces, simulating a real-world scenario where detection speed is critical. The main aim is to find the shortest observation window that allows for reliable detection and to measure the trade-off between detection speed and accuracy.

5.4.1 Experimental Procedure

To ensure the models are compared fairly, this experiment uses the best-performing hyperparameter setup for each of the three models, demonstrated in Appendix A. By keeping the model hyperparameters constant, any changes in performance can be linked directly to the varying lengths of the input sequence.

Similar to previous experiments, this follows the 10-fold cross-validation framework to maintain consistency. The initial 10-fold split is set once using the full 30-second dataset, resulting in 10 fixed sets of training and testing indices. For each of the 26 trace durations, the corresponding feature set is loaded, and these same indices are used to divide it into the training pool (90%) and the test set (10%). The training pool is then further divided to create the final training and validation sets. This method ensures that for any given fold, the same traces make up the test set across all evaluated time windows. This setup allows for a direct and unbiased comparison of performance based on trace duration.

The experiment consists of two distinct parts, each aimed at answering a different question about early detection capabilities.

Experiment A: Learning from Limited Data

The first part of this experiment looks at learning capability. The goal is to find the least amount of information each architecture needs during training to create an effective model. To do this, the models were retrained and tested for each of the 26 different feature sets. This method directly measures each model's ability to learn effective patterns when the training data is sparse.

Experiment B: Testing with Limited Data

This experiment shows a more realistic deployment scenario where a single, fully-trained model must predict using incomplete information. For the sequential LSTM and Transformer models, a model is trained once on the complete 30-second feature set. This model is then tested with progressively shorter test sets, starting from 30 seconds and going down to 6 seconds. This setup evaluates how well the sequential models manage incomplete data during inference.

Since the non-sequential XGBoost model needs a fixed-size input vector, this method does not apply to it, as the remaining windows would need to be padded, and is therefore excluded from this experiment.

5.5 Zero-Day Generalization

The experiment uses the best-performing hyperparameter setup for each model, demonstrated in Appendix A. A Leave-One-Family-Out cross-validation strategy is applied. For each of the six ransomware families in the dataset, a complete evaluation is done by excluding that family from the training process and using it as the malicious component of the test set.

To ensure the results are statistically reliable, the process follows the 10-Fold framework. In each of the 10 runs, a different reproducible random seed is used to split the benign traces into a training pool (80%) and a test set (20%). Although Recall is the primary metric for this experiment, unseen benign traces are included in the test set. This step is crucial to ensure the experiment's validity. It compels the model to learn a meaningful decision boundary and stops it from getting a high score by simply labeling all inputs as malicious.

The final datasets for each run are set up as follows:

- **Test Set**: Includes all samples from the single held-out ransomware family and the 20% of unseen benign traces.
- **Training Pool**: Contains all samples from the other five ransomware families and the 80% of benign traces. This pool is then divided into the final training and validation sets.

Chapter 6

Results and Analysis

This chapter demonstrates and analyzes the results from the experiments outlined in Chapter 5. The goal is to examine the findings to give data-driven answers to the research questions of this thesis. The chapter is organized to follow the experimental process. It starts with analyzing the hyperparameter optimization sweep to find the best configuration for each model. Next, it presents and analyzes the results from the baseline performance comparison, the early detection analysis, and the zero-day generalization experiments.

6.1 Hyperparameter Optimization Results

This section presents the results of the hyperparameter optimization experiment outlined in Section 5.2. To address the research question, "What kind of AI model is best suited for accurately detecting ransomware attacks in real-time?", it is insufficient to only focus on the model's final performance score. A model's suitability is also determined by its sensitivity to hyperparameters, the complexity of its tuning process, and the trade-offs it presents between performance and computational cost. This section analyzes the results of the hyperparameter sweep to understand these characteristics for the XGBoost, LSTM, and Transformer models. By observing the correlations between each model's hyperparameters and its resulting performance metrics, insight into the models behavior is gained allowing for a more informed decision regarding which architecture is "best suited" for ransomware detection.

To interpret the correlation heatmaps consistently, a standard way to categorize the strength of the Pearson correlation coefficient (r) [44] is used:

- **Weak correlation:** $|r| < 0.3$
- **Moderate correlation:** $0.3 \leq |r| < 0.7$
- **Strong correlation:** $|r| \geq 0.7$

The following subsections look at the results for each model using this framework.

6.1.1 XGBoost

The analysis of the hyperparameter sweep for XGBoost reveals several characteristics about its performance and efficiency. The correlation heatmap in Figure 6.1 shows a strong negative correlation of -0.73 between the `learning_rate` and the `Avg Training Time`, along with a moderate negative correlation of -0.64 to the `Avg Inference Time`. This indicates that as the learning rate decreases, the computational cost tends to increase.

Another notable result is that the direct correlations between the tested hyperparameters and the final `Avg F1-Score` are all weak. The `max_depth` parameter, for example, has a weak negative correlation of -0.17. This suggests that deeper trees did not consistently produce higher F1-scores within the tested range.

The scatter plot in Figure 6.2 demonstrates the relationship between the F1-score and the inference time. Across all hyperparameter combinations, the average inference time stays in a very narrow and low range from 0.0004 to 0.0012 seconds. The F1-score is also consistently high, ranging from 0.958 to 0.981. The plot shows that for XGBoost, different hyperparameter combinations do not lead to a significant trade-off between performance and speed. When taking a closer look, the scatterplot shows a pattern of horizontal clusters of data points. These clusters indicate that various hyperparameter setups

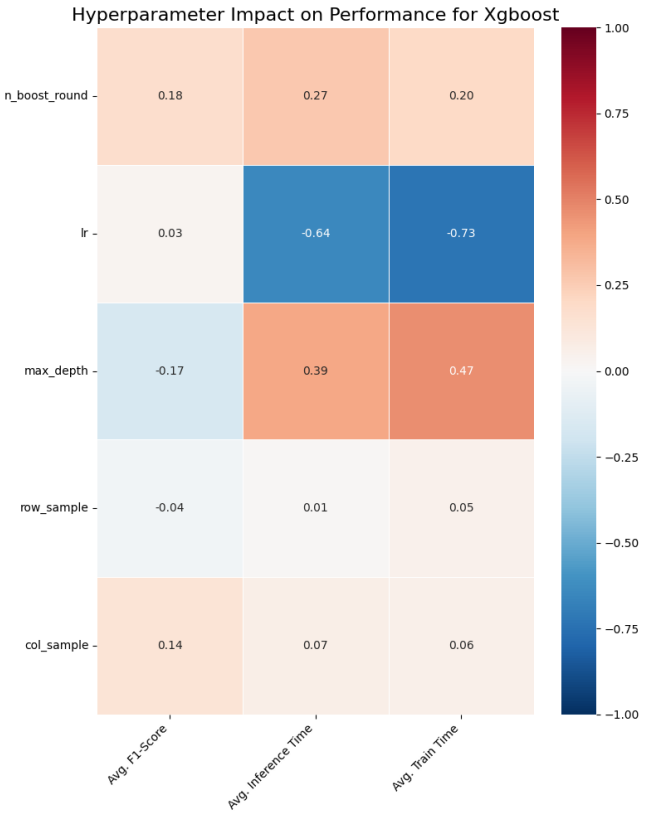


Figure 6.1: XGBoost heatmap showing parameter impact

can reach near-identical performance levels. For example, examining the top-performing cluster **(1)**, it is found that all hyperparameter combinations are evenly distributed, for example two data points have a `max_depth` of 3 and the other two of 7. This trend shows that for the XGBoost model with this dataset, there is no single configuration that stands out. The same was found for cluster **(2)**, where there was not one hyperparameter value that was constantly present in the configurations. **This shows that for XGBoost there is a broad range of different high-performing setups, highlighting the model's strength and versatility.**

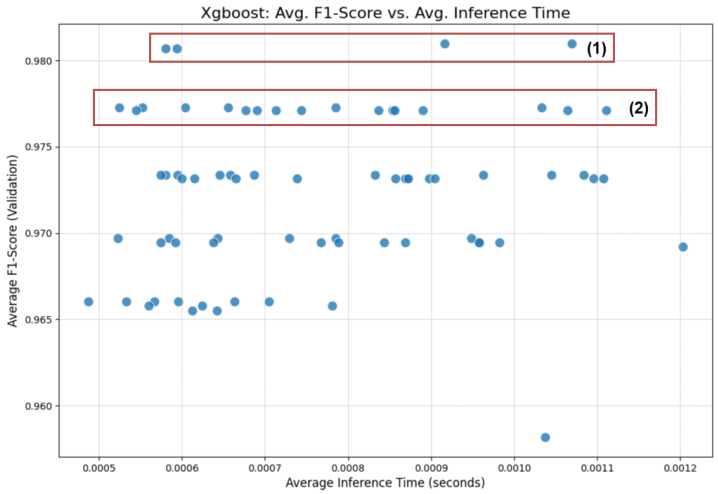


Figure 6.2: XGBoost: Performance vs. Efficiency Across Hyperparameter Configurations

6.1.2 LSTM

The analysis of the hyperparameter sweep for the LSTM model shows that this architecture is more sensitive to its configuration than XGBoost. The correlation heatmap in Figure 6.3 indicates that the relationships between hyperparameters and performance are more complex.

The most influential parameter on the `Avg F1-Score` is the learning rate `lr`. It has a moderate positive correlation of 0.56, which means the higher rate of 0.001 was generally more effective in the tested range. The model's capacity, represented by `hidden_dim`, also shows a moderate positive correlation with the F1-score, at 0.33. Regarding computational cost, there is a strong positive correlation between training and inference time, at 0.94. The parameters that most affect this cost are `hidden_dim`, with correlations of 0.62 for train time and 0.59 for inference time, and `num_layers` with a correlation of 0.43 for inference time. This confirms that larger and deeper models are more costly to compute.

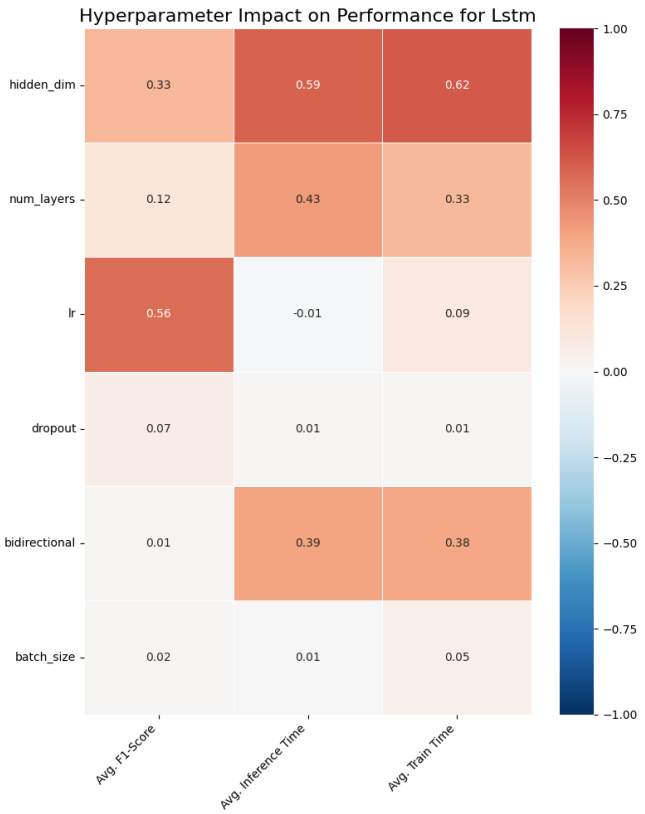


Figure 6.3: LSTM heatmap showing parameter impact

The scatter plot in Figure 6.4 shows these trade-offs. Unlike the tight cluster seen with XGBoost, the results for the LSTM are widely spread, with F1-scores ranging from 0.675 to 0.963. This wide spread confirms that the model is highly sensitive to its hyperparameter setup. A closer look at the plot reveals multiple distinct clusters, of which three will be highlighted:

The optimal cluster, found in the top-left **(1)**, includes configurations that achieve a high F1-score with low inference time. These top-performing models consistently have a `learning_rate` of 0.001 and are not bidirectional, along with a moderate capacity (`hidden_dim` of 64 or 128). In contrast, a small cluster of inefficient configurations in the top-right **(2)** also achieves a high F1-score but at a much higher inference time. These models consistently feature more complex architectural settings, with a large `hidden_dim` of 256, a depth of 2 `num_layers`, and were all bidirectional. Lastly, it is worth noting that nearly all low performing results with an F1-score below 0.75 **(3)** share a lower `learning_rate` of 0.0005, suggesting this was not enough for the model to converge effectively.

This cluster analysis highlights that tuning the LSTM involves a complex process of balancing its architectural capacity and learning rate. Achieving both high accuracy and efficiency makes it a more challenging model to optimize compared to XGBoost.

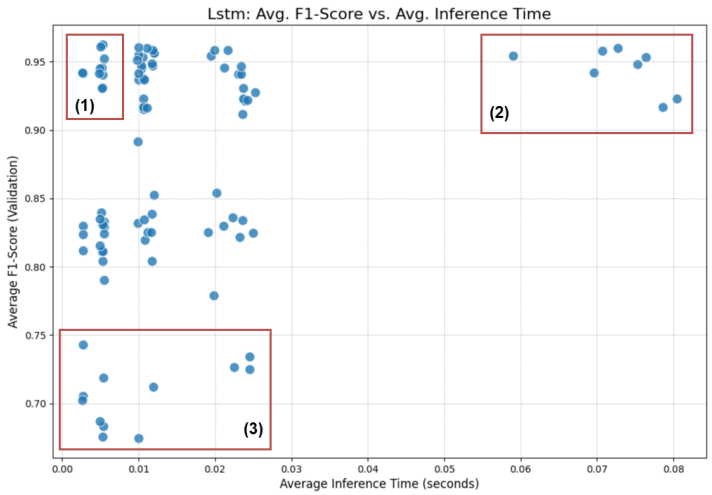


Figure 6.4: LSTM: Performance vs. Efficiency Across Hyperparameter Configurations

6.1.3 Transformer

Similar to the LSTM, the analysis of the hyperparameter sweep for the Transformer model shows its sensitivity to configuration. The correlation heatmap in Figure 6.5 shows that, unlike the other models, the Transformer's performance has a strong positive correlation of 0.75 with its training time. This suggests that configurations that need longer training times generally produce higher F1-scores.

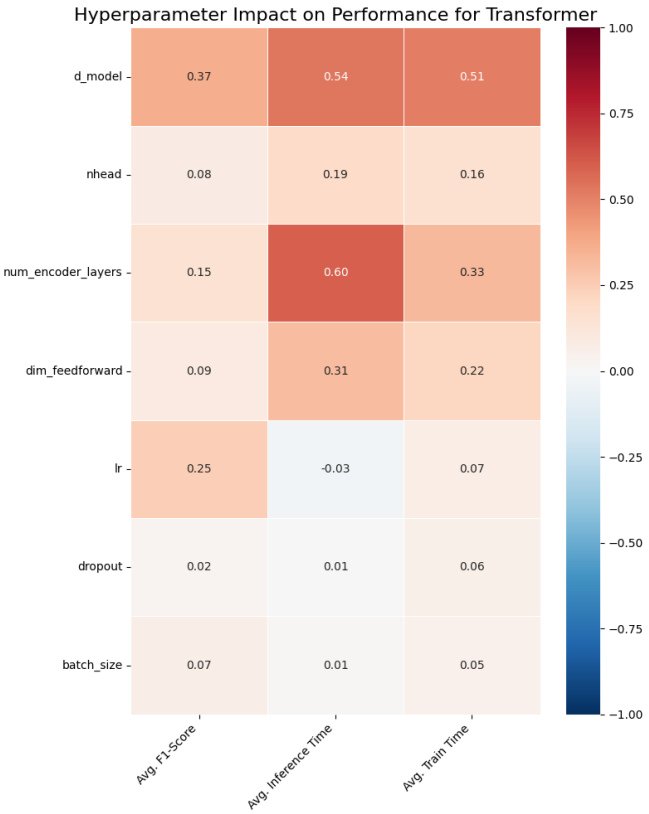


Figure 6.5: Transformer heatmap showing parameter impact

Several architectural parameters are moderately linked to computational cost. The model's capacity, denoted by `d_model`, has a moderate correlation with both `Avg Inference Time` at 0.54 and `Avg Train`

`Time` at 0.51. Similarly, the model's depth, indicated by `num_encoder_layers`, has a moderate correlation with `Avg Inference Time` at 0.60. These correlations show that larger and deeper Transformer models are in general more resource-intensive to train and use. In terms of performance, `d_model` has the strongest direct effect on the `Avg. F1-Score`, showing a moderate positive correlation of 0.37.

The scatter plot in Figure 6.6 shows a wide range of outcomes from different configurations. The F1-score varies significantly from 0.396 to 0.935, while the inference time ranges from 0.0066 to 0.0308 seconds. The broad distribution of F1-scores across a narrow band of inference times highlights the model's high sensitivity to its hyperparameter settings.

The figure does not display as obvious of clusters as the previous models. An analysis of the data points achieving an F1 score above 0.9 **(1)**, had similar configurations for `d_model` (128), `dim_feedforward` (256) and `lr` (0.0001), with singular exceptions. A subtle pattern that stands out in the scatterplot are some distinct vertical clusters. Multiple configurations seem to share similar inference times. An examination of one such cluster **(2)** shows that these configurations have the same core architectural parameters: `d_model`, `nhead`, `num_encoder_layers`, and `dim_feedforward`. This indicates that the model's speed mainly depends on its static architecture.

However, the most important finding is the wide range of F1-scores within this vertical cluster. For the configurations in cluster **(2)**, which all have near-identical inference time, the F1-score varies dramatically from approximately 0.40 to 0.85. This difference is entirely driven by other hyperparameters, such as `learning_rate`, `dropout`, and `batch_size`, but not following a clear pattern.

This demonstrates the Transformer's extreme sensitivity to its training setup. While it can achieve high performance, a poor choice of these non-architectural parameters can result in a model that is near non-functional, even though is as computationally expensive as a high-performing one. This makes the tuning process for the Transformer critical and complex, requiring a careful, systematic approach to find an effective configuration.

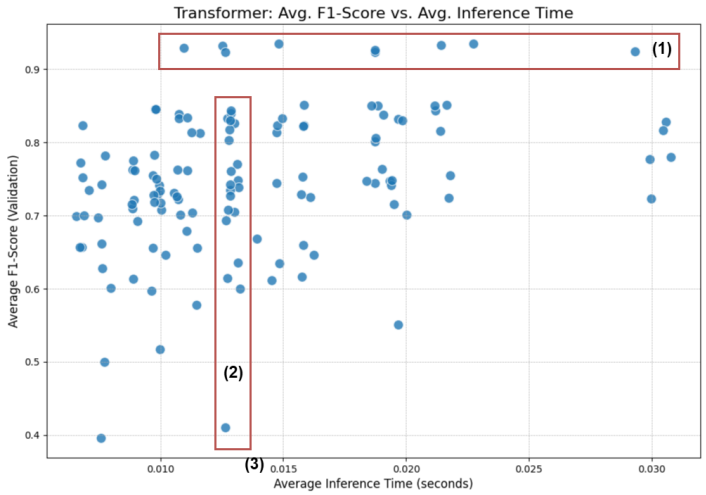


Figure 6.6: Transformer: Performance vs. Efficiency Across Hyperparameter Configurations

6.1.4 Conclusion

The hyperparameter analysis of the XGBoost, LSTM, and Transformer models shows that each architecture has unique tuning traits with important practical implications. This analysis aims to answer the part of RQ1.1, focusing on their tuning complexity and how sensitive they are to configuration changes.

The analysis shows the following range of tuning difficulty:

- **XGBoost** is a strong and predictable model. The scatter plot analysis revealed a broad area of high-performing configurations, showing a low sensitivity to specific hyperparameter choices within the tested range. This makes it the easiest model to optimize.
- **LSTM** is a very complex model to tune. Its performance relies on subtle interactions among its hyperparameters. The scatter plot showed distinct groups for optimal, inefficient, and suboptimal

results. A small change in a key parameter, like `learning_rate`, could lead to a significant and unpredictable shift in performance.

- **Transformer** is also sensitive to its configuration, but it has a clearer and more structured tuning process than the LSTM. Its F1-scores had the widest range, but the vertical clusters in its scatter plot show a more predictable pattern. The model's efficiency depends on its core architecture, while accuracy is adjusted by the other parameters within that fixed performance budget. This makes its tuning process complex, yet more systematic than the LSTM's, which is influenced by less predictable parameter interactions.

This analysis highlights that the difficulty of tuning differs greatly among various architectures. These findings helped choose the best-performing configuration for each model, which will be used in the upcoming comparative experiments.

6.2 Baseline Model Comparison

After the tuning process in the previous Section, the best configuration for each of the three models was selected, according to F1-score and inference time, and this was set as the default configuration. The complete details of these final configurations are listed in Appendix A. This experiment compares the non-sequential XGBoost, the recurrent LSTM, and the attention-based Transformer models. The results in this section demonstrate the trade-offs between performance and efficiency for these different architectural approaches. Note that the hyperparameter sweep was run over two folds and this experiment will follow the 10-Fold framework, resulting in different performance metrics.

Table 6.1 presents the aggregated performance metrics from the 10-fold cross-validation. The table shows the mean and standard deviation for each metric, showcasing both the average performance and consistency of each model across different data splits.

6.2.1 Analysis of Results

The results in Table 6.1 show a clear ranking among the three models. The non-sequential XGBoost model consistently outperformed the two other models across all predictive metrics. It achieved the highest average F1-score (0.9794 ± 0.0073), accuracy (0.9786 ± 0.0076), precision (0.9796 ± 0.0150), recall (0.9793 ± 0.0129), and ROC AUC score (0.9974 ± 0.0026). The difference in computational cost is even more significant. XGBoost shows itself to be very efficient, with a mean inference time of about 0.0005 seconds. In contrast, the mean inference times for the LSTM and Transformer models were 0.0460s and 0.0302s, respectively, making them significantly slower.

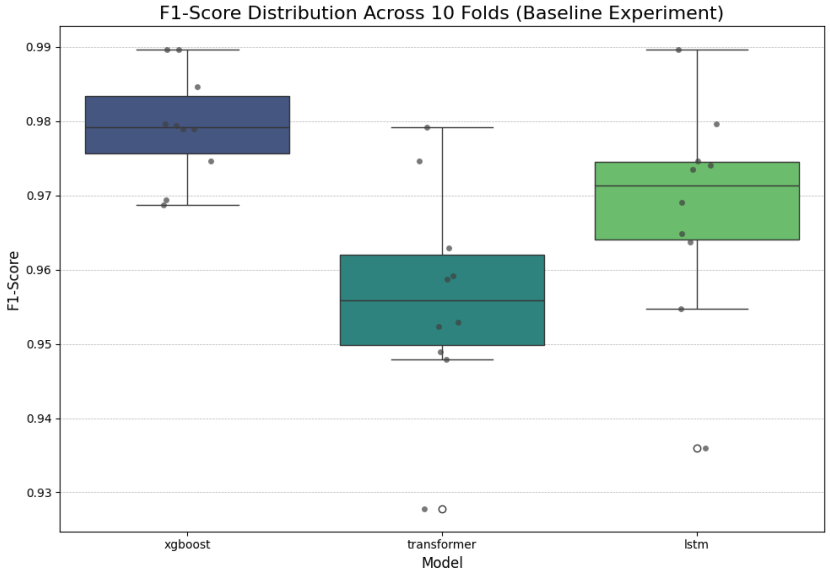
The box plots in Figure 6.7 display the distribution of performance metrics. Although the whiskers for all three models overlap, the median values show that the F1-score for the XGBoost model is entirely above the interquartile range of both the LSTM and Transformer models.

To test if this difference is significant, a Wilcoxon signed-rank test was conducted on the paired F1-scores from the 10 folds. The comparison between XGBoost and LSTM gave a p-value of 0.027, while the comparison between XGBoost and Transformer produced a p-value of 0.004. Since both p-values are below the standard significance level of 0.05, this confirms that the performance difference is statistically significant.

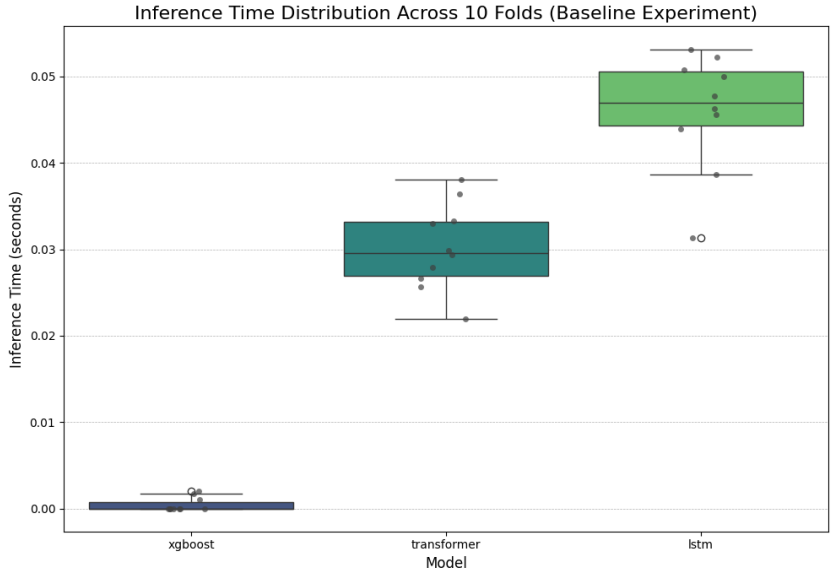
Additionally, the plots reveal differences in consistency. XGBoost has the shortest box and whiskers for both F1-score and inference time, indicating the most stable and predictable performance across the 10 folds. The LSTM and Transformer models have taller boxes, which means they show more variability in their results based on the specific data split. This is particularly clear in the inference time plot. **The box for XGBoost does not overlap with the wider distributions of the LSTM and Transformer, visually illustrating its better efficiency and stability.**

Model	F1-Score	Accuracy	Precision	Recall	ROC AUC	Train (s)	Inference (s)
XGBoost	0.9794 ± 0.0073	0.9786 ± 0.0076	0.9796 ± 0.0150	0.9793 ± 0.0129	0.9974 ± 0.0026	1.41 ± 0.09	0.0005 ± 0.0008
LSTM	0.9680 ± 0.0147	0.9663 ± 0.0161	0.9594 ± 0.0290	0.9773 ± 0.0127	0.9910 ± 0.0054	31.36 ± 10.76	0.0460 ± 0.0067
Transformer	0.9565 ± 0.0144	0.9551 ± 0.0150	0.9608 ± 0.0201	0.9525 ± 0.0202	0.9909 ± 0.0071	21.77 ± 5.91	0.0302 ± 0.0050

Table 6.1: Performance metrics for the best models with mean and standard deviation.



(a) F1-Score Distribution



(b) Inference Time Distribution

Figure 6.7: Distribution of key performance and efficiency metrics across 10 cross-validation folds for the baseline experiment.

6.3 Early Detection Results

This section demonstrates the results of the early detection experiments described in Section 5.4. The goal is to measure the trade-off between detection latency and model performance. The analysis consists of three parts where the first experiment evaluates each model's ability to learn from limited data, the second assesses a fully trained model's performance on incomplete data and lastly these scenarios will be compared.

6.3.1 Experiment A: Learning from Limited Data

The first experiment aimed to find the minimum amount of information each architecture needs during training to create an effective model. To do this, a new model was retrained for all three models and evaluated on different feature sets with trace durations (T_d) ranging from 6 to 30 seconds. The results are shown in Figure 6.8.

The results show that performance for all models improves as the trace duration (T_d) increases. For the F1-Score, shown in Figure 6.8a, the XGBoost model consistently achieves a higher mean score than the LSTM and Transformer models across all trace durations. While the 95% confidence intervals of the sequential models occasionally overlap with XGBoost at longer durations, the mean performance of XGBoost remains higher.

A similar pattern can be seen for the ROC AUC results in Figure 6.8b. The mean ROC AUC for the XGBoost model rises quickly, indicating high classification certainty even with short trace durations. In contrast, the LSTM and Transformer models require a longer observation window to come close to the performance level of XGBoost.

The inference time for the models is displayed in Figure 6.8c and shows a clear distinction. The mean inference time for XGBoost stays consistently low and stable across all trace durations. This is not the case for both the LSTM and Transformer, as these are much higher and increase as the trace duration grows.

6.3.2 Experiment B: Testing with Limited Data

The second experiment simulated a more realistic deployment scenario where a single, fully optimized model must make decisions with incomplete information. For the sequential LSTM and Transformer models, a single model was trained once on the full 30-second data and then evaluated on progressively shorter test sets. The results are presented in Figure 6.9.

The results show a general trend where performance drops as the trace duration (T_d) decreases. In Figure 6.9a, it is interesting to note that while the mean F1-score for the LSTM follows a relatively smooth trajectory, the Transformer's performance shows greater variation across the folds, especially at shorter trace durations. Both models achieve a mean F1-score above 0.80 only after the observation window exceeds 19 seconds. This means that while these fully trained models can be strong in theory, in this case, their learned patterns depend on observing a significant portion of the attack lifecycle to achieve high accuracy.

The ROC AUC results in Figure 6.9b, which measure classification certainty, show that the Transformer model generally performs better than the LSTM at shorter trace durations. As T_d increases, their performance converge, with the first mean intersection occurring at $T_d = 21$ seconds.

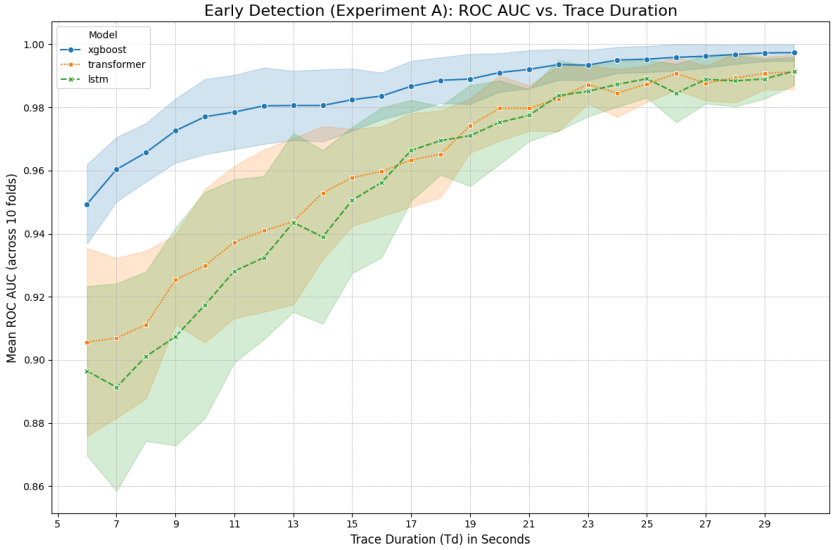
The analysis of inference time, displayed in Figure 6.9c, does not show a clear trend. However, a difference in behavior appears as the trace duration increases, where the average inference time for the LSTM model seems to keep rising, while the inference time for the Transformer model stabilizes after about 15 seconds.

6.3.3 Analysis of Results

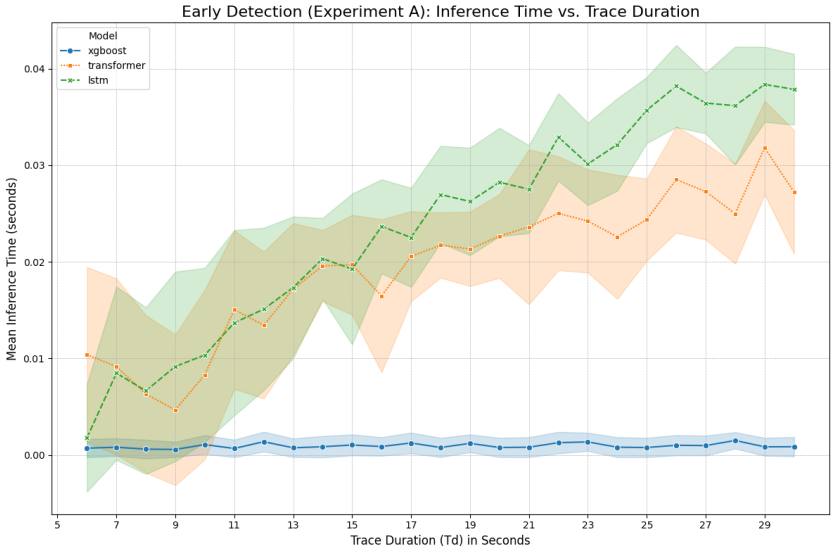
Comparing the results of both experiments offers important insights into the models' behaviors. For the sequential LSTM and Transformer models, the performance in Experiment A, where models were retrained for each T_d, is consistently better than in Experiment B, particularly at shorter trace durations. **This suggests that a model trained specifically for limited data will generally perform better than a general-purpose model when data is sparse.**

Regarding the performance of the models in Experiment A, the XGBoost model consistently outperforms both the LSTM and the Transformer for shorter trace durations. It achieves a significantly higher F1-score and ROC AUC for any observation window below about 20 seconds.

(a) F1-Score vs. Trace Duration

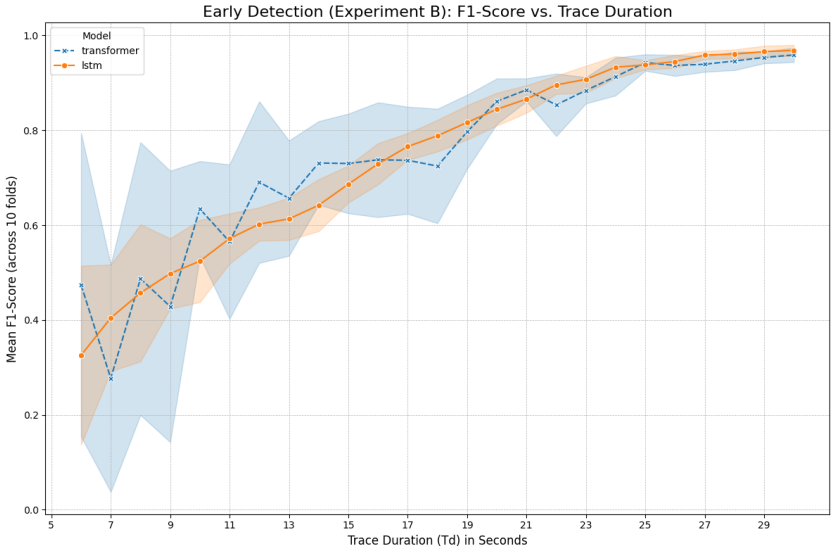


(b) ROC AUC vs. Trace Duration

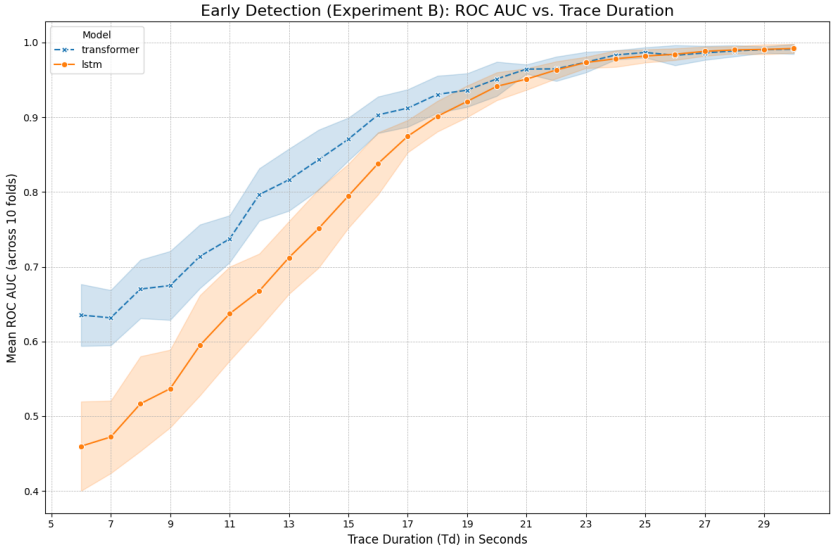


(c) Inference Time vs. Trace Duration

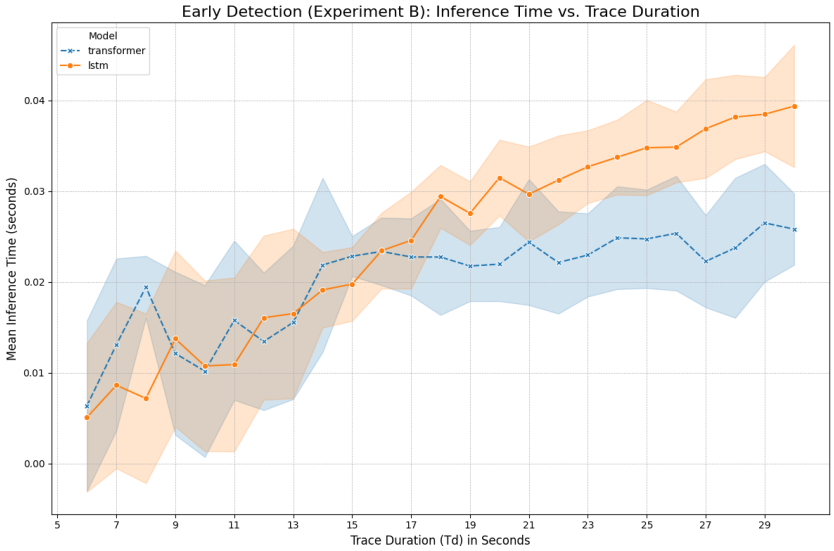
Figure 6.8: F1-Score, ROC AUC and Inference Time for Experiment A.



(a) F1-Score vs. Trace Duration



(b) ROC AUC vs. Trace Duration



(c) Inference Time vs. Trace Duration

Figure 6.9: F1-Score, ROC AUC and Inference Time for Experiment B.

6.4 Zero-Day Generalization Results

This experiment assesses each model's ability to identify new, unseen threats by simulating various zero-day attack scenarios. A Leave-One-Family-Out cross-validation method was used to evaluate how well each model could detect a ransomware family completely excluded from its training data. Recall is the main metric for this evaluation as it measures the percentage of new ransomware attacks successfully detected.

Table 6.2 shows the combined results from 10 runs for each scenario. The table presents the mean Recall and standard deviation for each model against six held-out ransomware families.

Table 6.2: Mean Recall and standard deviation for each model in the Zero-Day Generalization experiment across 10 runs.

Model	Conti	Darkside	Lockbit	Revil	Ryuk	WannaCry	Average Recall
XGBoost	0.926 ± 0.011	0.988 ± 0.004	0.946 ± 0.008	0.742 ± 0.041	0.051 ± 0.007	0.000 ± 0.000	0.609 ± 0.407
LSTM	0.930 ± 0.014	0.990 ± 0.004	0.936 ± 0.021	0.625 ± 0.192	0.052 ± 0.022	0.007 ± 0.011	0.590 ± 0.412
Transformer	0.925 ± 0.029	0.994 ± 0.002	0.959 ± 0.021	0.977 ± 0.035	0.057 ± 0.025	0.018 ± 0.017	0.655 ± 0.434

6.4.1 Analysis of Results

The results in Table 6.2 reveal significant differences in the models' ability to generalize based on the ransomware family tested. For families with more typical behavior, such as Conti, Darkside, and Lockbit, all three models showed high and consistent recall scores, often exceeding 0.92. The performance against the Darkside family was particularly strong, with all models achieving nearly perfect recall.

In contrast, the models faced considerable challenges with Ryuk [26] and WannaCry [25, 45]. The average recall for all three models against these two ransomware families was exceptionally low, often below 0.06. This suggests that the behavior of Ryuk and WannaCry differs significantly from other families in the training set, preventing the models from applying their learned knowledge effectively. For example, the XGBoost model recorded a mean recall of 0.000 for WannaCry, indicating it could not detect any of the unseen samples.

When evaluating overall generalization capabilities, the Transformer model attained the highest average recall of 0.655 across all six zero-day scenarios, mainly due to its strong performance in detecting Revil samples, achieving a recall of 0.977. The XGBoost and LSTM models had similar overall average recall scores of 0.609 and 0.590. The high standard deviation in average recall across models highlights the notable variation in performance against different families.

The varying range of performance for all three models among different families shows a key challenge, where the learned behavior patterns are not universal and do not apply to ransomware that uses differing attack strategies.

Chapter 7

Discussion

The previous chapter presented the results of the experiments. This chapter offers a deeper look at those findings. It connects them to the research questions and discusses their wider impact on ransomware detection by going beyond the statistical results. The chapter starts with a summary of the key findings, then offers a deeper interpretation of each experimental result. Finally, it will answer the research questions and address the study's limitations and present potential future work.

7.1 Summary of Key Findings

The experimental results in Chapter 6 showed several findings about the performance, efficiency, and generalization of the evaluated models. The main outcomes of the investigation are summarized below:

- The non-sequential XGBoost model showed better overall performance in the baseline comparison. It consistently outperformed both the recurrent LSTM and the attention-based Transformer models in all primary metrics for predictive accuracy and computational efficiency when tested on complete data traces.
- Early detection performance relies heavily on the available data context. The experiments indicated a clear trade-off between detection speed and accuracy. While the sequential models eventually performed well with longer data traces, the non-sequential XGBoost model was more effective in making reliable predictions when the observation window was much smaller.
- Generalizing to unseen ransomware families poses a significant challenge. The zero-day experiment showed a wide range in the models' ability to detect new threats. Although all models were able to recognize some unseen ransomware families, they had a hard time detecting others. This suggests that the behavioral patterns learned from the training set did not apply to all types of ransomware.

These findings are the basis for the upcoming discussion, where the implications of each outcome will be examined in more detail.

7.2 Interpretation of Experimental Results

The results in the previous chapter show a series of subtle and, in some cases, surprising outcomes. This section provides a deeper look at these findings. It explores the reasons behind the observed behaviors of the models and discusses their real-world implications for ransomware detection. The analysis begins by examining the most important finding from the baseline comparison: the unexpected success of the non-sequential XGBoost model. Next, it addresses the real-time detection challenges by combining results from the early detection experiments. Finally, an examination of the zero-day generalization test, noting the limitations of behavior-based detection when encountering new ransomware families.

7.2.1 The Efficacy of a Non-Sequential Model

The most significant and perhaps unexpected finding from the baseline comparison is that the non-sequential XGBoost model performs better than the recurrent LSTM and attention-based Transformer across all main metrics, including F1-score, accuracy, recall, and ROC AUC. This outcome raises an

important question: why would a model that ignores the order of events be more effective than models designed to analyze sequences?

The most plausible reason for this lies in the feature engineering process described in Section 3.3. Transforming raw, high-frequency hypervisor events into fixed-time windows, each summarized by 23 statistical features, is a key step in simplifying information. This process keeps the overall sequence of behaviors, but combines the detailed events within each window. The resulting feature set seems to contain a signal that relies more on the size and combination of these statistical indicators throughout the entire trace, rather than their specific order.

XGBoost excels in this situation. By converting the sequence of windows into a single high-dimensional feature vector, it effectively identifies complex, non-linear relationships among any of the features, regardless of their timing. In contrast, the sequential models can only learn from the order of the windows. This is especially important for LSTMs, which often have difficulty remembering subtle events in very long sequences. Although the Transformer's self-attention mechanism can theoretically capture these long-range dependencies, the most important signals might be sparse. They could show up as specific combinations of features across distant, noncontiguous time windows. XGBoost, by treating the entire trace as one flat feature space, is naturally good at finding these sparse, combinatorial patterns without the limits of sequential processing. Although it is possible that a more extensive hyperparameter search or a deeper neural architecture could have improved the performance of the sequential models slightly, the clear advantage of XGBoost in both performance and efficiency indicates a stronger match between the model and the data representation.

This finding does not imply that sequential modeling is ineffective. Instead, it highlights that the feature engineering in this study creates a data representation well-suited to a non-sequential classifier like XGBoost. This insight shows that the choice of model architecture should be closely linked to the type of features analyzed.

7.2.2 The Trade-off Between Latency, Accuracy, and Practicality

The early detection experiments further highlight the practical challenges of a real-time system. The results from Experiment A, where models were fully retrained on shorter traces, confirm XGBoost's strength. For any observation window under about 24 seconds, XGBoost was the fastest and most accurate model, indicating that it is the best choice when information is limited.

However, this also reveals a limitation of a static model like XGBoost. As a non-sequential model, it needs a fixed-size input vector. A model trained on 10-second traces cannot process a 30-second trace, and the reverse is also true. Therefore, a real-world system aiming for maximum accuracy at all possible trace durations would need to maintain a separate, retrained XGBoost model for every possible T_d, which is not feasible

This is where the benefits of sequential models become clear. Experiment B showed that a single sequential model, trained on full 30-second traces, is able to handle variable-length, incomplete data. Although its accuracy was lower at shorter intervals, this flexibility is an important practical advantage.

A solution that could potentially combat this would be a more practical and robust tiered detection strategy that uses the unique strengths of both model types. The goal is to provide immediate early detection and high-confidence, context-aware confirmations. In such a system:

- **Tier 1 (Early Detection)** would feature a single, lightweight XGBoost model, specifically trained on short, fixed-length traces (e.g., 10 seconds). Its main role is to serve as a low-latency detection layer. This tier sends a quick alert when it detects a suspicious process, allowing immediate actions, such as quarantining the process to contain the threat.
- **Tier 2 (Confirmation)** would involve a fully-trained sequential model (e.g., a Transformer). This model activates only after receiving an alert from Tier 1. The system would continue to gather data up to a full specified amount of seconds, and this complete, variable-length trace would be analyzed by the sequential model for a final, high-confidence decision.

The case for this second tier is not that the Transformer is more accurate on this dataset, as according to the experiments it is not. Instead, the argument centers on its theoretical ability to combat sophisticated future threats. The second tier model could even be trained on the volume-rich raw data, as tier 1 has ensured the threat is contained, and the second tier can therefore have a relatively slower inference time.

Furthermore, research shows that ransomware can mimic the statistical patterns of benign software [7]. A sequential model, by examining the order of events, is theoretically better equipped to detect

such stealthy threats than a model that only analyzes a flattened statistical summary. This does not mean they are immune to adversarial attacks, but their ability to analyze the context and order of events does provide a complementary defensive layer to the purely statistical approach of XGBoost, making the entire system harder to evade when combined. The tiered approach thus represents a practical strategy that combines the proven speed and accuracy of XGBoost against common threats with the more robust sequential models as protection against advanced evasion tactics. It is important to recognize that this tiered system must fit into a larger security framework. This framework should include ways to manage alert volume and prevent possible denial-of-service attacks on the secondary analysis tier. However, as a key detection method, the two-tiered approach remains a strong and practical design.

7.2.3 The Challenge of Generalization

The zero-day experiment offers insight into the limitations of behavior-based systems. The near-total failure of all three models to detect the Ryuk and WannaCry families shows that even low-level behavioral patterns are not universal.

This indicates two challenges, which are the limitation of the feature set and difficulty for the models in learning a true abstract representation of what is malicious. One possible reason for this failure relates to the specific attack methods of these ransomware families. WannaCry, for example, is known for its self-replicating behavior, which creates substantial network-level signals that the purely storage- and memory-focused feature set used in this research does not capture [3]. Similarly, Ryuk is recognized for targeting remote network shares, producing I/O patterns that differ fundamentally from the local file encryption typical of the other families in the training set [26]. The models seem to have learned the specific patterns of local file encryption effectively, but they did not extend that understanding to identify malicious I/O in a different context. If a new ransomware variant employs a sufficiently different attack chain, a model trained on past data may not recognize it, no matter how sophisticated its design.

This should not be seen as a limitation of hypervisor-level monitoring itself. It successfully captured the true events. Instead, it highlights a challenge for the feature engineering and modeling layers built on top of it. A future system could overcome this by including a better feature set that covers network I/O, or by using better models that can learn from a wider variety of malicious behaviors. Ultimately, this finding suggests that although behavior-based detection is a strong tool, a complete defense strategy will likely always need a hybrid approach, combining machine learning with other security layers to address the constantly changing and unpredictable nature of new threats.

7.3 Answering the Research Questions

This section provides direct answers to the research questions from Chapter 1, based on the findings from the experimental evaluation:

- **RQ1.1:** *How do non-sequential, recurrent, and attention-based models compare in their empirical trade-offs between predictive performance, computational efficiency, and tuning complexity?*
  The non-sequential XGBoost model gave the best overall trade-off for this specific feature set. The baseline comparison in Section 6.2 showed that XGBoost was the most efficient model in terms of computation, achieving the highest predictive accuracy across all main metrics. Furthermore, the hyperparameter analysis in Section 6.1 indicated that it was the easiest to tune, showing predictable and stable performance.
- **RQ1.2:** *What is the impact of detection latency on model accuracy, and what is the minimum observation window required for reliable detection?*
  Detection latency significantly affects model accuracy, and the minimum required observation window varies by model architecture. The early detection analysis in Section 6.3 showed a clear positive link between trace duration and F1-score for all models. For learning from sparse data (Experiment A), the non-sequential XGBoost model performed best at shorter durations, achieving an F1-score above 0.9 after just 7 seconds of observation. The sequential models needed a much longer window to reach a similar level of performance.
- **RQ1.3:** *To what extent can the models generalize their learned behavioral patterns to identify ransomware variants not seen during training?*
  The models' ability to generalize depends heavily on the behavioral similarity between the unseen variant and the training set. The zero-day experiment in Section 6.4 found that all three models could successfully detect some unseen families (e.g., Conti, Darkside) with high recall. However,

they struggled to detect families with fundamentally different attack methods (e.g., Ryuk and Wannacry), which were not well-represented by the feature set. This shows that while behavior-based detection is a strong approach, its effectiveness against new threats is limited by the variety of behaviors found in its training data.

RQ1: *How well can machine learning models, trained on hypervisor-level behavioral data, detect known and unknown ransomware with minimal latency?*

Machine learning models can detect known ransomware with high accuracy, achieving an F1-score of up to 0.98. However, their effectiveness greatly depends on the model architecture and the available data context. A non-sequential XGBoost model is the most effective, especially with minimal latency. It can achieve reasonable detection performance within 7 seconds of an observation. On the other hand, detecting unknown ransomware is inconsistent. It succeeds only when the new variant shares behavioral patterns with the training data and fails otherwise.

7.4 Limitations of the Study

This research offers insights into detecting ransomware using hypervisor-level data. However, it must be acknowledged that there are limitations which affect the conclusions. These limitations involve the dataset, the methodology, and the practical use of the findings. They provide essential context for interpreting the results and suggest areas for future work.

Dataset and Feature Limitations

The conclusions in this thesis are limited by the scope and nature of the RanSMAP dataset. The experiments included a relatively small set of six ransomware families and six benign applications. Therefore, the findings on model performance and generalization may not be relevant to the wide and constantly evolving landscape of real-world ransomware variants. This was evident in the zero-day experiment, where the models did not detect families whose behaviors were not well represented in the training data.

Furthermore, the dataset is balanced, with nearly equal numbers of benign and malicious traces. This is not realistic in a real-world environment, where malicious activity is rare. This class balance may have affected how the models learned to make decisions and could result in an underestimation of the false positive rate in practical use. Likewise, the limited benign applications may not reflect the complexity and noise of typical user activity on an enterprise network. The benign applications mainly imitate high-intensity I/O operations, which help in the encryption phase of an attack. However, they do not accurately represent other critical steps, like data exfiltration. This limits the model's ability to tell apart different types of malicious and benign system activity. Lastly, focusing solely on memory and storage access patterns means the system misses other important signs of compromise, such as network C2 communication or the creation of malicious processes.

Methodological Limitations

The experimental methodology has its own limitations. The feature engineering process aggregates raw events into statistical summaries over fixed time windows, resulting in a loss of detailed temporal information. An attack that happens within a single window might get averaged out and missed by the models. Furthermore, the models were trained and evaluated on a static dataset gathered at a specific time. In real life, ransomware tactics continuously change, meaning a model that works well today may become ineffective as attackers alter their techniques to avoid detection, which is a phenomenon known as concept drift. Lastly, while the hyperparameter sweep was thorough, it was not comprehensive. Different search spaces or better tuning algorithms might discover configurations for the LSTM or Transformer models that could close the performance gap with XGBoost.

Real-World Applicability Limitations

Applying these offline experimental results to a live environment presents several challenges. The study used pre-collected traces, which do not take into account the performance overhead and data processing delays of hypervisor-level monitoring. The reported inference times only refer to the classification step and do not reflect the entire detection process. The specific hardware used in these experiments also affects both the training and inference times. While the relative performance differences among models

are valid, the exact timings may vary on other systems. Additionally, the analysis is limited due to the lack of ground-truth labels for specific attack stages in the traces. While early detection experiments evaluate performance based on the duration of the traces, it cannot be clearly determined when detection happens during the attack timeline. Finally, although it suggests that sequential models may be more resistant to adversarial mimicry, this was not tested directly. A skilled attacker could design a ransomware variant that mimics the statistical profile of benign software, allowing it to evade detection by any of the models evaluated in this study.

7.5 Future Work

The findings and limitations of this research open up several promising opportunities for future work. The following sections outline potential directions for building on this study's methods and results to further advance the field of hypervisor-based ransomware detection.

Expanding Data and Features

A key area for future research is enhancing the dataset and feature set to create a better view of system behavior. One next step would be to incorporate network and process-level features, such as network flow data, DNS requests, and system call sequences. These could be collected at the hypervisor level by examining virtual network interfaces and monitoring guest OS process tables. This approach would maintain the same low-level, tamper-resistant perspective as the current feature set. As the zero-day analysis showed, the current model's lack of awareness of network activity is an important limitation, and adding these features would likely boost its ability to detect variants like WannaCry.

Additionally, testing the models on a broader dataset with more diverse ransomware families and a greater variety of benign applications would provide a more thorough test of their real-world generalization abilities. Finally, a dedicated experiment on the "mixed" traces, which is an existing folder in the RanSMAP dataset containing concurrent activity of both a benign application and a ransomware variant, should take place to assess each model's robustness to the background noise. Finally, future work could involve creating a new and more realistically imbalanced dataset. The balanced nature of the RanSMAP dataset is convenient for training, but evaluating the models on a dataset that reflects the real-world rarity of ransomware events would offer a much more accurate assessment of their potential false positive rates.

Exploring Advanced Modeling Techniques

While this thesis examined three well-established architectures, the field of machine learning is always changing. Future work could look into training end-to-end models directly on the raw sequence of hypervisor events. This method would skip the current feature engineering step and could help a model discover more detailed temporal patterns that are lost during statistical aggregation.

In line with this, investigating newer architectures such as State Space Models (e.g., Mamba) is a promising direction. As explained in Section 3.4.6, architectures like Mamba provide linear-time scaling, making them theoretically better suited for analyzing very long behavioral traces than Transformers, in the case where the model is trained on raw data. However, implementing these often requires specialized GPU hardware, which was a practical restraint for this research. Future work with access to such resources could focus on evaluating these emerging models. Another promising direction is using Graph Neural Networks (GNNs), which would involve turning the sequential traces into provenance graphs to more explicitly model the causal relationships between system events.

Improving Real-World Applicability

To connect this offline analysis with a practical, deployable solution, several additional steps could be taken. The most immediate next step would be an empirical evaluation of the tiered detection strategy proposed in the discussion. This would include implementing and measuring the end-to-end performance of the combined XGBoost early detection and sequential confirmation models. To further test the theoretical resilience of this system, future work should involve adversarial resilience testing, where adversarial ransomware samples are created to intentionally mimic benign profiles and examine the system for weaknesses. Lastly, developing a live system prototype that operates on a real-time data

stream from a hypervisor would be crucial for analyzing the actual end-to-end detection latency and the monitoring performance overhead.

Chapter 8

Conclusion

This thesis explored to what capacity machine learning models can identify known and unknown ransomware by looking at low-level behavioral data collected at the hypervisor. Through a study of non-sequential, recurrent, and attention-based models, this research produced several findings regarding the field of cybersecurity.

The main finding is that for the specific feature set created from the RanSMAP dataset [10], the non-sequential XGBoost model outperformed the more complex LSTM and Transformer models. This result highlighted that the way features are represented plays a crucial role in model selection, sometimes outweighing the sequential nature of the data.

Additionally, the research measured the practical trade-offs between detection speed and accuracy. It found that while early detection is possible, reliability improves significantly with a longer observation period. The study also showed the limits of behavior-based detection. The models generally performed well with some unseen ransomware families but struggled with those exhibiting fundamentally different behaviors. This shows that having a diverse training set is essential for robust protection against zero-day threats.

In conclusion, this thesis shows that monitoring at the hypervisor level has potential of being an effective method for behavior-based ransomware detection. It answers the main research question by showing that a non-sequential model provides the best balance between performance and efficiency in this specific area.

Bibliography

[1] S. H. Kok, A. Abdullah, N. Z. Jhanjhi, and M. Supramaniam, "Ransomware, threat and detection techniques: A review," *International Journal of Computer Science and Network Security*, vol. 19, no. 2, pp. 136–146, 2019. [Online]. Available: http://paper.ijcsns.org/07_book/201902/20190217.pdf.

[2] C. Beaman, A. Barkworth, T. D. Akande, S. Hakak, and M. K. Khan, "Ransomware: Recent advances, analysis, challenges and future research directions," *Computers Security*, vol. 111, p. 102 490, 2021, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2021.102490. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016740482100314X.

[3] Q. Chen and R. A. Bridges, *Automated behavioral analysis of malware: A case study of wannacry ransomware*, 2017.

[4] M. Mimoso, *Maersk shipping reports $300m loss stemming from notpetya attack*, Aug. 2017. [Online]. Available: https://threatpost.com/maersk-shipping-reports-300m-loss-stemming-from-notpetya-attack/127477/.

[5] R. A. Lika, D. A. Murugiah, D. A. V. Ramasamy, and S. N. Brohi, *Notpetya: Cyber attack prevention through awareness via gamification*, 2017.

[6] P. O'Kane, S. Sezer, and D. Carlin, "Evolution of ransomware," *IET Networks*, vol. 7, no. 5, pp. 373–383, 2018. DOI: 10.1049/iet-net.2017.0207. [Online]. Available: https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/iet-net.2017.0207.

[7] Y. Xu, Y. Chen, Z. Zhou, and Y. Lin, "Adversarial i/o workloads: Exploiting storage behavior to evade ransomware detection," in *Proceedings of the 44th IEEE Symposium on Security and Privacy*, Demonstrates how ransomware can imitate benign I/O to bypass detection, 2023.

[8] A. Zaal, *Unleashing direct syscalls: Evading edr detection*, 2024. [Online]. Available: https://redfoxsec.com/blog/unleashing-direct-syscalls-evading-edr-detection/.

[9] Q. Zhu, M. Li, W. Liu, J. Zhang, and D. Gao, "Early detection of ransomware by monitoring program execution and system calls," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 2, pp. 1057–1071, 2022, Details the latency vs. detection trade-off in ransomware defense. DOI: 10.1109/TDSC.2022.3144680.

[10] M. Hirano and R. Kobayashi, "Ransmap: Open dataset of ransomware storage and memory access patterns for creating deep learning based ransomware detectors," *Computers Security*, vol. 150, p. 104 202, 2025, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2024.104202. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404824005078.

[11] M. Hirano and R. Kobayashi, "Machine learning-based ransomware detection using low-level memory access patterns obtained from live-forensic hypervisor," in *2022 IEEE International Conference on Cyber Security and Resilience (CSR)*, IEEE, Jul. 2022, pp. 323–330. DOI: 10.1109/csr54599.2022.9850340. [Online]. Available: http://dx.doi.org/10.1109/CSR54599.2022.9850340.

[12] N. Idika and A. Mathur, "A survey of malware detection techniques," *Purdue University*, Mar. 2007.

[13] A. P. Namanya, A. Cullen, I. U. Awan, and J. P. Disso, "The world of malware: An overview," in *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*, 2018, pp. 420–427. DOI: 10.1109/FiCloud.2018.00067.

[14] J. Beerman, D. Berent, Z. Falter, and S. Bhunia, *A review of colonial pipeline ransomware attack*, 2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW), 2023. DOI: 10.1109/CCGridW59191.2023.00017.

[15] U. H. of Representatives Committee on Oversight and G. Reform, "The opm data breach: How the government jeopardized millions of americans' personal data," U.S. Government Printing Office, 2016. [Online]. Available: https://web.archive.org/web/20180921190218/https://oversight.house.gov/wp-content/uploads/2016/09/The-OPM-Data-Breach-How-the-Government-Jeopardized-Our-National-Security-for-More-than-a-Generation.pdf.

[16] Ö. A. Aslan and R. Samet, "A comprehensive review on malware detection approaches," *IEEE Access*, vol. 8, pp. 6249–6271, 2020. DOI: 10.1109/ACCESS.2019.2963724.

[17] K. Victor, *Reflective loading runs netwalker fileless ransomware*, 2020. [Online]. Available: https://www.trendmicro.com/en_us/research/20/e/netwalker-fileless-ransomware-injected-via-reflective-loading.html#:~:text=Threat%20actors%20are%20continuously%20creating,the%20system%20to%20initiate%20attacks.

[18] T. Garfinkel and M. Rosenblum, "Virtual machine introspection: A bridge between virtualization and security," *IEEE Security Privacy*, vol. 5, no. 4, pp. 66–70, 2007, Discusses the semantic gap and challenges of interpreting low-level VM data. DOI: 10.1109/MSP.2007.102.

[19] A. Lutaş, D. Ticle, and O. Creț, "Hypervisor-based memory introspection: Challenges, problems and limitations," in *Proceedings of the 3rd International Conference on Information Systems Security and Privacy (ICISSP)*, SciTePress, 2017, pp. 285–294. DOI: 10.5220/0006210202850294.

[20] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," *Proceedings of the 15th ACM conference on Computer and communications security (CCS)*, pp. 51–62, 2008, Evaluates the performance overhead of trapping page access via EPT and VM-exits. DOI: 10.1145/1455770.1455779.

[21] M. Gaber, M. Ahmed, and H. Janicke, "Zero day ransomware detection with pulse: Function classification with transformer models and assembly language," *Computers Security*, vol. 148, p. 104 167, 2025, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2024.104167. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404824004723.

[22] L. O'Donnell-Welch, *Blackcat ransomware actors use malicious drivers to evade detection*, 2023. [Online]. Available: https://duo.com/decipher/blackcat-ransomware-actors-use-malicious-drivers-to-evade-detection#:~:text=The%20BlackCat%20ransomware%20group%20has,and%20fly%20under%20the%20radar.

[23] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "Holmes: Real-time apt detection through correlation of suspicious information flows," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1137–1152. DOI: 10.1109/SP.2019.00026.

[24] V. Canja, *Hypervisor introspection redefines security for virtualized environments*, Jun. 2016. [Online]. Available: https://www.bitdefender.com/en-us/blog/businessinsights/hypervisor-introspection-security-virtualized-environments.

[25] United States Computer Emergency Readiness Team (US-CERT), *Indicators associated with wannacry ransomware*, Jun. 2018. [Online]. Available: https://www.cisa.gov/uscert/ncas/alerts/TA17-132A.

[26] Cybersecurity and Infrastructure Security Agency (CISA), *Ransomware activity targeting the healthcare and public health sector*, Nov. 2020. [Online]. Available: https://www.cisa.gov/uscert/ncas/alerts/aa20-302a.

[27] Cybersecurity and Infrastructure Security Agency (CISA), *Conti ransomware*, Mar. 2022. [Online]. Available: https://www.cisa.gov/news-events/alerts/2021/09/22/conti-ransomware.

[28] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001, ISSN: 1573-0565. DOI: 10.1023/A:1010933404324. [Online]. Available: https://doi.org/10.1023/A:1010933404324.

[29] M. Hirano, R. Hodota, and R. Kobayashi, "Ransap: An open dataset of ransomware storage access patterns for training machine learning models," *Forensic Science International: Digital Investigation*, vol. 40, p. 301 314, 2022, ISSN: 2666-2817. DOI: https://doi.org/10.1016/j.fsidi.2021.301314. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666281721002390.

[30] Z. Wang *et al.*, "Ransom access memories: Achieving practical ransomware protection in cloud with deftpunk," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, Santa Clara, CA: USENIX Association, 2024. [Online]. Available: https://www.usenix.org/conference/osdi24/presentation/wang-zhongyu.

[31] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," *CoRR*, vol. abs/1603.02754, 2016. arXiv: 1603.02754. [Online]. Available: http://arxiv.org/abs/1603.02754.

[32] P. S. Yoshua Bengio and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE TRANSACTIONS ON NEURAL NETWORKS*, Mar. 1994. [Online]. Available: https://www.comp.hkbu.edu.hk/~markus/teaching/comp7650/tnn-94-gradient.pdf.

[33] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997, ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735.

[34] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "Lstm: A search space odyssey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 10, pp. 2222–2232, 2017. DOI: 10.1109/TNNLS.2016.2582924.

[35] R. Zhao, D. Wang, R. Yan, K. Mao, F. Shen, and J. Wang, "Machine health monitoring using local feature-based gated recurrent unit networks," *IEEE Transactions on Industrial Electronics*, vol. 65, no. 2, pp. 1539–1548, 2018. DOI: 10.1109/TIE.2017.2733438.

[36] S. Kiranyaz, O. Avci, O. Abdeljaber, T. Ince, M. Gabbouj, and D. J. Inman, "1d convolutional neural networks and applications: A survey," *Mechanical Systems and Signal Processing*, vol. 151, p. 107 398, 2021, ISSN: 0888-3270. DOI: https://doi.org/10.1016/j.ymssp.2020.107398. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0888327020307846.

[37] S. Bai, J. Z. Kolter, and V. Koltun, "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling," *CoRR*, vol. abs/1803.01271, 2018. arXiv: 1803.01271. [Online]. Available: http://arxiv.org/abs/1803.01271.

[38] Y. Liu, H. Dong, X. Wang, and S. Han, "Time series prediction based on temporal convolutional network," in *2019 IEEE/ACIS 18th International Conference on Computer and Information Science (ICIS)*, 2019, pp. 300–305. DOI: 10.1109/ICIS46139.2019.8940265.

[39] A. Vaswani *et al.*, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017. arXiv: 1706.03762. [Online]. Available: http://arxiv.org/abs/1706.03762.

[40] L. D. Manocchio, S. Layeghy, W. W. Lo, G. K. Kulatilleke, M. Sarhan, and M. Portmann, "Flowtransformer: A transformer framework for flow-based network intrusion detection systems," *Expert Systems with Applications*, vol. 241, p. 122 564, May 2024, ISSN: 0957-4174. DOI: 10.1016/j.eswa.2023.122564. [Online]. Available: http://dx.doi.org/10.1016/j.eswa.2023.122564.

[41] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *CoRR*, vol. abs/1901.00596, 2019. arXiv: 1901.00596. [Online]. Available: http://arxiv.org/abs/1901.00596.

[42] A. Gu and T. Dao, *Mamba: Linear-time sequence modeling with selective state spaces*, 2024. arXiv: 2312.00752 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2312.00752.

[43] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* (Springer Series in Statistics), 2nd. New York, NY: Springer, 2009, ISBN: 978-0-387-84857-0. DOI: 10.1007/978-0-387-84858-7. [Online]. Available: https://doi.org/10.1007/978-0-387-84858-7.

[44] H. Akoglu, "User's guide to correlation coefficients," *Turk J Emerg Med*, vol. 18, no. 3, pp. 91–93, Aug. 2018. DOI: 10.1016/j.tjem.2018.08.001.

[45] M. N. Alenezi, H. Alabdulrazzaq, A. A. Alshaher, and M. M. Alkharang, "Evolution of malware threats and techniques: A review," *International Journal of Communication Networks and Information Security*, vol. 12, no. 3, pp. 326–334, 2020. DOI: 10.17762/ijcnis.v12i3.4723. [Online]. Available: https://www.researchgate.net/publication/349324759.

Appendix A

Hyperparameter Configurations

Table A.1: Best-performing hyperparameter configurations identified by the sweep.

Parameter	Value
XGBoost	
n_boost_round	200
lr	0.05
max_depth	7
row_sample	0.8
col_sample	1.0
LSTM	
hidden_dim	64
num_layers	2
lr	0.001
dropout	0.3
bidirectional	False
batch_size	32
Transformer	
d_model	128
nhead	8
num_encoder_layers	2
dim_feedforward	256
lr	0.0001
dropout	0.1
batch_size	16