UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Metamorphic Malware Detection with Machine Learning

Yuri Lopes Jorge Chiado de Andrade

Mestrado em Segurança Informática

Dissertação orientada por:
Prof. Doutor André Osório e Cruz de Azerêdo Falcão
e co-orientada pelo Prof. Doutor Nuno Fuentecilla Maia Ferreira Neves

2024

Acknowledgments

First and foremost, I would like to express my profound gratitude to my advisors, Prof. André Falcão and Prof. Nuno Neves, for their incredible patience, kindness, and guidance throughout this thesis.

I would also like to thank my parents, for raising me and for the incredible love and support they have given me throughout my life. To my mother, for being nurturing and caring, and always trying to impart strength and perseverance to me. I hope you'll be by my side for many years to come.

And to my father, who passed away during the writing of this thesis, for being my best friend and role model, and for always trying to cheer me up and comfort me. I miss you every single day. It's only because of their unwavering support and belief in me that I was able to finish this thesis. I hope I made them at least a little bit proud.

Finally, I'd like to thank all my friends for the company and encouragement they've given me on this journey. Your kindness always brightens my day. I'm looking forward to sharing many more good times.

Special thanks to Prof. Fabio Di Troia for sharing the Malicia dataset used in this study, and to Prof. Mark Stamp for offering access to the Linux metamorphic malware (MWOR) dataset.

This work was partially supported by FCT through the LASIGE Research Unit, ref. UIDB/00408/2020[1] and ref. UIDP/00408/2020[2].

[1] https://doi.org/10.54499/UIDB/00408/2020
[2] https://doi.org/10.54499/UIDP/00408/2020

In loving memory of my father.

Resumo

Nos últimos anos, o aumento significativo tanto em quantidade como em sofisticação do malware tem gerado desafios substanciais para a sua deteção. Além da ocorrência esporádica de malware zero-day, têm sido aplicadas técnicas de ofuscação cada vez mais complexas a malware existente, criando novas variantes capazes de contornar os métodos tradicionais de deteção. O malware metamórfico utiliza estas técnicas para modificar completamente o código, mantendo apenas o comportamento do programa, o que resulta num maior nível de evasão. A deteção baseada em assinaturas, a abordagem usada pelas ferramentas antivírus tradicionais, é ineficaz contra este tipo de malware.

A literatura recente propõe o uso de técnicas de aprendizagem automática como uma solução para melhorar a deteção, mas há ainda a falta de uma metodologia robusta capaz de identificar todas as variantes de malware, suportada por testes rigorosos em datasets diversos. Além disso, é necessário considerar a eficiência computacional da abordagem proposta, de modo a ser possível realizar a deteção em tempo real em diferentes cenários e dispositivos, incluindo dispositivos IoT, que são um alvo crescente de ataques.

Problema e Motivação Os ataques cibernéticos têm vindo a aumentar tanto em número quanto em sofisticação, com o custo global do cibercrime projetado para alcançar 10,5 trilhões de dólares até 2025, afetando empresas de todos os tamanhos, das quais 60% encerram as suas atividades após uma violação de dados. O malware é uma ferramenta utilizada em várias fases de um ataque, sendo, portanto, um importante Indicador de Ataque (IoA). Quando identificado de forma atempada, pode ser utilizado para detectar e neutralizar o ataque antes que a exfiltração de dados ocorra, minimizando os danos causados.

A deteção de malware, tal como outras áreas da cibersegurança, requer a recolha e processamento de grandes volumes de dados. Neste contexto, a automação torna-se indispensável, e a eficácia das ferramentas de inteligência artificial (IA) desempenha um papel crucial no futuro da segurança digital. Organizações que implementam IA de automação nas suas processos de segurança conseguem reduzir substancialmente os custos e o tempo de resposta a violações de dados, o que tem impulsionado um aumento de 3% na sua adoção no último ano.

Deteção de Malware Metamórfico Estima-se que mais de 66% do malware em circulação sejam variantes ofuscadas de malware conhecido. Além disso, variantes polimórficas têm vindo a tornar-se cada vez mais predominantes, aumentando de 50% para 96% das ameaças bloqueadas pelo Windows Defender entre 2010 e 2017. Entre os malwares ofuscados, o malware metamórfico apresenta os maiores desafios devido à sua capacidade de alterar a sua estrutura interna na sua integridade enquanto mantém o mesmo comportamento. Métodos de deteção baseados em assinaturas, que dependem dessa estrutura, tornam-se

ineficazes contra variantes metamórficas. Já a deteção baseada em comportamento avalia o comportamento do programa de forma independente da sua estrutura interna. Contudo, essa abordagem apresenta limitações práticas significativas. Entre elas, destaca-se o elevado overhead associado à execução do programa para análise. Além disso, muitos dos métodos propostos baseiam-se quase exclusivamente em chamadas de API, que podem ser manipuladas de modo a imitar o comportamento de software benigno, comprometendo a precisão e a eficácia na distinção entre malware e software legítimo.

Contribuições A metodologia proposta faz uso de decoder-based transformers para identificar padrões em software benigno por meio da análise estática dos binários. Esta abordagem supera limitações como a *Curse of dimensionality* associada à extração extensiva de features e elimina o overhead das abordagens comportamentais. O objetivo é criar um modelo robusto que possa detetar qualquer malware, incluindo variantes polimórficas e metamórficas, bem como malware zero day. A abordagem busca também ser computacionalmente eficiente e escalável, viabilizando o seu desenvolvimento e implementação como software antivírus, tanto para grandes organizações quanto para endpoints de utilizadores, com capacidade de deteção em tempo real. Além disso, pretende-se que a metodologia seja adaptável a diversos cenários, incluindo dispositivos com recursos computacionais limitados e situações onde as informações sobre malware sejam restritas.

Para atingir esses objetivos, o modelo é treinado para identificar os padrões do software benigno. Pressupõe-se que o malware exiba padrões distintos devido à presença de instruções maliciosas ou sinais de ofuscação. Assim, um transformer, após aprender os padrões do software benigno, deve ser capaz de identificar e classificar corretamente o malware, baseando-se nas discrepâncias entre os padrões aprendidos e os apresentados por amostras desconhecidas. Para determinar se uma amostra é benigna ou maligna, o modelo calcula a probabilidade de cada byte ocorrer no contexto dos anteriores. Essas probabilidades são usadas para gerar o Aggregated Cumulative Deviation (ACD), que é uma medida do quanto a amostra se desvia dos padrões do software benigno. Este valor é comparado a um valor threshold e, se fôr maior, a amostra é classificada como malware. Caso a premissa seja válida, esta abordagem apresenta potencial para identificar qualquer tipo de malware, incluindo malware zero-day e variantes polimórficas e metamórficas de malware conhecido.

Esta abordagem também resolve o problema de cénarios em que a informação sobre ameaças é escassa, um dos principais desafios da área tendo em conta a falta de datasets curados para inúmeros tipos de malware. Ao alimentar diretamente os executáveis não processados ao modelo, também se dispensa a informação de compilação necessária para a engenharia reversa, evita-se os problemas de dimensionalidade associados à extração de features, e não se incorre no overhead de uma abordagem baseada em comportamento. A possibilidade de aplicação da metodologia para deteção de malware em tempo real não foi verificada mas não se encontra excluída, embora haja uma preocupação com dispositivos de computação single-core e de baixa conectividade. Uma segunda contribuição complementar deste trabalho é fornecer alguma perspetiva sobre a possibilidade de utilizar as probabilidades emitidas pelo modelo para a extração de assinaturas de malware para classificação multi-família.

Dados e Testes Realizados O modelo foi treinado exclusivamente com software benigno, e várias configurações de hiperparâmetros foram testadas de modo a obter os menores valores de loss possíveis dentro

das restrições computacionais deste trabalho. A metodologia foi validada na classificação de software benigno e malware. Para estas duas fases, o malware foi proveniente do DikeDataset e o software benigno foi obtido tanto do DikeDataset quanto do sistema de ficheiros local. Os testes foram realizados com vários datasets, de modo a avaliar a eficácia desta classificação, nomeadamente: um dataset de programas benignos obtidos localmente, um subset do dataset de malware metamórfico Malicia, e um dataset de ficheiros .DLL também obtidos localmente. Todas as amostras utilizadas consistiram em programas Windows.

Conclusão e Trabalhos Futuros Os testes realizados apresentam resultados promissores na distinção entre software benigno e malware, incluindo o evasivo malware metamórfico, com precisão e recall satisfatórios para uma porção significativa das amostras. Para trabalhos futuros sugere-se treinar o modelo com um conjunto de dados mais largo e variado e otimizar os parâmetros de modo a ser possível detetar qualquer tipo de malware, independentemente da plataforma e compilador. Quanto à presença de assinaturas de malware nos datasets usados, foi observado que as probabilidades emitidas pelo modelo apresentavam padrões comuns para a maior parte das famílias. Com base nestes resultados, sugere-se explorar a possibilidade de utilização do modelo para classificação multi-família ou o uso destas observações para refinamento da metodologia de classificação binária. Para além disso, sugerem-se ajustes relativos à definição de thresholds e à partição dos dados de teste.

Palavras-chave: Malware metamórfico, deteção de malware, deteção de anomalias, aprendizagem automática, transformers

Abstract

Malware has been growing rapidly in both volume and sophistication, with increasingly complex obfuscation techniques used to create variants that evade detection. Beyond zero-day threats, metamorphic malware poses a particularly significant challenge by mutating its entire code while preserving its behavior, achieving an exceptional level of evasiveness. Metamorphic malware leverages these techniques by mutating the entire code and maintaining only program behavior, thus achieving the highest level of evasiveness. Traditional signature-based detection methods employed by commercial antivirus tools are inadequate against such threats. This has led to a surge in research proposing machine learning techniques for improved detection. However, a robust methodology capable of accurately detecting and classifying all malware types in the wild, supported by rigorous testing on varied datasets, remains elusive. Furthermore, ensuring computational efficiency is critical to enable real-time malware detection in practical scenarios.

Deep learning models based on the transformer architecture have gained significant traction due to their power and effectiveness in language modeling tasks. The aim of this work is to develop a methodology capable of effectively and efficiently detecting all malware, with a simple approach applicable to a wide range of scenarios. For this purpose, transformers are leveraged to learn patterns in benign software, creating a robust anomaly-based detector that relies on the static analysis of unprocessed binaries. The approach demonstrates promising results in detecting metamorphic malware, achieving reasonable precision and recall in experimental testing. While further optimization is required to enhance generalization, accuracy, and scalability, these findings mark a significant advance in malware detection methodologies.

Keywords: Metamorphic malware, malware detection, anomaly detection, machine learning, transformers

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Malware, short for malicious software, refers to ”programs covertly inserted into a system with the intent to compromise the confidentiality, integrity, or availability of the victim’s data, applications, or operating system, or to otherwise disrupt or harm the victim [2]. It serves as a critical tool in executing cyberattacks, often being deployed in one or more stages of the attack cycle. Over the years, both the prevalence and sophistication of malware have increased significantly [3], making it an ever-growing threat to individuals, businesses, and organizations. Malware distributors often target companies, notorious individuals and governmental organizations. Malware is also used in state-sponsored Advanced Persistent Threats (APTs) to target political organizations, defense firms and national critical infrastructures [4], serving as a tool for waging war, as recent events have corroborated [5].

The number and sophistication of malware have steadily increased each year. According to a recent threat report [6], the first half of 2024 recorded an increase of 30% in malware over 2023. Significant spikes were observed in IoT malware (+107%) and encrypted threats (+92%), demonstrating increasing efficiency and sophistication among cybercriminals. In fact, 92% of the increase was in May alone, with the exploitation of the TP-Link command injection vulnerability, allowing the spread of Mirai malware, which hijacks IoT devices in order to form botnets to leverage in Distributed Denial of Service (DDoS) attacks. The report also declares 78,923 new malware variants encountered in the first half of 2024, averaging 526 new variants a day.

The increasing sophistication of cyber attackers, exemplified by the rising complexity and volume of malware, contributes significantly to the escalating costs of cybercrime. As attackers deploy more advanced methods and techniques, the scale and impact of these attacks grow substantially. As a result, losses from cybercrime in the U.S. exceeded $12.5 billion in 2023, marking a 22% increase over 2022 [7]. IBM [8] estimates that the cost of a data breach is now $4.88 million USD, representing a 10% increase over 2023. Cybercrime thus poses a significant threat to a business’s livelihood, and not only large companies are targets. According to recent reports, 43% of cyber attacks target small businesses, of which 60% of victims go out of business within six months [9].

1.2 Problem and Motivation

Malware is a versatile tool employed at multiple stages of a cyberattack and functions as a critical Indicator of Attack (IoA). When detected promptly, it can play a key role in identifying and neutralizing threats before a breach occurs. However, the effectiveness of cybersecurity systems hinges on the collection and processing of vast amounts of data, making automation a fundamental necessity. Consequently, the future of cybersecurity is increasingly reliant on the efficiency of artificial intelligence (AI) tools to effectively manage and mitigate evolving cyber threats.

According to IBM [8], breaches in organizations with fully deployed security AI and automation cost $1.88 million USD less than those without, which amounts to a 33.2% difference in average breach cost. Moreover, these companies experienced, on average, a nearly 100-day shorter time to identify and contain a breach compared to those without, equating to 209 days versus 307 days. This improvement likely contributed to the increase in the adoption of security AI and automation, which rose from 28% in 2023 to 31% in 2024 [8].

For some time, commercial antivirus tools have provided a certain level of protection against malware infections. However, traditional signature-based tools are limited in their ability to detect new and complex obfuscated variants of known malware [3]. While dynamic analysis offers more robust detection capabilities, it comes with significant computational overhead. Moreover, it is not always effective in capturing the behavior of all known malware variants in the wild [10].

Recent literature proposes a variety of new approaches using data mining and machine learning for malware detection [3, 11]. However, the existing methods either lack the power to detect zero-day vulnerabilities and highly sophisticated concealed malware [12, 3], or they face challenges in scalability, reproducibility, generalization capabilities, and lack testing on real-world representative datasets [12].

Upon reviewing the literature, many studies yield promising results; however, these are often not backed by sufficient testing. The datasets used are typically small and lack diversity, with only one or two being employed in most cases. This limitation is likely due to the challenge of obtaining well-curated and representative malware datasets, a difficulty also encountered in this study. As a result, the generalization ability of existing methods remains uncertain, particularly when it comes to detecting new and obfuscated malware. Therefore, developing a comprehensive approach capable of detecting all variants of malware—one that is rigorously tested across diverse, real-world datasets—would be of significant importance.

In the absence of suitable datasets, an anomaly detection approach could be effective. Given that malware typically exhibits substantial differences from benign software, due to malicious or obfuscated instructions, such an approach can ensure that new or unseen malware is always flagged as a potential threat.

Several studies have utilized deep learning techniques to address this issue. Among these, Deep Neural Networks (DNNs), Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and transformers are common approaches [13], which have produced relevant results. Notably, Long Short-Term Memory (LSTM) networks and bidirectional transformers, such as BERT (a transformer encoder developed by Google), have shown significant promise by achieving state-of-the-art results in malware detection.

However, limited research has been conducted on the use of transformers with unidirectional decoding for malware detection, an architecture introduced by Vaswani et al. [1] in their foundational work on the Transformer model, the architecture behind the widely known GPT series. The potential of this unidirectional decoding approach for detecting malware, particularly metamorphic malware, remains largely unexplored. Consequently, this study aims to investigate the application of unidirectional decoder-based transformers for detecting metamorphic malware.

1.3 Metamorphic Malware Detection

It is estimated that over 66% of malware are obfuscated variations of previously known threats [14]. Polymorphic variants are becoming increasingly common, having increased from 50% to 96% of detected and blocked threats by Windows Defender from 2010 to 2017 [15]. Metamorphic malware, which modifies its internal structure while maintaining the same behavior, presents the greatest challenges among obfuscated malware.

Signature-based detection, which relies on recognizing the structure of malware, is generally insufficient for identifying metamorphic variants [14]. Behavior-based detection, on the other hand, analyzes program behavior regardless of structure and could offer a more effective solution. However, this approach requires executing the program, which introduces significant overhead. Furthermore, many existing methods focus exclusively on API calls, which may not always differentiate between malware and benign software, as both can use the same API calls. Additionally, API calls can be deliberately manipulated to make malware mimic benign software, achieving up to 93% similarity and evading detection. Malicious programs can also replace suspicious API calls with manually coded routines [16].

Opcode-based detection methods that rely on operand frequencies are also disadvantaged by obfuscation techniques, such as dead code insertion and equivalent code substitution, which alter operand statistics in order to make the malware appear more similar to benign software [17, 18]. Methods based on sequence patterns or structure often face challenges with high dimensionality, computational overhead, and scalability, which limit their efficiency and applicability to large-scale malware detection [17, 19].

Although several approaches, including those leveraging deep learning, have been proposed and show promise, their ability to generalize across diverse malware variants remains unproven [12, 3]. Therefore, an effective and efficient solution capable of detecting all variants of malware is still required.

1.4 Contributions

The objective of this work is to propose a novel methodology capable of detecting a wide range of malware, including polymorphic and metamorphic variants, as well as zero-day threats. This approach is designed to be both time-efficient and scalable, with the potential to be developed into anti-virus (AV) software suitable for deployment across large enterprises and individual endpoints, enabling real-time malware detection. Additionally, the methodology aims to be effective in scenarios where limited information about malware threats is available.

To achieve these objectives, the proposed method involves training a model exclusively on benign software. By learning the typical patterns found in benign software, the model should be able to detect

significant deviations that may indicate the presence of malware. The model predicts the probability of each byte occurring based on the context provided by the preceding bytes in benign software. From this, the Aggregated Cumulative Deviation (ACD) is computed. The ACD value is then compared to a threshold to differentiate between benign and malicious software.

Assuming that malware exhibits distinct code patterns compared to benign software, due to the presence of malicious or obfuscated instructions, the model should be capable of accurately labeling malware samples. If this assumption holds, anomaly-based detection could potentially detect all types of malware, including unknown variants and zero-day threats. This study explores the use of a unidirectional decoder-only transformer for recognizing these distinct patterns.

This approach also addresses the challenge of scenarios where curated information about malware threats is either unavailable or difficult to obtain, such as the scarcity of labeled datasets for certain malware families. By directly feeding unprocessed executable files into the model, it eliminates the need for compilation information, which is required for reverse engineering. This approach avoids the dimensionality problems associated with feature space, and does not incur the overhead of behavior-based detection methods, which require program execution and monitoring. Its applicability for real-time detection has not been fully verified; however, it has not been ruled out, although concerns remain regarding its use in devices with single-core processors and/or low connectivity.

A second complementary contribution of this work is offering insights into the potential application of this methodology for extracting malware signatures in the context of multi-family classification.

1.5 Organization

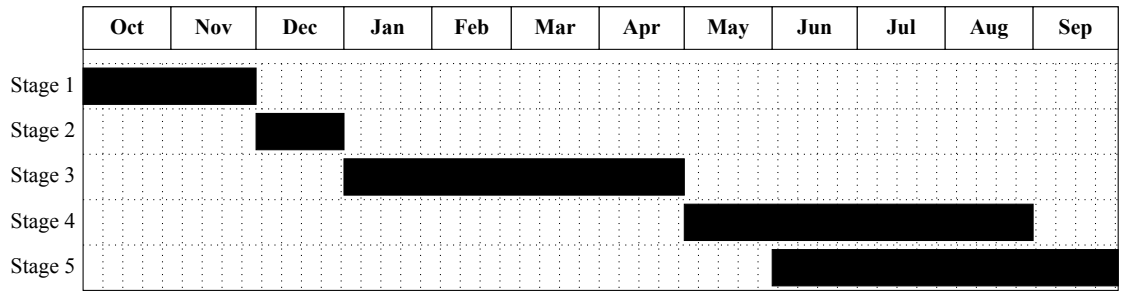


Figure 1.1: Gantt chart with the planned work.

Table 1.1: Description of each stage of the planned work.

Stage	*Description*
Stage 1	Data collection, problem familiarization and practicing basic algorithms
Stage 2	Collection and analysis of preliminary results
Stage 3	Refinement of algorithms
Stage 4	Analysis of results and comparison with the state of the art
Stage 5	Writing report

Figure 1.1 presents the timeline of the work initially planned for each stage of the project, and Table 1.1 the description of each stage. The initial stages of familiarization with the problem and the analysis

of preliminary results were concluded in a relatively timely manner, with only a slight delay of about half a month. The most relevant problems encountered at this point were accruing datasets representative of the problem and understanding the current state of the art. Despite the existence of malware databases, the listed malware lacked detailed categorization, such as the malware family and whether they were polymorphic or metamorphic variants, which delayed the acquisition of relevant datasets. The second issue arose from the works surveyed, which also appeared to suffer from similar difficulties, as they used datasets lacking in size and variety. This raised doubts about the generalizability of their results, which, in turn, led to questions on whether the problem had been effectively solved and whether the most important avenues had already been sufficiently explored.

A more critical delay occurred from the end of March to November, during which personal circumstances hindered progress on this work, resulting in a one-year postponement of its delivery. To accommodate this, new milestones were established for stages 3–5. Stage 3, which was left incomplete, was rescheduled for completion around January. Stage 4 was planned to run from February to June, while the final stage, focused on writing the report, was slated for July through September, when it was due. Although this revised plan was largely adhered to, another delay occurred during the final stage, with the thesis ultimately being delivered at the end of November.

1.6 Structure of the document

The remainder of this document is organized as follows:

- Chapter 2 - Background and Related Work: Provides key concepts in malware detection, an overview of detection techniques, and surveys recent studies using relevant technologies.
- Chapter 4 - Methods: Describes the approach taken to address the problem.
- Chapter 5 - Data: Describes the datasets used in this work.
- Chapter 6 - Results: Presents the results obtained from the experiments.
- Chapter 7 - Conclusions: Summarizes the conclusions drawn from this work and outlines directions for future research.

Chapter 2

Background and Related Work

This chapter is divided into two sections. The first section, Background, introduces foundational concepts related to obfuscation, discussing various types of obfuscated malware and their implementation techniques. It also provides an overview of malware detection methods, emphasizing the evolution from traditional signature-based methods to modern behavioral, heuristic, and machine learning approaches, and concludes with a detailed explanation of the transformer architecture, the deep learning framework employed in this study.

The second section, Related Work, reviews recent studies on malware detection, covering deep learning architectures such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). Special emphasis is placed on transformers, a type of deep learning architecture, with a more detailed exploration of studies utilizing specific transformer models such as BERT and GPT.

2.1 Background

This section introduces key concepts related to obfuscation, which will be used in subsequent chapters. It describes various types of obfuscated malware and the techniques employed to implement them. Additionally, an overview of different malware detection methods is provided. The section concludes with an in-depth discussion of the transformer architecture, the deep learning framework utilized in this study.

2.1.1 Malware Obfuscation

Obfuscation techniques are strategies used to deliberately obscure the structure or logic of software code, making it difficult to analyze, reverse-engineer, or understand. These techniques are commonly employed in two primary contexts. In malicious software, obfuscation is used to evade detection by anti-virus systems or hinder forensic analysis. In benign software, it serves to protect intellectual property by obscuring implementation details of certain parts of the system, and to enforce Digital Rights Management (DRM) by safeguarding crucial pieces of information such as encryption keys and communication protocols [20, 21].

Malware can use several concealment approaches to avoid detection. According to Aslan and Samet [3], the most common obfuscation approaches are the following:

- **Encryption:** Encrypted malware consists of two main components: a decrypter routine and an

encrypted payload. When the malware is executed, the decrypter decodes the encrypted main body, revealing the malicious code. The encryption key is changed with each new infection, making it more difficult to detect its signature. However, the decrypter itself often remains the same and may be simpler to detect since it doesn't change across different instances of the malware [22].

- **Oligomorphism:** An initial attempt at creating malware capable of dynamically changing its decrypter. However, this approach was limited to producing only a few hundred variants, which still allowed for the identification of the decrypter's signature, making detection easier[22].
- **Polymorphism:** Polymorphic malware overcomes the limitation of static decryption signatures by employing code obfuscation techniques that generate a vast number of variants for the decrypter. Mutation engines automate this process, enabling malware authors to quickly generate polymorphic versions of non-obfuscated malware, complicating detection efforts. However, despite the frequent changes to the decryption routine, the core malicious payload, revealed after decryption, typically remains constant. This constancy in the malicious code is a key vulnerability that anti-virus software can exploit for detection, as it provides a stable pattern that can be identified even if the obfuscated decrypter changes with each infection cycle [22].
- **Metamorphism:** Metamorphic malware differs from polymorphic malware in that it does not rely on encryption to hide its payload. Instead, it uses obfuscation techniques to mutate its entire code structure with each infection, making the malware's appearance highly variable. This allows metamorphic malware to change significantly with every execution, while still retaining its malicious functionality. Because of these changes, traditional signature-based detection, which relies on static code patterns, becomes ineffective. The challenge for anti-virus software is that there is no consistent decryption routine or even structural similarity in the code, making the malware harder to detect through conventional methods [22].
- **Stealth:** A variety of techniques used by malware to evade detection and analysis [3]. These techniques include the use of rootkits, which conceal large files from the system and make detection more difficult [23]. Other tactics involve targeting mechanisms that limit the malware's execution or spread to specific systems, thus reducing its exposure to detection methods [24]. Additionally, anti-emulation strategies are used to alter the malware's behavior when running in virtualized environments, such as sandboxes or emulators, further complicating the analysis process [24].
- **Packaging:** Software compression or encryption is commonly used to conceal malware, making detection more difficult by altering its appearance. When the malware code is compressed or encrypted, even a small change in the original code can cause a significant alteration in the compressed or encrypted file's structure. This technique makes signature-based detection methods less effective, as the compressed or encrypted file will differ with each infection [23].

In order to generate mutations in the code, polymorphic and metamorphic malware use obfuscation techniques. According to You and Yim [22], the most commonly used techniques are:

- **Dead-Code Insertion:** For example, the insertion of no operation (`nop`) instructions. It can be easily bypassed by signature-based malware detectors, which are often designed to eliminate or

ignore such instructions before performing analysis. To counter this, more complex inefficient code sequences are used.

- **Register Reassignment:** Another simple obfuscation technique where the registers used in the code are replaced with different ones. This creates additional complexity, making the malware harder to analyze.
- **Subroutine Reordering:** A technique that obfuscates the code by randomly changing the order of subroutines to confuse analysis tools and obfuscate the program's behavior.
- **Instruction Substitution:** A technique that replaces some instructions with equivalent ones. For example, replacing `xor` with `sub` and `mov` with `push`/`pop`. These substitutions are drawn from a library of equivalent instructions, making the malware harder to analyze and detect using static analysis techniques.
- **Code Transposition:** This technique involves reordering the sequence of instructions without changing the program's behavior. One approach achieves this by randomly reordering the instructions and using branches or jumps to ensure the correct execution order. However, this method can be easily reverted by removing unnecessary branches or jumps. A more complex method involves identifying independent instructions and shuffling them. This method is harder to implement but more effective at maintaining functionality while complicating the analysis.
- **Code Integration:** A complex technique that inserts the malicious code into a target program. The target is first split into manageable sections, then the malware is inserted and the program is reassembled.

These obfuscation techniques are facilitated by metamorphic malware construction kits, which are widely available tools used by malware authors to introduce code-morphing capabilities into their malicious software. The most popular in literature are PSMPC (Phalcon-Skism Mass-Produced Code Generator), NGVCK (Next Generation Virus Creation Kit), G2 (Second Generation Virus Generator) and MWOR (Metamorphic Worm) [17]. Most of the available kits are designed to build malware for Windows. These morphing engines are available at VX Heavens website[1] and are an important tool to generate polymorphic and metamorphic variants for testing AV software.

2.1.2 Malware Detection Approaches

Academic studies on novel malware detection techniques have been growing in number in recent years. Signature-based detection is a widely used traditional approach, but due to its limitations in detecting zero-day threats and new variants of existing malware, there has been a growing shift toward behavioral, heuristic, and model-checking detection methods.

Studies applying data mining and machine learning techniques have become increasingly popular in the latter approaches. However, despite significant progress, no method has been proven to detect all malware variants in a time-efficient manner. This is especially true for more sophisticated or unknown malware, which remains an open challenge in the field [3].

[1] https://1vx.ug/archive/VxHeaven/

This section provides an overview of the most common approaches currently used in malware detection, along with representative examples of research conducted in each.

Signature-Based Malware Detection

Signature-based malware detection is a fast and efficient method widely used in commercial antivirus tools for identifying known malware. This technique works by extracting features from a known malware executable, creating a unique signature, and storing it in a database. When a new sample is encountered, its signature is generated using the same feature extraction method and compared against the stored signatures in the database. Some techniques used to extract features are string scanning, top-and-tail scanning, entry point scanning and integrity checking [3].

String scanning is a malware detection technique that identifies byte sequences, or ”signatures,” unique to specific malware. These sequences are not expected to appear in benign software, making them useful for identifying known threats [25]. When signatures are designed to detect multiple variants within the same family of malware, they are referred to as generic detection signatures [26]. This method has been widely adopted by antivirus software to detect malware [3]. However, the scanning process can be optimized with various techniques to increase its efficiency [25].

Top-and-tail and entry point scanning are examples of optimization techniques that also rely on byte signatures. However, rather than scanning the entire file, these methods focus on specific areas: Top-and-tail scanning examines the beginning and end points of files, while entry point scanning targets the program's entry points. These strategies aim to speed up the detection process by limiting the areas being analyzed [3].

The challenge of generating unique signatures with minimal error rates for unknown malware is time-consuming and often requires significant manual effort [27]. To address this, machine learning and data mining techniques have been employed to extract features that are more resilient to mutating variants, reducing the inefficiencies of manually generated signatures. One of the pioneers in this area, Newsome et al. [28], proposed extracting polymorphic worm signatures using three distinct approaches, namely one based on the Naive Bayes classifier and another using clustering combined with sequence alignment techniques, commonly used in bioinformatics to detect patterns in genetic data. These methods were integrated into the Polygraph suite, a set of algorithms designed for robust signature generation.

Tang et al. [29] also focus on detecting polymorphic worms by proposing a method for generating exploit-based signatures, which are derived from the specific exploit code used by the malware. This approach also employs sequence alignment techniques, aiming to improve accuracy by increasing noise tolerance, thereby offering more reliable signatures compared to prior methods. Similarly, Naidu et al. [30] used sequence alignment methods to create signatures for polymorphic malware, exploring how variations in alignment parameters—such as gap penalties—could improve the detection of unknown variants.

More recently, Kakisim et al. [17] proposed the extraction of engine-specific signatures from co-opcode graphs, and Du et al. [31] presented the use of combination rule mining for malware signature generation. In a related approach, deriving word embeddings from opcode sequences has been shown to produce more robust and generalizable features, which can further enhance the effectiveness of malware detection [32, 33].

On the other hand, Hidden Markov Models (HMMs) offer a dynamic, statistical-based approach to malware detection. Unlike traditional signature-based methods that rely on static byte patterns, HMMs focus on the probabilistic transitions between opcodes in the binary code. By analyzing these opcode sequences, HMMs generate behavioral profiles of malware that capture their typical patterns and structures. Several works have explored the utility of Hidden Markov Models (HMM) for the detection of metamorphic malware.

Wong and Stamp [34] were the first to propose it by developing a detector which succeeds in identifying metamorphic viruses created using the Next Generation Virus Creation Kit (NGVCK), which were not able to be detected by commercial anti-virus tools at the time. A sequence of opcodes is extracted from the disassembled executables of malware variants and concatenated into a long observation sequence. The HMM is then trained with multiple sequences, with the resulting model representing an average behavior for all the sequences—essentially a statistical profile for the malware family. After training, the resulting HMM is used to compute the log-likelihood for each sample in the test set. The results demonstrate high accuracy in detecting metamorphic variants generated with NGVCK.

Building on this approach, Austin et al. [35] proposed a dueling HMM strategy that involves training N HMMs for benign code, each representing code compiled by different compilers, and M HMMs for malicious code, each representing different malware families. The method then compares the probability of observing the opcode sequence for each new sample across all N + M HMMs. This method was proposed to improve the accuracy of malware detection; however, it incurs significant overhead.

In the same year, Toderici and Stamp [36] concluded that HMM-based detection is insufficient for identifying metamorphic malware, particularly when it is carefully crafted with benign code integration. To address this, they proposed a hybrid strategy that combines machine learning (specifically HMMs) with statistical analysis using the chi-squared test. Their results indicated that this hybrid approach outperforms the individual methods. However, they noted that the hybrid strategy still struggles to identify malware containing contiguous blocks of benign code, as opposed to malware with benign code inserted in smaller, discrete blocks

In a more recent work, Ling et al. [37] suggest an approach for the extraction of features by computing a stream of byte chunks using several metrics. The features are extracted based on the approach proposed by Menéndez et al. [38] of analyzing the entropy of binary strings without disassembling them. Nonnegative Matrix Factorization (NMF) is then used for dimensionality reduction, and K-Means is used to group the observed data into a small set of symbols. The reduced feature space is then used to train an HMM for each malware family, enabling both binary and multifamily classification.

Behaviour-based Malware Detection

This behavior-based approach monitors a program during runtime, typically within a sandboxed environment, capturing activities such as system calls, file changes, created processes, and network interactions. These behavioral features are then extracted and used for classification via machine learning algorithms [3]. It proves to be effective in identifying new and obfuscated malware variants, as their core behavior remains unchanged, even when the code is altered through obfuscation techniques [3, 39]. However, advanced obfuscation techniques can interfere with analysis by enabling malware to detect when it's in a sandboxed environment, causing it to alter its behavior and appear benign instead of malicious.

The difficulties with behavior-based malware detection include defining relevant behaviors, handling the large volume of features extracted, and accurately identifying similarities between features. As a result, an effective model for this approach has yet to be developed [3]. Additionally, this method requires more scanning time compared to traditional signature-based detection, making it less efficient in real-time applications [27].

To address the challenges in behavior-based malware detection, researchers have applied various techniques for feature reduction and classification. Hegedus et al. [40] used random projections for dimensionality reduction and employed a k-nearest neighbors classifier for malware detection. Mohaisen et al. [41] experimented with different classifiers, including SVM, decision trees, and logistic regression, finding that the dual SVM yielded promising results with relatively low overhead. Borojerdi and Abadi [42] proposed sequence clustering and alignment methods for detecting polymorphic malware, with the authors suggesting that the extracted patterns could be use to detect all polymorphic malware.

On a different approach, Alqurashi and Batarfi [43] proposed a hybrid method combining K-means clustering with Hidden Markov Models (HMM), where genetic operators enhance the K-means algorithm. The method clusters program behaviors based on HMM scoring and compares the performance of genetic K-means with standard K-means. Another experiment evaluates the effectiveness of HMMs trained on statically extracted opcode sequences versus dynamically extracted API calls. The study suggests that the optimized genetic K-means improves accuracy and that API calls are more effective for malware detection than opcode sequences. More recently, Maniriho et al. [44] used neural networks with word embeddings of dynamically extracted API calls for malware detection.

Heuristic-based Malware Detection

Heuristic-based malware detection is a more recent strategy designed to address the limitations of signature and behavior-based approaches. This complex method uses machine learning and data mining to model the behavior of executable files [27], relying on initial expert-defined rules and patterns [45]. While it can detect many known and unknown malware variants, it struggles with complex obfuscated malware and tends to produce a higher false positive rate [3]. Moreover, the construction of these rules is time-consuming and prone to human error [45], limiting the approach's overall effectiveness and reliability.

Komashinskiy and Kotenko [46] used file format specifics to create "Position-Byte on position" pairs as features for malware detection. Several classifiers, including decision tables, C4.5, random forests, and Naive Bayes, were tested, with random forests yielding the best performance. Islam et al. [47] integrated static and dynamic features in their approach, testing classifiers like SVMs, decision trees, random forests, and instance-based learning, along with boosting techniques. Once again, random forests showed the best results, but the authors noted the need to include older malware samples for accurate detection of more sophisticated variants.

Machine Learning-Based Malware Detection

Machine learning plays an increasingly important role in malware detection due to its ability to analyze large volumes of data, identify patterns, and generalize across previously unseen malware samples. This

makes it particularly useful in addressing the evolving threat landscape of zero-day attacks and obfuscated malware [45, 3, 19].

Malware detection using machine learning techniques typically involves two main steps: feature extraction and classification/clustering. In the feature extraction step, one or more features are extracted statically and/or dynamically from the sample using a variety of criteria. In the classification/clustering step, machine learning algorithms are employed to categorize the samples into distinct classes or groups based on their extracted features [45, 19]. To enhance efficiency and accuracy, feature selection and dimensionality reduction methods are also used to reduce the feature space [45, 19, 48].

Deep-learning-based malware detection is a relatively recent approach that has been proposed for large-scale malware classification [3]. This technique is particularly effective because it captures high-level feature representations within the hidden layers of a model in a scalable manner [49]. This is the approach used in this work. However, a key limitation is its susceptibility to adversarial attacks, where crafted inputs deceive the model, resulting in misclassifications [3]. Additionally, training deep-learning models can be time-consuming due to the complexity of training the hidden layers.

Other Approaches

Model checking is a technique used to validate programs by comparing their behavior against predefined specifications. In the context of malware detection, this method involves modeling malware behavior using linear temporal logic (LTL) and then comparing the behavior of samples against this model. While this approach can identify some new types of malware, it struggles with complex obfuscation and introduces significant computational overhead, making it less efficient for real-time or large-scale detection [3].

Cloud-based detection is a popular recent approach used by AV vendors to supplement signature-based methods. When files being analyzed don't match local signatures, they are sent to the cloud for classification [45]. However, this method raises privacy concerns since file information is transmitted to external servers. Additionally, cloud communication adds overhead to the scanning process, which can hinder real-time detection, especially with slower internet connections [3].

Mobile and IoT devices-based malware detection strategies have also been growing in popularity, as these devices are becoming a more visible target for the malware industry. In these areas, it is common to use both static and dynamic features, and the proposed systems frequently use machine learning algorithms. These approaches seem efficient in locating traditional malware, however they are lacking in the detection of novel obfuscated malware. These areas are still new, and more studies are needed, especially in the area of IoT [3].

2.1.3 Transformers

This section explains the transformer architecture, the main technology used in this work, as well as examples of implementations and limitations.

The Transformer

The Transformer architecture for neural networks was first proposed in Vaswani et al. [1], designed to capture global dependencies between input and output solely through self-attention. Previous sequence learning methods typically relied on recurrence, convolution, or a combination of recurrence and attention [50].

Attention is a mechanism that models dependencies regardless of their distance in input or output sequences. It does so by mapping a query and a set of key-value pairs to an output, with the output computed as the weighted sum of values, where weights reflect the compatibility between the query and keys. This mechanism had previously been combined with recurrent neural networks to enhance performance on complex tasks [1].

Self-attention, or intra-attention, is an attention mechanism that relates different positions within the same sequence to compute a sequence representation [1]. It enables the network to focus on distinct parts of an input sequence while predicting. First applied to improve the memory capacity of LSTM recurrent networks by Cheng et al. [51], self-attention has since proven successful in many tasks when used alongside recurrent networks [1].

In the Transformer architecture, relying solely on self-attention posed challenges in maintaining representational richness for complex dependencies. To address this, Vaswani et al. [1] introduced multi-head self-attention, enabling the model to simultaneously attend to various aspects of the input sequence and combine their representations, thereby improving resolution and prediction accuracy.

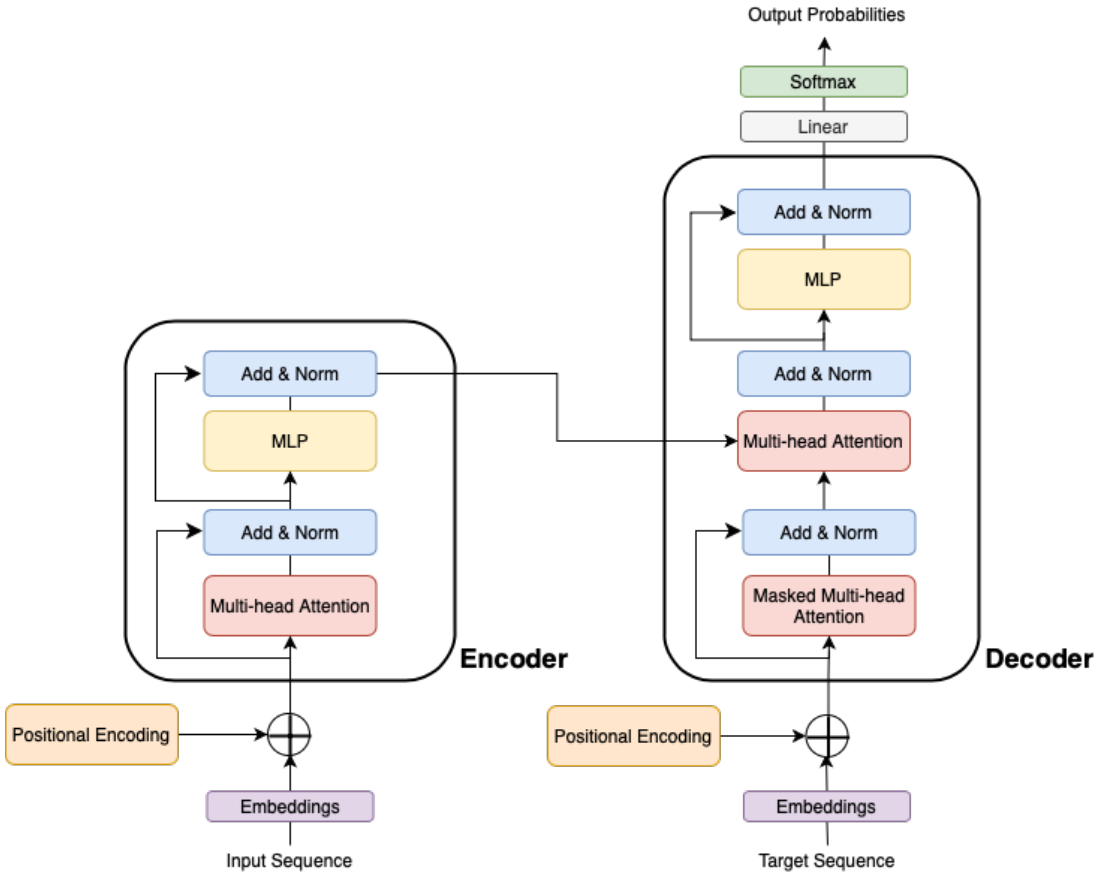


Figure 2.1: Transformer Architecture (based on [1]).

As can be observed in 2.1, the Transformer architecture consists of an encoder and a decoder, each containing several layers [1]. The encoder transforms an input sequence into a fixed-length representation, and the decoder generates an output sequence from it. Both components use two sub-layers: multi-head self-attention and a position-wise fully connected feed-forward network. The position-wise processing uses the same weights across all positions, enabling efficient handling of long sequences while maintaining accuracy in learning global dependencies.

Self-attention allows the model to focus on different input parts in parallel, and the feed-forward network applies non-linear transformations. Residual connections mitigate the vanishing gradient problem, improving stability. Positional encodings address the lack of recurrence or convolution by encoding the relative position of tokens [1].

The decoder adds a masked self-attention layer to prevent future tokens from being attended to during training. Both encoder and decoder stacks consist of identical layers, capturing increasingly complex patterns. The number of layers is a tunable hyperparameter. In the original Transformer paper, Vaswani et al. [1] used a stack of 6 layers for both the encoder and decoder, but other configurations have also been used in subsequent work [52, 53]. The Transformer achieves state-of-the-art results in machine translation, offering more parallelization and shorter training times compared to earlier models [1].

Decoder-Based Transformers

Radford et al. [54] introduced GPT (Generative Pre-trained Transformer), the first major application of the Transformer architecture [1] for language modeling. This model leverages a semi-supervised approach, consisting of two phases: generative pre-training, where the model learns from a large corpus of text data, and discriminative fine-tuning, which refines the model for specific tasks. GPT showed strong performance across various natural language processing tasks, highlighting the effectiveness of the Transformer in generative modeling.

GPT's architecture has evolved significantly since its initial release. The original GPT model had 12 layers and 12 attention heads, but with each subsequent version, the number of layers and heads has grown. The largest model, GPT-3, features 96 attention layers and 96 heads[2]. This increase in size enables the model to capture more intricate patterns in the input data, resulting in improved output quality. Larger models like GPT-3 have demonstrated superior performance in natural language generation tasks due to their scale.

Bidirectional Transformers

Bidirectional Encoder Representations from Transformers (BERT) is a family of transformer-based models introduced by Google in 2018, achieving state-of-the-art results in Natural Language Processing (NLP) tasks when compared to OpenAI's original GPT model [55]. Its key distinction from GPT is the implementation of bidirectional pre-training, a design choice that limits GPT's performance on certain tasks [56].

RoBERTa is an optimized version of BERT, utilizing different hyperparameters and a larger training dataset, resulting in enhanced performance [57, 58]. DistilBERT, on the other hand, is a 40% smaller

[2]https://lambdalabs.com/blog/demystifying-gpt-3

variant of BERT, maintaining most of its language understanding capabilities while training 60% faster [59].

Limitations

Transformers are a major advancement in deep learning, but they come with challenges, notably the need for large datasets to build robust models[3]. This can be problematic in situations where sufficient data is not available, such as with malware detection, where datasets are limited. To address this limitation, this work proposes a solution by training the model using benign software instead of relying on scarce malware datasets.

Training a transformer model is also very computationally demanding, both in terms of GPU and memory resources3. The feasibility of training very large transformers in a reasonable amount of time depends on the availability of advanced hardware. While model training is a one-time event, inference also requires significant resources, mainly due to the self-attention mechanism, which processes every part of the input data simultaneously. This may make it impractical for running on mobile or embedded devices. To deploy this methodology as AV software, a cloud-based approach could be necessary: endpoints would send queries to a server that evaluates the sample and returns the result. However, this approach would require the endpoints to have reliable and fast connectivity.

2.2 Related Work

This section reviews recent literature on malware detection with related technologies to the ones used in this work. The first section covers deep learning architectures, specifically Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). The later section examines the use of transformer models, such as BERT and GPT, in anomaly (including malware) detection.

2.2.1 Deep Learning

Convolutional Neural Networks (CNNs) are widely known for their ability to process visual data, but they have also been effectively applied to other domains, such as text and video processing. In the context of malware detection, CNNs have been extensively explored for identifying malicious patterns and classifying malware [60].

Gibert and Bejar [61] applied CNNs in two approaches to tackle the Microsoft Malware Classification Challenge (BIG 2015) dataset [62] for multi-class classification. The first approach, based on Nataraj et al. [63], involves visualizing malware binaries in grayscale and using standard image classification techniques, achieving 98.08% accuracy on a custom dataset. In their experiments, the best model achieved nearly 100% accuracy on the challenge dataset.

A second approach is based on Kim [64]. Instead of classifying sentences from the English language the proposed architecture was reused to classify malware samples using x86 instructions from disassembled executables. Two models were trained for this architecture, one without pre-trained word embeddings and another with pre-trained word embeddings using Word2Vec with the Skip-Gram model. The

[3] https://medium.com/p/28abcf7cee6b

CNN without pre-trained word embeddings obtained an accuracy of 99.52% and the one with pre-trained word embeddings of 99.47%. The authors suggest that better results with the latter might be obtained by using benign software as the training data due to several factors that introduce noise when training with malware that might lead to poorer embeddings.

A second approach is based on Kim [64], where the architecture designed for classifying English sentences was adapted for classifying malware samples using x86 instructions from disassembled executables. Two models were trained: one without pre-trained word embeddings and another with pre-trained Word2Vec embeddings using the Skip-Gram model. The model without embeddings achieved an accuracy of 99.52%, while the one with embeddings achieved 99.47%. The authors suggest that better results in the latter model could be obtained if benign software were used for training instead of malware, as malware data may introduce noise that affects the quality of the embeddings.

A similar approach was chosen by Xusheng and Yang [65], but instead of grayscale, an RGB color matrix was used to convert bytecode into image files. The authors worked with a dataset of Dalvik bytecode for Android and applied the model for binary classification. This method achieved a state-of-the-art detection time of 0.22 seconds, but with an accuracy of around 93%.

Recurrent Neural Networks (RNNs), particularly Long Short-Term Memory (LSTM) networks, are designed for sequence learning. Jha et al. [66] effectively utilized an LSTM for malware detection, testing it with three different feature vectors. The best performance was achieved using Word2Vec, an NLP embedding technique, applied to opcode sequences. The approach was evaluated on the Microsoft Malware Classification Challenge (BIG 2015) dataset, where it achieved an average accuracy of 91.75

Akhtar and Feng [67], on the other hand, propose a hybrid deep neural network approach that combines CNNs, which capture local spatial correlations, with LSTM networks to learn long-term sequences. They test this approach for malware detection on IoT devices using samples from Kaggle[4], achieving an accuracy of 99%. However, comparisons with decision tree (DT) and support vector machine (SVM) models, which also achieved high accuracy, suggest that the dataset may not be sufficiently varied or representative of real-world scenarios.

Xiao et al. [68] propose an approach using system call sequences, encoded as vectors, as features for training two LSTM models—one for malware and one for benign applications. Similarity scores are computed from these models to determine whether a sample is malicious or benign. When tested on an Android dataset, the approach achieved an accuracy of approximately 93.7%.

2.2.2 Transformer Models

Anomaly detection in cybersecurity requires AI tools to handle large volumes of system or network events. Lakha et al. [58] explore the use of BERT and RoBERTa for processing event metadata, generating robust features that are then fed into a Graph Neural Network (GNN) or anomaly detector. Their results demonstrate that combining these approaches significantly enhances anomaly detection performance on a well-known dataset.

Rahali and Akhloufi [69] propose using BERT for malware detection, marking the first use of a Transformer-based approach for this task. BERT is applied to both binary and multi-class classification using Android samples. The results show that BERT outperforms other state-of-the-art models, achieving

[4]https://www.kaggle.com/

an accuracy of 97.61% for binary classification and 91.02% for multi-class classification, surpassing both RoBERTa and DistilBERT in performance.

Alvares [32] use BERT for generating word embeddings instead of as a classifier. These embeddings capture contextual information of opcodes and are used to create training features for malware classification. When compared with embeddings generated using Word2Vec, BERT embeddings yield slightly higher accuracy across all classifiers. The best performance is achieved with a Random Forest classifier, obtaining 91.81% accuracy with BERT embeddings compared to 89.6% with Word2Vec embeddings.

Kale et al. [70] also compare several techniques for generating word embeddings for malware detection, including Word2Vec, HMM2Vec, BERT, and Embeddings from Language Models (ELMo). ELMo, an NLP model, generates embeddings using a trained bidirectional Long Short-Term Memory (biLSTM). The authors tested these features with various classifiers, finding that BERT achieved the highest accuracy at 96%, closely followed by ELMo and HMM2Vec, with accuracies ranging from 94% to 95%. In contrast, Word2Vec achieved only 90% accuracy. Given that BERT and ELMo have similar training times to Word2Vec, both are promising techniques for generating embeddings for malware detection.

Li et al. [71] introduce the I-MAD tool, which employs a ”galaxy” of transformer models to encode the semantic meaning of assembly code into vector representations. These are combined with vectors of other features and processed by an Interpretable Feed-Forward Neural Network (IFFNN) to yield interpretable detection results. This deep-learning model enhances static analysis by both detecting malware effectively and offering insights into its results. This interpretability aids malware analysts in identifying malware payloads or patterns in malware samples. The model achieves a detection accuracy of 91.5%.

The use of the GPT-2 model for malware detection has also been explored. Şahin [72] leverage a custom pre-trained model to understand the semantics of Assembly code from disassembled PE files. They then train a GPT-2 model for binary classification of malicious versus benign Assembly code, followed by fine-tuning using the pre-trained model to enhance performance. Their approach achieves an accuracy of 85.4% on a custom dataset, demonstrating that GPT-2 can effectively detect malware by identifying patterns within disassembled Assembly code.

Demirci et al. [73] expand on the findings of Şahin [72], comparing the GPT-2-based approach to other models for static malware detection. Their analysis reveals that while the GPT-2 approach achieves good accuracy, it falls short of state-of-the-art methods. The paper highlights the DLAM-BiLSTM model as the most competitive, achieving an accuracy of 98.4% on a custom dataset. This leads to the conclusion that unidirectional transformers like GPT-2 are not yet a mature methodology for malware detection, and additional research is needed to evaluate their potential.

The Vision Transformer (ViT) is a transformer-based architecture initially designed for computer vision but with origins in NLP models. It processes images by dividing them into patches, tokenizing these patches, and using attention mechanisms to learn semantic relationships between them. Seneviratne et al. [74] apply self-supervised learning, where the model generates its own supervisory signals from the data (e.g., predicting missing input parts), with the vision transformer architecture for Android malware detection. This approach achieves 97% accuracy and robust multi-family classification results, demonstrating the efficacy of self-supervised computer vision models in malware detection.

Park and Cho [75] also utilize a vision transformer, enhanced with the encoding of multiple patches to better capture the location of local features and their interrelationships. The model is trained and eval-

uated on the Microsoft Malware Classification Challenge (BIG 2015) dataset [62], achieving a detection accuracy of 96.83%. This marks a significant improvement over other image-based approaches, particularly in detecting three malware families that are frequently misclassified when using CNN-based methods.

All of these models were trained on malware datasets, focusing on identifying patterns from known malware families. However, due to the limited size and diversity of these datasets, there are concerns about their ability to generalize to unseen malware families. In contrast, the approach in this work, which relies exclusively on benign software, holds the potential for identifying previously unseen malware. Assuming malware inherently contains distinguishable patterns, such as compromising or obfuscated instructions, an anomaly-based detector could identify new or unknown variants by recognizing deviations from benign software behavior.

Chapter 3

Methods

The primary objective of this work is to develop a reliable methodology for distinguishing malware from benign software by identifying unique code patterns. This approach is based on the assumption that malware contains distinct sequences compared to benign software: non-obfuscated malware exhibits compromising instruction sequences and resource accesses, while obfuscated malware displays unnatural code patterns. If these patterns are sufficiently discernible, a transformer model—well-suited for detecting semantic patterns—should be able to effectively identify them.

3.1 Model Selection

To achieve the goal of this study, a decoder-based unidirectional transformer was chosen for experimentation. This decision was driven by the proven effectiveness of transformer architectures across a variety of domains, coupled with the relatively limited application of such models in malware detection research.

For this purpose, the open-source implementation of nanoGPT by Karpathy [76] was leveraged, as it allows for easy customization and training/finetuning of medium-sized GPT models. With sufficient computational resources and a sufficiently large corpus, performance levels comparable to GPT-2 can be reached by such models.

Several model configurations were tested by experimenting with different hyperparameters, as described in Section 5.2.1. Unlike Şahin [72], the closest related work, a single transformer model was trained directly on the benign executables without first disassembling them. This decision stemmed from the inherent non-linearity of the disassembly process.

Disassembly attempts to reverse the compilation process but is limited by the lossy nature of compiling, which strips away much of the source code's structural, symbolic, and high-level information. This issue is particularly critical in the context of malware samples, where both the compiler and the instruction set used to generate them are unknown, further complicating accurate reconstruction of the original assembly code.

Each compiler has unique optimization patterns, instruction choices, and calling conventions; without knowing the compiler, accurately interpreting or reconstructing the original code becomes even harder, increasing the likelihood of errors or misinterpretations in the disassembled output. This information loss could negatively impact classification accuracy.

In fact, some malware authors even employ anti-disassembly techniques to prevent analysis of the

code by malware analysts and hinder malware detection tools that rely on this analysis [21].

Additionally, this minimal information approach offers several advantages. It is broadly applicable across various scenarios and avoids the computational overhead associated with pre-processing steps like disassembly. It is both efficient and adaptable, making it a suitable choice for malware detection.

3.2 Training

The training process involved feeding large and diverse samples of benign code into the transformer model. During training, the model learns patterns from these benign executables and develops an understanding of the statistical properties of legitimate software. This knowledge will be used to assess whether new samples are benign or malicious.

In each iteration of training a transformer model, the process begins by feeding a batch of input data into the model. This data is first tokenized and converted into numerical embeddings, which serve as the model's input.

For example, given a sequence of bytes $x = (x_1, x_2, \ldots, x_S)$ where S is the length of the sample, each byte in the sequence is represented as a vector embedding. These embeddings capture the contextual information of the bytes and are processed by the transformer through multiple layers of self-attention and feed-forward networks.

Within these layers, the model progressively builds a contextual representation of each byte in the sequence. The final layer of the transformer produces a set of logits $z = (z_0, z_1, \ldots, z_{255})$, where each z_b corresponds to the raw score for byte b, representing the likelihood of byte b being the next in the sequence, based on the context of the previous bytes. The index b ranges from 0 to 255, representing the 256 possible byte values.

To convert these logits into a probability distribution, the softmax function is applied. The softmax function is defined as:

$$P(x_{s+1} = b \,|\, x_1, x_2, \ldots, x_s) = \frac{e^{z_b}}{\sum_{i=0}^{255} e^{z_i}} \tag{3.1}$$

Here, $P(x_{s+1} = b \,|\, x_1, x_2, \ldots, x_s)$ represents the probability of the next byte being byte b given the context of the previous bytes $x_1, x_2, ..., x_s$. z_b is the logit (raw score) for byte b, and the denominator is the normalization term, which sums over the exponentiated logits for all possible bytes, ensuring that the probabilities sum to 1[1].

Once the probabilities are computed, the model evaluates its performance by comparing its predictions to the actual target values using a loss function. This function measures how far the model's predictions are from the true labels.

The loss for each byte position s is calculated using the cross-entropy loss function. This loss function measures the difference between the predicted probability distribution and the true distribution. Specifically, for each byte position s, the probability of being the true byte y_s is calculated as:

[1] https://machinelearningmastery.com/softmax-activation-function-with-python/

$$loss_s = -\log P_s(y_s) \tag{3.2}$$

where $P_s(y_s)$ is the probability assigned to the true byte y_s at position s.

The total loss for the entire sequence is then computed as the average of the individual losses across all byte positions:

$$loss = \frac{1}{S}\sum_{s=1}^{S} loss_s \tag{3.3}$$

where S is the length of the sequence. This average loss guides the model's optimization process.

The computed loss is then used to perform a backward pass, where gradients are calculated through backpropagation. This step determines how much each model parameter contributed to the error.

Using these gradients, the optimizer updates the model's parameters, adjusting the weights to minimize the loss and improve prediction accuracy. This adjustment process is essential for the model to learn from its errors.

These steps are repeated for many iterations, with each cycle refining the model's performance as it learns from new batches of data. The training is continued until the model's performance stabilizes and the loss stops decreasing, indicating that the model is effectively learning and generalizing from the training data [13].

3.3 Aggregated Cumulative Value

After training, the model is ready to make predictions. To predict whether a sample is benign or malware, a forward pass is employed to obtain the probabilities of each byte occurring in benign software, given the context of the preceding bytes. These probabilities are used to calculate a value called the Aggregated Cumulative Deviation (ACD), which is a measure of the level of "abnormality" of a program. This value is then compared against a predefined threshold to classify the sample as benign or malware.

In a generative model, the next byte would be selected using the probabilities obtained with a forward pass[2]. For the purpose of this work, however, it's sufficient to isolate the actual next byte's probability from the probability distribution matrix.

After obtaining the probabilities of every byte in the sample in this way, the moving average is calculated and plotted to visualize code patterns. For a sample s constituted of data array $a = (a1, a2, \ldots, a_N)$ and a window size n, the moving average centered at index i can be written as:

$$\text{MA}_i = \frac{1}{n}\sum_{j=i}^{i+n-1} a_j \tag{3.4}$$

where i is the starting index of each window of size n, and MA_i gives the average probability over

[2]https://jalammar.github.io/illustrated-gpt2/

this window.

The resulting plot, referred to as the "moving average of code predictability plot", offers insights into the consistency of the code's predictability. Figure 3.1(a) illustrates a simplified example scenario of overlapping predictability curves for malware and benign software, using simulated data.

As depicted, it is anticipated that in both cases, a significant portion of the code will exhibit high probabilities, reflecting the software patterns learned by the model. However, in benign software, it is anticipated that the code will maintain a relatively stable and predictable structure throughout. In contrast, malware is expected to show sudden and extreme drops in predictability due to the presence of malicious or obfuscated instructions.

Subsequently, M evenly spaced discrete numbers between 0 and 1 are generated, each number representing a probability ratio. The percentage of code that falls below each of these values is calculated. A plot, referred to as the "cumulative deviation plot", is then constructed, displaying the percentage of the code below each threshold. Figure 3.1(b) presents the curves corresponding to the simulated data used to generate Figure 3.1(a).

As shown, malware tends to have a significant portion of its code falling below 50% or even 25% probability, whereas benign software typically maintains more predictable behavior, without such pronounced deviations. Consequently, the curve for malware grows faster, resulting in a larger total area under the curve and, therefore, a larger mean.

In this work, the mean area under the curve, referred to as the Aggregated Cumulative Deviation (ACD), is used to classify malware. The cumulative deviation plot can be understood as a function $f(x)$ that maps probability thresholds to percentages of code. The total area under the curve corresponds to the definite integral of $f(x)$ over the interval [0, 1], which captures the total "accumulated percentage" across all thresholds. Consequently, the mean area under the curve serves as a measure of the function's overall magnitude. This aligns with the use of definite integrals to analyze accumulated quantities [3].

In order to calculate the mean area, the total area under the curve is first approximated using a trapezoidal approach, which divides the curve into small intervals of width dx. For each interval $[x_i, x_{i+1}]$, the area is calculated by summing the area of the rectangle at the left endpoint $y_i \cdot dx$ with the area of the trapezoid formed between y_i and y_{i+1}. The approximate area for each interval val_i is given by:

$$\text{val}_i = y_i \cdot dx + \frac{dx}{2} \cdot (y_{i+1} - y_i) \tag{3.5}$$

where:

- y_i and y_{i+1} are the values of the function $f(x)$ at points x_i and x_{i+1}, respectively
- dx represents the width of each interval.

Then, the total area under the curve is calculated as the sum of the areas of all intervals, which is an approximation of the definite integral:

$$\int_0^1 f(x)\,dx \approx \sum_{i=1}^{n} \text{val}_i \tag{3.6}$$

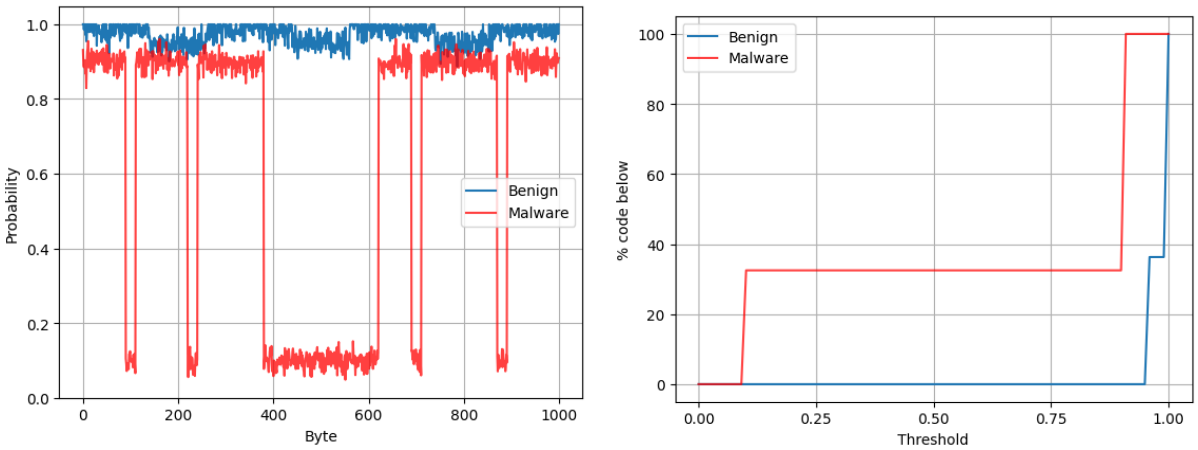
[3] https://www.examples.com/ap-calculus/using-definite-integrals-to-determine-accumulated-change-over-an-interval

In order to obtain the mean area under the curve, the total area under the curve is divided by the number of intervals:

$$\text{Mean Area} = \frac{\sum_{i=1}^{n} \text{val}_i}{n} \tag{3.7}$$

This value, named in this work as the Aggregated Cumulative Deviation (ACD), is used to differentiate between malware and benign software by comparison to a threshold value.

3.4 Class Prediction



(a) Example of moving average of code predictability plot.

(b) Example of cumulative deviation plot.

Figure 3.1: Comparison of the moving average of code predictability and cumulative deviation plots for malware and benign software.

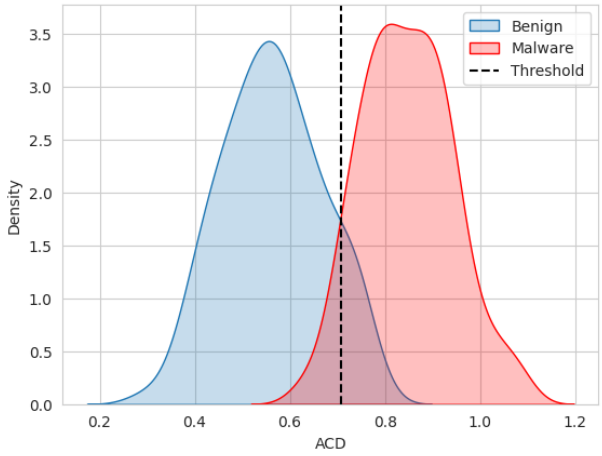


Figure 3.2: Example of density plot.

Class prediction is done by comparison of a given sample with a pre-defined threshold. The threshold is determined by calculating the ACD for both benign and malware samples across all known data and

plotting these values.

In this plot, the curves of malware and software likely overlap in a confusion zone, where the probability of it being malware and the probability of it being benign software are both greater than 0.

In this confusion zone, the threshold is defined as the point where samples below the threshold are more likely to be benign software, and those above it are more likely to be malware. If the ACD of a sample falls below the threshold it is classified as benign software, and if higher it is classified as malware[4].

Figure 3.2 presents an example using real data, showing the density curves of the ACD for benign software and malware. As observed, despite significant overlap between the two curves, a threshold can be established—indicated by the black line—such that samples to the right of the line are more likely to be malware, and those to the left are more likely to be benign.

[4]In the unlikely eventuality that benign software and malware are perfectly linearly separable due to no overlap between the ranges of their ACD values, the threshold could be selected as the middle point between the maximum ACD for benign software and the minimum ACD for malware, in order to maximize its distance to both intervals.

Chapter 4

Data

This chapter details the datasets used for training, validation, and testing in this work. It describes the benign and malware executable samples, their sources, and the selection steps taken to ensure representativity while addressing computational constraints. To optimize efficiency, samples were filtered based on file size, and, in most cases, only a subset of data was selected. Finally, the composition of malware families is presented, providing insights into the diversity of the collected samples.

The data consists of both malware and benign software datasets. The first benign software dataset is a mix of the DikeDataset and locally sourced files and is used for training and validation. For testing, two additional locally sourced benign datasets are utilized: one containing executable files (.exe) and another with dynamic link library files (.dll). For malware, the DikeDataset is employed for training and validation, while the Malicia dataset is used for testing. All files used are Windows PE (Portable Executable) files.

4.1 Training and Validation

This section outlines the sources, composition, and preprocessing of the datasets used for training and validating the malware detection model.

4.1.1 Benign software dataset

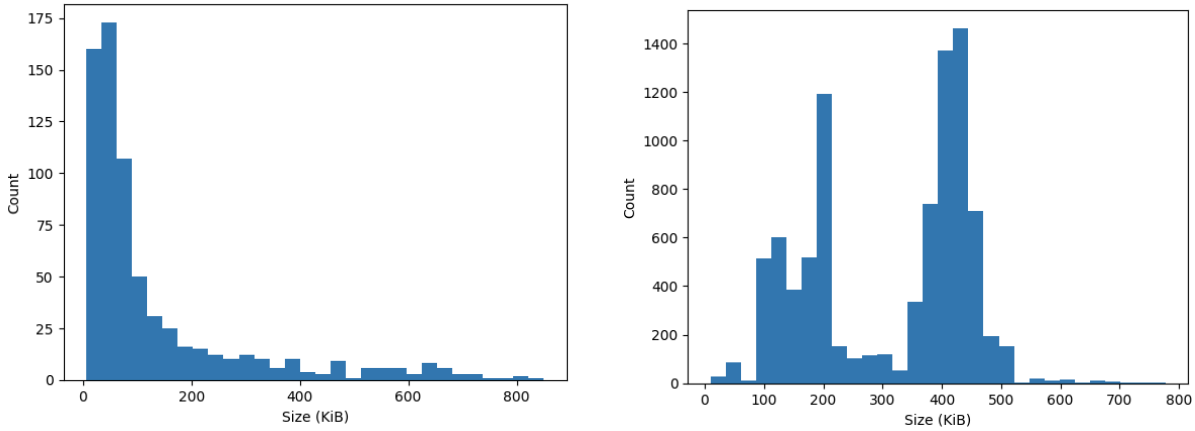
Table 4.1: Summary of the benign executables dataset used for training and validation.

Source	Type	Count	Size	Details
DikeDataset	32-bit programs	441	102.8 MiB	Mostly GUI or console programs
	64-bit programs	521	80 MiB	Mostly GUI or console programs
	Other types	20	1.2 MiB	MS-DOS executables and data
Local File System	32-bit programs	388	2 GiB	Mostly GUI or console programs
	64-bit programs	379	1.8 GiB	Mostly GUI, console, or DLL GUI
	Other types	620	12.8 MiB	MS-DOS executables, data, OpenPGP keys

The benign executables dataset used for training were obtained from two sources. The first was DikeDataset [77], an open-source repository of labeled benign and malware Windows PE and OLE files. In total, 982 benign PE executables (.exe), with a combined size of 184 MiB, were obtained from this

source. Additionally, 1387 benign PE executables, totaling 3.7 GiB, were obtained from the local file system.

The linux command `file`[1] was used to determine the file type. As shown in Table 4.1, the 32-bit programs represent the largest portion of both datasets in terms of total file size, consisting of various types of programs. Combining these two sources resulted in a total of 829 (2.1 GiB) benign 32-bit PE executables and 900 (1.8 GiB) benign 64-bit PE executables.



(a) Benign software (b) Malware

Figure 4.1: Comparison of the histograms for benign and malware program sizes in the training and validation data.

Due to insufficient computational resources for training with the entire dataset, particularly when dealing with programs of large file sizes, only files under 100 KiB were used for training. The viability of this decision, while maintaining adequate representativity, was assessed by studying the distribution of file sizes.

The file size distribution, shown in Figure 4.1(a), follows a Weibull-like pattern, with most samples being smaller than 100 KiB, and larger sizes becoming progressively rarer. Consequently, removing larger files preserves the dataset's representativity by focusing on the predominant smaller files, which capture the main characteristics of typical benign samples.

After filtering the larger samples, a validation set was created by randomly selecting 30 samples from the remaining data, evenly divided between two size intervals: 15 samples under 50 KiB and 15 between 50 and 100 KiB. This approach ensures coverage across the full size range while allowing an analysis of file size effects on experimental outcomes.

After the validation samples were excluded, the training set contained 437 samples (20 MiB), with 260 files (7.5 MiB) under 50 KiB and 177 files (12.5 MiB) between 50 and 100 KiB—a dataset size that balances computational feasibility and representativity.

4.1.2 Malware Dataset

The malware data used for training and validation was obtained from DikeDataset [77], consisting of 8967 (2.7 GiB) 32-bit PE executables and 6 (846.5 KiB) 64-bit PE executables. The 64-bit executables were exclusively DLL GUI, while the 32-bit executables were predominantly GUI, DLL GUI, and na-

[1] https://linux.die.net/man/1/file

tive. It was observed that a subset of these samples had UPX compression applied, a known obfuscation technique commonly associated with packaged malware.

Figure 4.1(b) presents the histogram of the file sizes for the entire dataset. Unlike the benign software, this dataset does not follow a Weibull distribution. Instead, two size intervals dominate: between 100 and 200 KiB and between 350 and 475 KiB. This distribution pattern may result from the predominance of samples from a few malware families, with similar core sizes and minor variations due to obfuscation.

Like the benign software validation set, only samples with sizes less than or equal to 100 KiB were used for the malware validation set. From the 32-bit samples, 15 samples with program sizes less than 50 KiB and another 15 samples with sizes between 50 and 100 KiB were randomly selected for validation. This selection utilized the same size intervals as those used for benign software for the same reasons outlined in the previous section and to facilitate comparison.

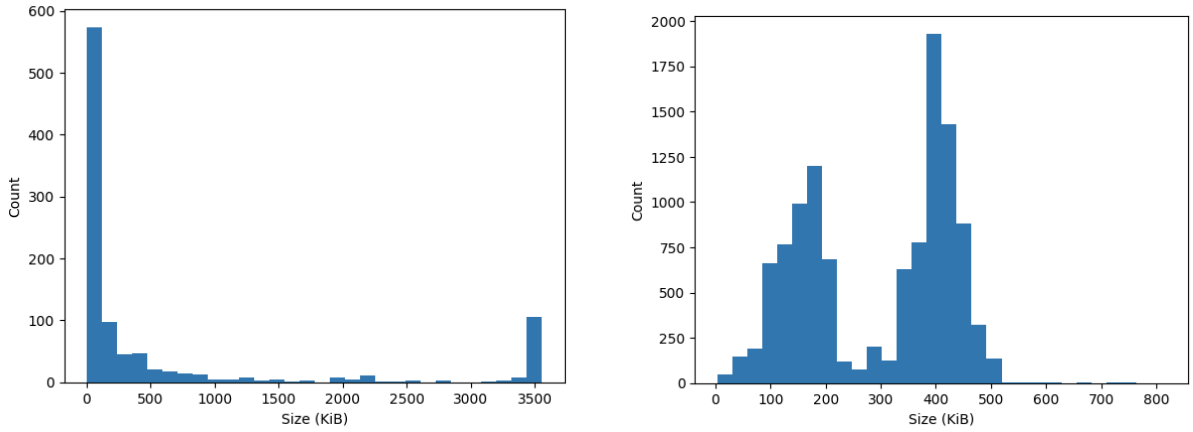
4.2 Testing

This section covers the sources, composition, and pre-processing of datasets for testing the malware detection model, with attention to the family composition of malware samples. The malicious samples were obtained from the Malicia dataset, which includes a diverse range of families such as zbot, zeroaccess, and cridex, while the benign software samples were sourced from local file systems.

4.2.1 Benign Programs Dataset

A set of 1,071 (1.0 GiB) PE executable files was obtained from a different machine than the one used for the training and validation datasets. As shown in Figure 4.2(a), the distribution closely resembles that of the previously used benign dataset, except for an unusual spike around 3,500 KiB (3.4 MiB).

Due to computational restraints, only programs with sizes less than or equal to 200 KiB were considered, and a subset of 500 was selected for testing. The resulting dataset consisted of 294 samples between 0 and 50 KiB, 116 samples between 50 and 100 KiB, and 90 samples between 100 and 200 KiB. The majority of these were 64-bit programs, all of which were GUI or console applications.



(a) Benign software

(b) Malware

Figure 4.2: Comparison of the histograms for benign software and malware program sizes in the testing data.

4.2.2 Malicia Dataset

The test dataset of malicious samples was constructed using the Malicia dataset, a collection of 11,368 (3.1 GiB) malicious PE files (.exe and .dll) gathered from 500 drive-by download servers over an 11-month period [78]. All samples are 32-bit GUI applications or DLLs, except for one console application.

The histogram of file sizes presented in Figure 4.2(b) exhibits the same unusual pattern as observed in the first malware dataset. The dataset that is at the base of DikeDataset cites both the Malicia dataset and Virusshare as its sources [79], indicating a possibly significant redundancy between the Malicia and DikeDataset, which raises questions regarding the robustness of the observed patterns.

However, it seems unlikely that the size distribution of the samples could be maintained when mixed with Virusshare data unless the number of Virusshare samples is minimal. Confirmation of whether this pattern is generalizable to other malware datasets would be of significant interest.

From the samples with sizes less than or equal to 200 KiB, a subset of 1,000 was selected for testing due to computational constraints. In the overall set of samples under 200 KiB, the zbot and zeroaccess families were the most prevalent, with 1,797 and 939 samples, respectively, while all other families had fewer than 74 samples. This distribution is not representative of the entire dataset, where the winwebsec family is the largest but is absent in this subset.

To ensure balanced representation in the test set, 250 samples each of zbot (32.3 MiB), zeroaccess (41.2 MiB), and various smaller families (23.6 MiB) were randomly selected, all with sizes less than or equal to 200 KiB. The resulting distribution is shown in Table 4.2, which illustrates that zbot and zeroaccess dominate the sample pool, while numerous other families are represented by significantly fewer samples.

The presence of families or clusters of families[2] with only 1-2 samples raises concerns regarding representativity. These underrepresented families may lead to overfitting, as the model might learn features specific to a few samples rather than generalizable patterns, ultimately compromising classification accuracy.

Table 4.3 summarizes the distribution of file sizes among the sampled malware families, highlighting the absence of zbot samples in the size category of 50 KiB or less, and the predominance of zeroaccess samples in the 100-200 KiB range.

The data reveals an uneven representation of malware families, particularly with regards to zbot and zeroaccess, which may impact the reliability of the results. Future research should aim to balance the dataset by incorporating additional samples from underrepresented malware families to mitigate the effects of uneven distribution.

4.2.3 Benign DLLs Dataset

In addition to the other datasets, 5,480 (3.0 GiB) benign DLL samples were obtained from the local file system. Since the benign datasets used for training, validation, and testing consisted solely of executable files, This DLL dataset serves as a test set to ensure that benign DLL files are not misclassified as malware.

[2]The clusters identified in the table, which are determined by the authors of the Malicia dataset, group related malware families based on shared characteristics or behaviors. While distinct from malware families, these clusters serve a similar function in labeling and organizing the dataset, helping to simplify the classification task and enhance data analysis.

Table 4.2: Number of samples per family.

Family	*Samples*	*Family*	*Samples*
zbot	250	CLUSTER:colorballs	1
zeroaccess	250	CLUSTER:azonpowzanadinoar.com	1
cridex	53	securityshield	1
CLUSTER:85.93.17.123	42	CLUSTER:blackandwhite	1
harebot	32	CLUSTER:bundlemonkey.com	1
CLUSTER:newavr	26	CLUSTER:clarkclark	1
cleaman	21	fakeav-webprotection	1
CLUSTER:astaror	20	dprn	1
CLUSTER:chapterleomemorykombo.eu	7	CLUSTER:mergenew	1
spyeye-ep	5	CLUSTER:mindamura.com	1
ransomNoaouy	5	CLUSTER:up2x.com	1
ramnit	4	CLUSTER:price.dtdns.net	1
CLUSTER:positivtkn.in.ua	4	CLUSTER:in7cy	1
WinRescue	3	CLUSTER:paydayloatnsstc.ru	1
winwebsec	3	CLUSTER:justontime-12.com	1
CLUSTER:foreign	3	CLUSTER:m9swachu.be	1
CLUSTER:online-police.com	2	CLUSTER:mycomputer	1
fakeav-rena	2	CLUSTER:dzony3777.su	1

Table 4.3: Distribution of file sizes among sampled malware families.

Malware Family	**Size $\leq$ 50 KiB**	**Size 50-100 KiB**	**Size 100-200 KiB**
zbot	-	28	222
zeroaccess	-	1	250
other families	56	91	203

From the set of programs with sizes less than or equal to 200 KiB, 250 were randomly selected for testing, the majority of which were 64-bit DLL GUI and DLL console programs. The selected subset consisted of 136 samples within the 0-50 KiB size range, 54 samples between 50-100 KiB, and 60 samples between 100-200 KiB.

Similar to the benign executable testing dataset, Figure 4.3 shows a very similar distribution to the first benign dataset, albeit with most of the samples smaller than 50 KiB instead of 100 KiB.

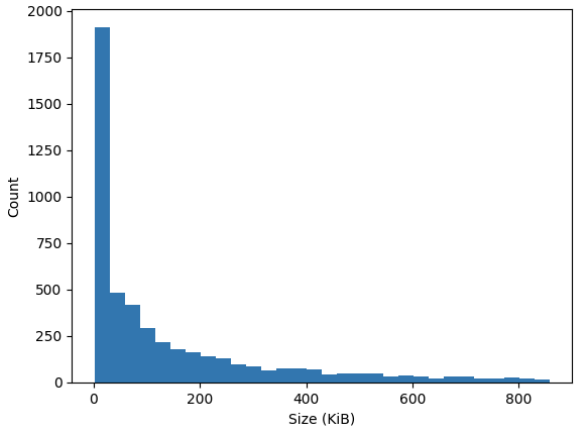


Figure 4.3: Histogram of benign DLL program sizes.

Chapter 5

Results

This chapter presents the results of training and evaluating the transformer-based model to distinguish between benign and malicious software. The chapter is organized into three main sections: Training, Validation, and Testing. The datasets used for these phases, including file size distribution and malware family composition where applicable, are detailed in Chapter 4.

The Training section outlines the process of fine-tuning the model, focusing on systematically adjusting key hyperparameters such as the number of layers, attention heads, batch size, block size, and learning rate. This phase aimed to optimize model performance within computational constraints, and the best-performing configuration was identified after testing several setups.

The Validation section describes the initial assessment of the model's generalization ability using a separate subset of data. This phase ensured that the model could effectively recognize patterns in data it had not seen during training.

The results indicated that patterns in benign software and malware were sufficiently distinct, allowing the calculated Aggregated Cumulative Deviation (ACD) to serve as a reliable differentiator, especially when file size was considered in the classification.

Finally, the Testing section reports on the model's performance on unseen data to evaluate its ability to generalize. Evaluation was conducted on multiple datasets, including locally sourced benign software and malware samples from the Malicia dataset. A classification threshold based on ACD values was set for binary classification, and the model's performance was measured using several metrics.

Performance was strongest for file sizes in the 50–100 KiB range, with reduced accuracy in other size intervals, where the distinction between the ACD of benign software and malware became less clear. These results demonstrate the model's potential to accurately classify malware and benign software.

In this chapter, the mean and Standard Error of the Mean (SEM) are used to summarize the results of several experiments. A detailed explanation of these measures, including their definitions and formulas, is provided in Section 5.1.

5.1 Statistical Measures

This section details the formulas used for the mean, the standard deviation, and the Standard Error of the Mean (SEM). These statistical measures are used throughout the chapter to assess the precision and reliability of the results.

The mean is calculated using the following formula:

$$\text{Mean} = \frac{\sum_{i=1}^{n} x_i}{n} \tag{5.1}$$

where x_i represents the individual values and n is the total number of samples.

The standard deviation s is used to measure the dispersion of the data around the mean and is calculated as follows:

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \text{Mean})^2}{n-1}} \tag{5.2}$$

where xi represents individual values, Mean is the mean of the dataset, and n is the number of samples.

The Standard Error of the Mean (SEM) is calculated to measure the variability of the mean across the samples:

$$\text{SEM} = \frac{s}{\sqrt{n}} \tag{5.3}$$

where s is the sample standard deviation and n is the number of samples.

5.2 Training

The training process focused on fine-tuning a transformer-based model by systematically adjusting key hyperparameters, such as the number of layers, attention heads, batch size, block size, and learning rate. Several configurations were tested to maximize model learning within computational constraints, balancing model complexity with computational efficiency.

The best-performing configuration stabilized at 10,750 iterations, achieving a minimal loss of 1.2683, demonstrating it effectively captured the patterns in benign software.

5.2.1 Model Parameters

The model used for training was nanoGPT, an open-source implementation by Karpathy [76]. This framework simplifies the customization and fine-tuning of medium-sized GPT models by allowing easy adjustment of several key hyperparameters.

The primary parameters modified, which are standard across transformer-based architectures, include the number of layers, attention heads, batch size, learning rate and block size:

- **Number of layers**: Refers to the number of decoder blocks in the model, with each block containing core components like self-attention mechanisms and feed-forward networks. These layers contribute to the depth of the model, determining its ability to capture complex patterns.
- **Number of heads**: The number of attention heads in each self-attention layer. Each attention head

performs self-attention independently, allowing the model to focus on different parts of the input sequence in parallel before combining them into a cohesive output[1].

- **Batch size**: The number of training samples used in each optimization step. Adjusting the batch size can significantly affect training performance. Typically, batch sizes range from 1 to several hundred, balancing memory constraints and training efficiency.

- **Learning rate**: Controls the step size taken during optimization toward minimizing the loss function [80]. It is a critical hyperparameter that significantly impacts model training stability. Learning rates are typically set to values less than one but greater than 10^{-6}. The default value is commonly 0.01[2].

- **Block size**: Refers to the maximum sequence of continuous tokens a transformer model can process at once during training and inference[3]. Also known as context length or context window size.

5.2.2 Configurations Tested

Table 5.1: Model configurations tested.

	Batch	*Block*	*Layers*	*Heads*	*Embeddings*	*LR*	*Params*	*Train*	*Val*	*Iter*
1	64	256	6	6	$heads \times 64$	6×10^{-4}	10.72M	0.5332	1.3451	26k
2	64	256	6	6	$heads \times 64$	1×10^{-3}	10.72M	0.5488	1.3070	23k
3	64	256	6	6	$heads \times 64$	1×10^{-4}	10.72M	0.7400	1.3784	25k
4	64	256	8	8	$heads \times 64$	1×10^{-3}	25.31M	0.1913	1.3273	20k
5	64	256	4	4	$heads \times 64$	1×10^{-3}	3.21M	1.0034	1.4489	20k
6	256	256	6	6	$heads \times 64$	1×10^{-3}	10.72M	0.3065	1.3798	8k
7	256	256	6	6	$heads \times 64$	6×10^{-4}	10.72M	0.3064	1.4073	8k
8	**64**	**512**	**6**	**6**	**heads** $\times$ **64**	$\mathbf{1 \times 10^{-3}}$	**10.72M**	**0.3619**	**1.2600**	**10k**
9	64	512	8	8	$heads \times 64$	1×10^{-3}	25.31M	0.0971	1.2690	4k
10	64	512	6	6	$heads \times 64$	6×10^{-4}	10.72M	0.3810	1.2953	10k
11	64	512	6	6	$heads \times 128$	1×10^{-3}	42.68M	0.0674	1.2891	5k
12	64	512	6	6	$heads \times 32$	1×10^{-3}	2.71M	0.9409	1.3198	40k

Note: "Batch" - batch size; "Block" - block size; "Layers" - number of layers; "Heads" - number of heads; "Embeddings" - number of embeddings; "LR" - learning rate; "Params" - number of parameters; "Train" - train loss; "Val" - validation loss; "Iter" - number of iterations at which validation loss was higher.

The hardware utilized for these experiments comprised an Intel Xeon Silver CPU with 16 cores and hyper-threading, 64 GB of RAM, and two NVIDIA T4 GPUs, each equipped with 16 GB of VRAM. A comprehensive and systematic fine-tuning of the model is necessary to identify the optimal configuration. However, this work was constrained by limited time and computational resources. Consequently, the focus was directed toward achieving the best possible configuration within these limitations while gaining valuable insights into how parameter adjustments influence performance.

[1] https://medium.com/the-research-nest/explained-transformers-for-everyone-af01cbe600c5

[2] https://medium.com/the-research-nest/explained-hyperparameters-in-deep-learning-9b1e0f3b9029

[3] https://medium.com/sage-ai/demystify-transformers-a-comprehensive-guide-to-scaling-laws-attention-mechanism-fine-tuning-fffb62fc2552

To identify the optimal model, various configurations of the model parameters were evaluated, as shown in Table 5.1.

Preliminary tests indicated that more complex models generally yielded better results. Therefore, the initial configuration was selected to approach the upper limits of computational resources while allowing room for experimental adjustments.

These tests also revealed that, for most configurations, the model would enter a loss plateau before 5,000 iterations, after which the loss either stabilized, decreased at a slower rate, or, in some cases, even increased. To ensure sufficient time for convergence and thorough minimization of the loss function during training, 60,000 iterations were run for all configurations.

The majority of tests used a parameter count of 10.72M, chosen based on a widely-used heuristic suggesting that the number of parameters should be proportional to the size of the dataset in bytes. This practice helps prevent overfitting by limiting model complexity relative to the data size while still providing sufficient capacity to learn meaningful patterns.

Initial tests employed learning rates of 6^{-4} and 1^{-3}, well established values for small datasets, resulting in configurations 1 and 2. Model 2, with a lower learning rate, outperformed Model 1, prompting further reduction of the learning rate in Model 3. However, no further decrease in validation loss was observed.

To enhance the model's discriminative capability, the number of layers and attention heads were increased to boost model complexity, using Model 2 as the baseline, which had demonstrated the best performance thus far. This resulted in Model 4. Conversely, a simplified version (Model 5) with reduced complexity was also tested. However, neither adjustment led to improved results.

An increase in batch size from 64 to 256 (Model 6) was explored, resulting in a larger number of bytes used to update the model at each iteration. However, this adjustment worsened results and caused the loss to converge prematurely at 8k iterations, potentially at a local optimum. Re-running the experiment with a lower learning rate (Model 7) did not resolve this issue.

Configuration 8 increased the block size to 512 bytes, expanding the prediction context and achieving the best performance, with a validation loss of 1.269 around 10k iterations. Attempts to further increase model complexity (Model 9) and decrease learning rate (Model 10) did not yield any improvements.

Further experiments with varying the number of embeddings (Models 11 and 12) also failed to produce notable enhancements.

As previously stated, the training data consists of 20 MiB of benign software. For batch sizes of 64 and 256, completing a single epoch would require 312,500 and 78,125 iterations, respectively. However, due to computational constraints, none of the tests exceeded 60,000 iterations—significantly fewer than needed for a full pass through the dataset.

Despite this, stability was observed in all models by this iteration count, suggesting that fundamental characteristics of benign software had been learned. With a larger and more diverse dataset, more iterations would likely be needed for model stabilization.

Given these constraints, Model 8 emerged as the top performer, followed closely by Model 9. To validate this outcome, each configuration was trained 10 times, and the average loss was computed using Equation 5.1. Model 8 achieved the lowest mean loss of 1.2683 with a Standard Error of the Mean (SEM) of 0.0017, as calculated in Equation 5.3, consistently reaching minimal loss at precisely 10,750 iterations.

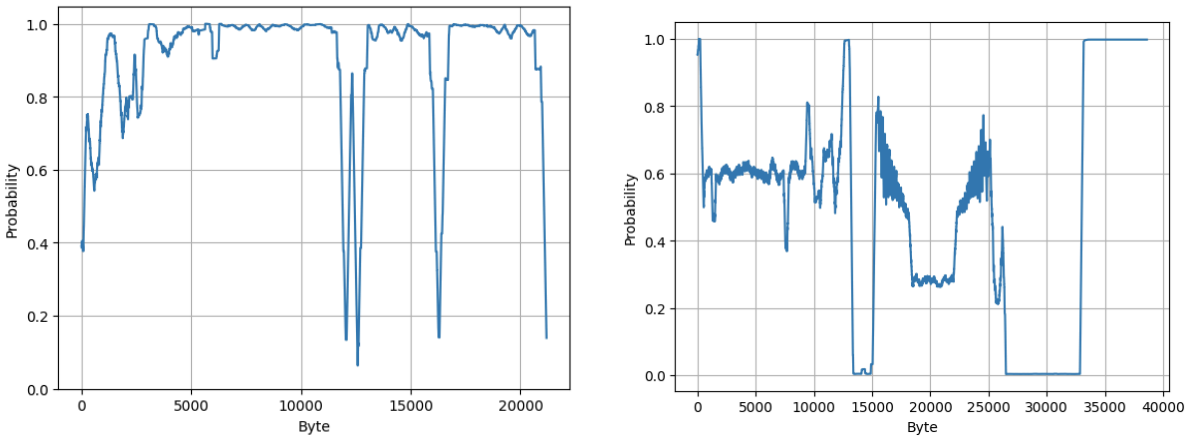
Model 9 had a mean loss of 1.2753 and a SEM of 0.0012.

5.3 Validation

In the validation phase, the model's performance was assessed using a separate subset of data that was not part of the training set. This step was crucial to check whether the model was learning effectively. If the validation results indicated poor performance, adjustments to the parameters would have been necessary to improve the model's ability to generalize to unseen data.

However, no further fine-tuning of parameters was performed during this phase. Results showed that the Aggregated Cumulative Deviation (ACD) of malware and benign software were significantly different, with distinct patterns emerging based on file size. This suggested that the two classes could be effectively distinguished.

The validation dataset consisted of 30 benign software samples and 30 malware samples, evenly split between the 0–50 KiB and 50–100 KiB size ranges. After running the samples through the model, which provided the probability of each byte for every sample, the moving average was calculated and plotted using a moving window of 300 bytes.



(a) Benign software

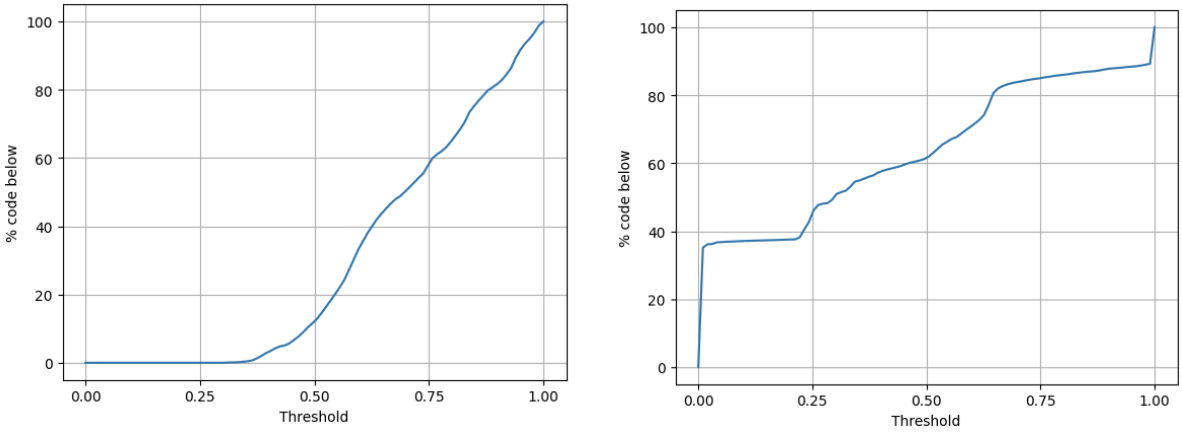
(b) Malware

Figure 5.1: Examples of moving average of code predictability plots for benign software and malware in the validation data.

Figures 5.1(a) and 5.1(b) present a common pattern observed in the resulting plots, referred to as moving average of code predictability plots. In benign software, prediction ability occasionally drops to very low values, but only for small intervals. In contrast, malware often exhibits entire sections of low probability. This distinctive characteristic helps to differentiate between benign software and malware.

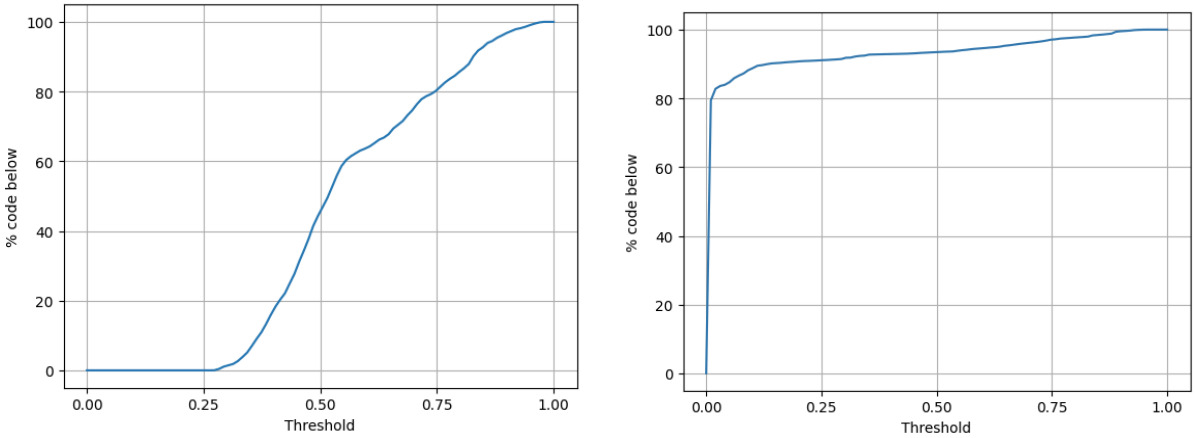
As previously discussed, 100 evenly spaced thresholds were generated between 0 and 1, and the percentage of code below each of them was calculated. The resulting cumulative deviation graphs revealed a significant distinction between benign software and malware, with greater consistency than in the moving average of code predictability plots.

For samples of 50 KiB or less (Figures 5.2(a) and 5.2(b)), the curves of the malware grow significantly faster than those of benign software, with a substantial portion of code falling below the 50% and even 25% probability thresholds. In contrast, benign software generally has little to no code below these thresholds. Overall, malware exhibits near-linear growth, whereas benign software follows an almost



(a) Benign software (b) Malware

Figure 5.2: Examples of cumulative deviation plots for benign software and malware (size 0-50 KiB) in the validation data.



(a) Benign software (b) Malware

Figure 5.3: Examples of cumulative deviation plots for benign software and malware (size 50-100 KiB) in the validation data.

exponential pattern. Additionally, malware often contains a notable amount of code with probabilities at or near 0%, a behavior rarely observed in benign software.

For samples between 50 and 100 KiB, most benign software curves resembled those in the 0–50 KiB range, with a slight increase in the amount of code below the 50% threshold (Figure 5.3(a)). However, a few exceptions displayed topologies similar to the malware in the 0-50 KiB range. Interestingly, the majority of malware in this size range showed even more pronounced growth, as depicted in Figure 5.3(b).

Table 5.2: Comparison of the mean and SEM of benign software and malware in the validation data.

	Mean	*SEM*
Benign 0-50 KiB	0.2296	0.0277
Benign 50-100 KiB	0.3694	0.0331
Malware 0-50 KiB	0.4983	0.0246
Malware 50-100 KiB	0.8846	0.0272

Subsequently, the area under the curve at each point was approximated and averaged to derive the Aggregated Cumulative Deviation (ACD) for each sample. The mean and Standard Error of the Mean

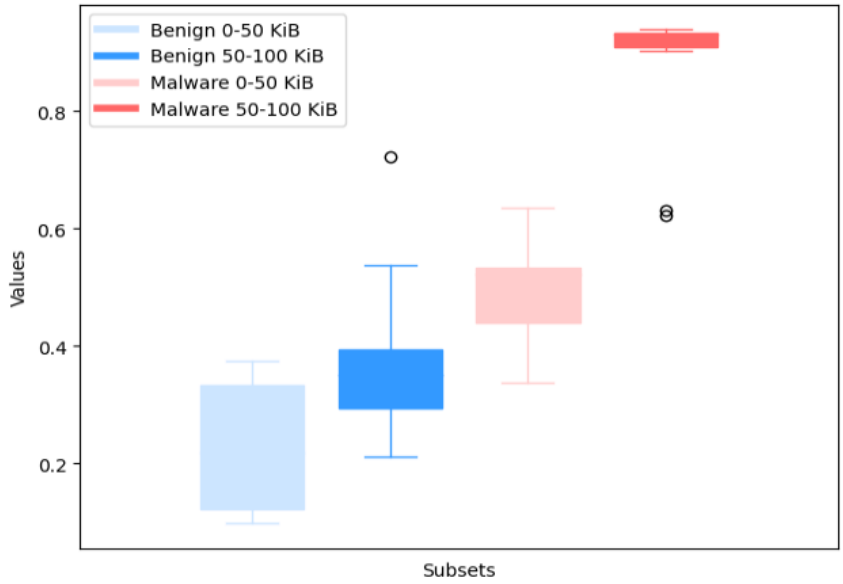


Figure 5.4: Comparison of the boxplots of benign software and malware in the validation data.

(SEM) were then computed using Equations 5.1 and 5.3, respectively. These values are presented in Table 5.2, and Figure 5.4 displays the corresponding boxplots.

Both the table and the boxplots reveal a significant difference between the mean ACD of benign software and malware, across both size intervals. The boxplots also show a pronounced difference in the maximum, minimum, and quartiles of the ACD values between the two classes. Therefore, it should be feasible for a classifier to distinguish between the two by comparing the ACD to a threshold value.

Additionally, the mean ACD appears to increase with the size interval for both benign and malware software, suggesting that the classification threshold may need to vary based on program size.

5.4 Testing

In the testing phase, the model's performance was evaluated on separate, unseen datasets to evaluate its generalization ability. This phase aimed to determine how well the model could classify data it had not encountered during training.

The evaluation was conducted on three datasets: the benign programs dataset, the Malicia dataset (consisting of malware families such as zbot and zeroaccess), and the benign DLLs dataset, with samples split into size intervals of 0-50 KiB, 50-100 KiB, and 100-200 KiB. To reflect this, the section is divided into three parts, each corresponding to one of these datasets.

In each section, the model is tested on each dataset, and the resulting mean and SEM of the ACD for the samples, along with their corresponding boxplots, are presented. The second part, which focuses on the Malicia dataset, also examines patterns in the output probabilities among samples from the same family and explores its potential for creating malware family signatures—an important complementary contribution of this work. The core results—the binary classification using the proposed methodology—are also outlined in this section.

As previously discussed, significant differences in the ACD between benign software and malware are used to establish a classification threshold. This threshold is determined by analyzing the distribution

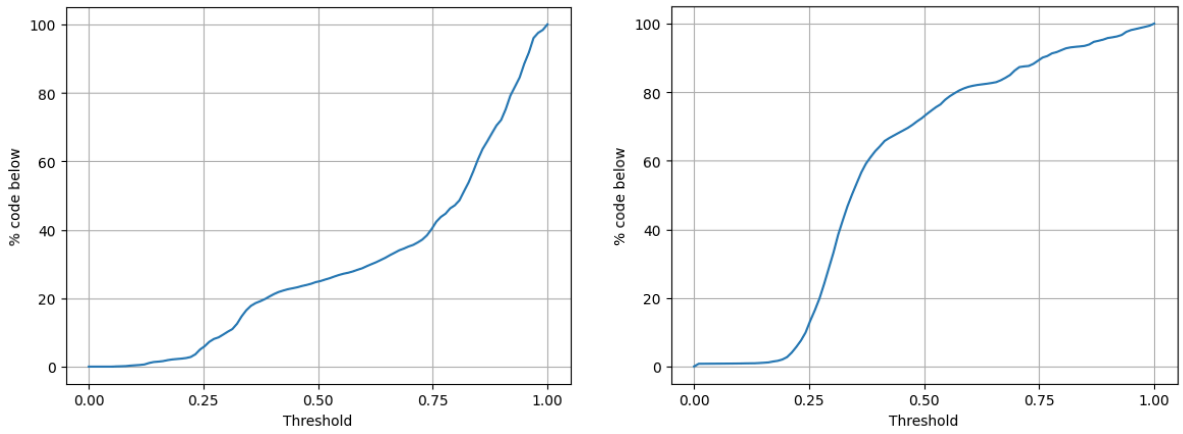
of ACD values for both classes and identifying the point that maximizes separation between the two distributions while minimizing overlap. The chosen threshold enables classification of each sample as benign or malicious, depending on whether its ACD exceeds or falls below this value.

This classification is performed using the benign programs and Malicia datasets, and performance metrics such as precision, recall, and the Matthews Correlation Coefficient (MCC) are calculated based on the results. These metrics showed that the model achieved high accuracy in distinguishing between the two classes in the 50-100 KiB range, although performance declined for other size intervals due to increased variability in the ACD patterns.

5.4.1 Benign Programs Dataset

The benign program dataset was comprised of 500 samples, which were divided into three size ranges: 0–50 KiB, 50–100 KiB, and 100–200 KiB.

Benign software samples in the 0–50 KiB range, despite showing significant variation in the moving average of code predictability plots, generally follow a curve similar to the one shown in Figure 5.5(a). This curve, while not far from the one observed for benign software in the validation dataset, has a much larger percentage of code below 50% probability and even a small amount below 25%. It is therefore more similar to a linear curve.



(a) Size 0-50 KiB

(b) Size 50-100 KiB

Figure 5.5: Examples of cumulative deviation plots of benign software in the benign testing data.

Software between 50 and 100 KiB, on the other hand, exhibits a curve similar to that in Figure 5.5(b), which is more like a logarithmic curve due to a sudden spike in code with a 25–50% probability. This pattern is analogous to the malware observed in the 0–50 KiB range during the validation phase and is markedly distinct from any previously observed.

In software between 100 and 200 KiB, the majority of the curves matched the pattern in Figure 5.5(b), while a few more closely resembled the pattern in Figure 5.5(a).

Table 5.3 presents the mean and SEM of the ACD for the samples, categorized by size intervals. The results show a slight increase in the mean ACD of benign software as program size grows, bringing it closer to the ACD of malware. This observation underscores the importance of separating samples by size interval for accurate analysis.

Table 5.3: Mean and SEM of benign samples in the benign testing data.

	Mean	*SEM*
Benign <= 50 KiB	0.4009	0.0050
Benign 50-100 KiB	0.5209	0.0076
Benign 100-200 KiB	0.5492	0.0103

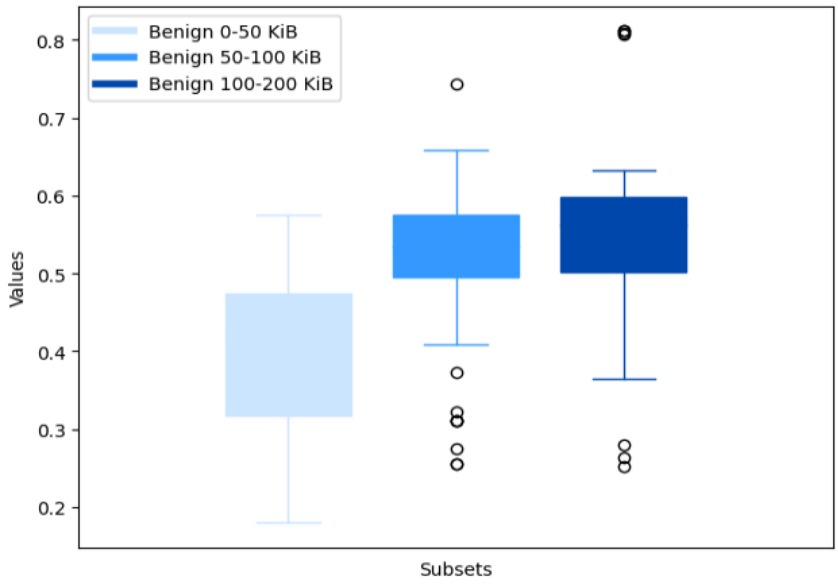


Figure 5.6: Boxplots of benign software in the benign testing data.

Figure 5.6, on the other hand, shows some overlap between the mean ACD of software in the 50-100 KiB and 100-200 KiB ranges, suggesting the necessity of refining these size intervals—a task that should be addressed in future work. Nevertheless, this overlap does not present a significant challenge for classification unless the mean ACD of malware falls within these ranges.

5.4.2 Malicia Dataset

This section presents the findings of the testing conducted using the Malicia dataset.

The first part examines the characteristics of the two major malware families, zbot and zeroaccess, as well as smaller malware families in the dataset. In particular, it focuses on the patterns observed in the moving average and cumulative deviation plots. The section also includes the Aggregated Cumulative Deviation (ACD) statistics for the malware samples, as was done above for the benign software dataset.

The core of the results lies in the final part of the section, which addresses binary classification. A threshold is established to distinguish between malware and benign software, followed by an analysis of the classification performance using this threshold.

Malware Family Analysis

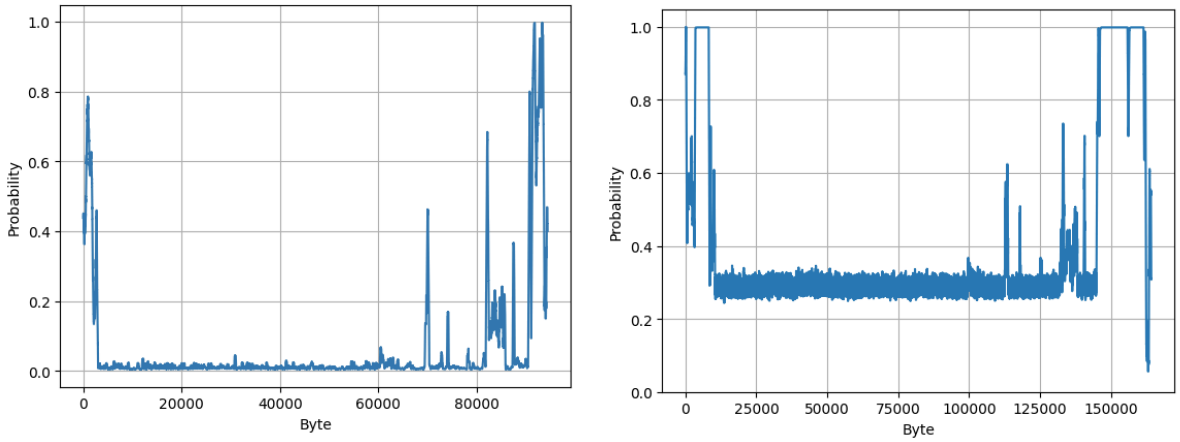
Zbot

The first major subset of the malware is the zbot family, comprising one-third of the samples. Of the 250 zbot samples, the majority were between 100–200 KiB, with only 28 falling in the 50–100 KiB range and

none smaller than 50 KiB.

Among the 50–100 KiB samples, most displayed a curve similar to Figure 5.8(a), with approximately 90% of the code below the 25% probability threshold. Figure 5.7(a) illustrates this pattern, which is characterized by a large area of code near 0% probability. A small subset of samples, however, exhibited a slower curve, as shown in Figure 5.8(b), resembling the malware patterns observed in the validation set (Figure 5.2(b)).

In contrast, a few samples displayed curves similar to those of benign software in the validation set (Figure 5.2(a)), which could potentially lead to false negatives.



(a) Size 50-100 KiB

(b) Size 100-200 KiB

Figure 5.7: Examples of moving average of code predictability plots of zbot malware in the Malicia testing data.

As for the samples between 100 and 200 KiB, most displayed a curve similar to that in Figure 5.8(c), with no code at or near 0% probability, and a sudden spike between 10% and 25% probability. This pattern is explained by a moving average of code predictability plot topology similar to Figure 5.7(b) which, while resembling the 50-100 KiB samples (Figure 5.7(a)), have a low probability plane around 10-30% instead of 0%.

A few samples, however, exhibited curves more akin to those of the 50-100 KiB range, as shown in Figure 5.8(d), or were in between these two topologies.

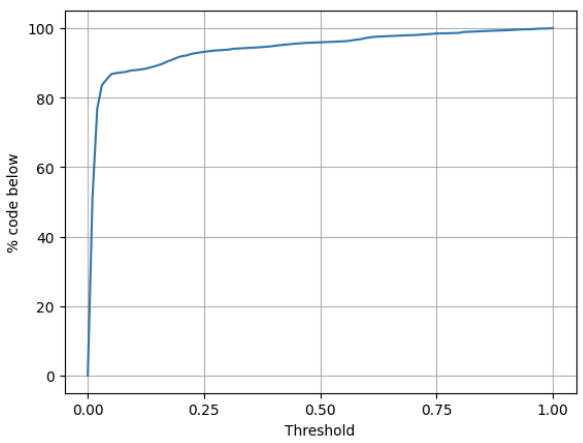
It is therefore plausible to conclude that the majority of zbot samples can be distinguished from benign software.

Zeroaccess

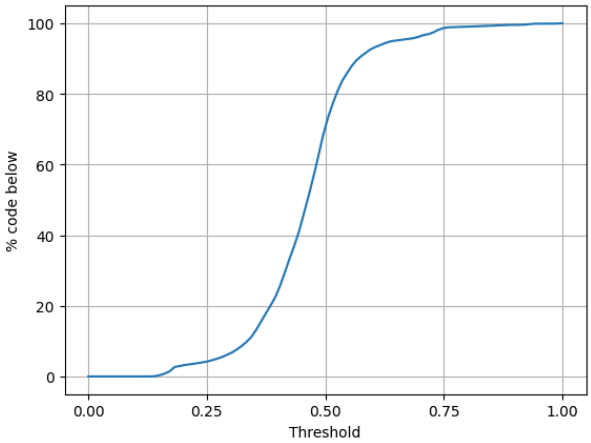
The other major family in the dataset is zeroaccess, which accounts for another third of the samples. Nearly all of these samples were between 100–200 KiB, with only one falling into the 50–100 KiB range.

Zeroaccess showed a few different repeating patterns, one of which is illustrated in Figure 5.9(a). Despite the variety in these patterns, the cumulative deviation plots were very similar.

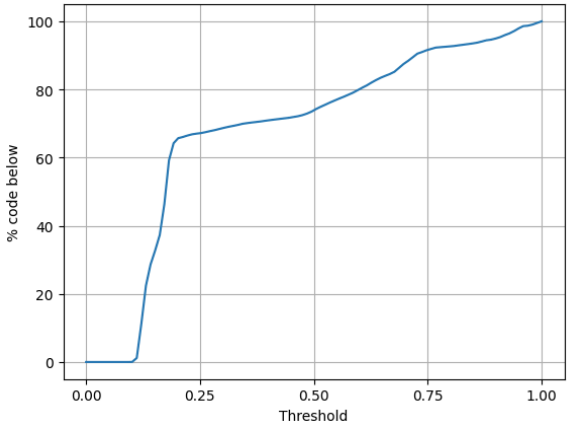
As shown in Figure 5.9(b), the curve starts with a rapid growth, with between 40% and 80% of the code near 0% probability, followed by a gradual increase up to the 50% threshold and a negligible amount of code above 50% probability. The ”concave” shape up to 25% and the ”convex” shape from 25% to 50% are present in the majority of samples, even when the values for these thresholds are significantly



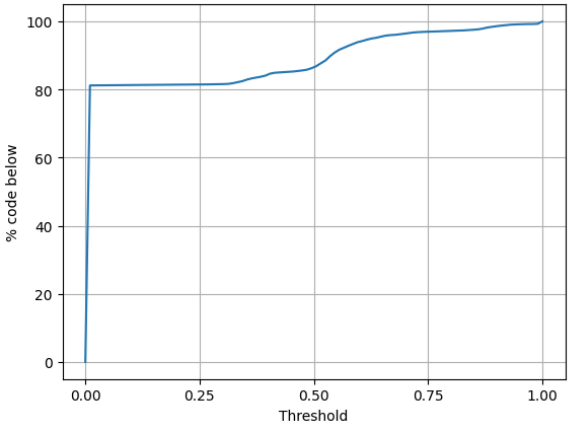
(a) Size 50-100 KiB



(b) Size 50-100 KiB

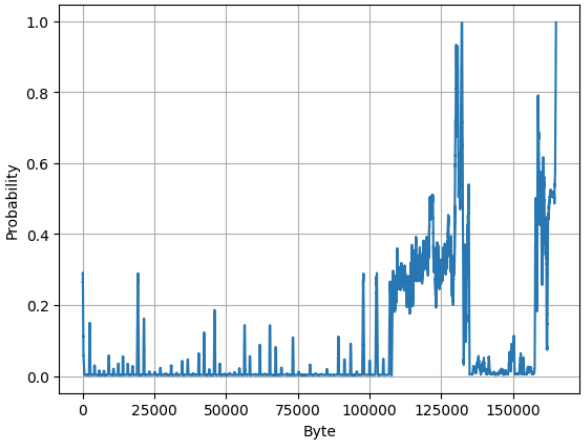


(c) Size 100-200 KiB

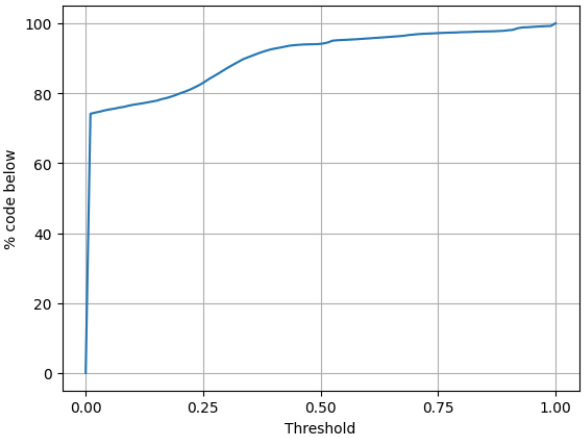


(d) Size 100-200 KiB

Figure 5.8: Examples of cumulative deviation plots of zbot malware in the Malicia testing data.



(a) Moving average of code predictability plot



(b) Cumulative deviation plot

Figure 5.9: Examples of moving average of code predictability and cumulative deviation plots of zeroaccess malware in the Malicia testing data.

different.

Given the substantial amount of code near 0% probability in zeroaccess samples, distinguishing them

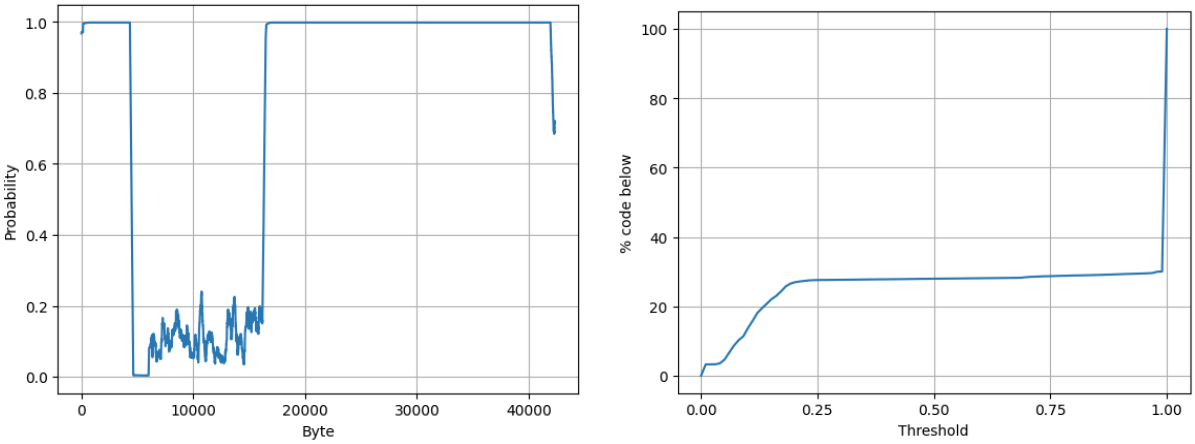
from benign software should be even simpler than distinguishing ZBot samples.

Other Families

The remaining portion of the dataset consisted of 250 samples from various smaller malware families, the majority of which (203 samples) were in the 100–200 KiB range.

The moving average of code predictability plots for these samples displayed a wide variety of shapes. While some malware exhibited curves similar to those of the two major families, others displayed distinct patterns. In addition, similar patterns could also be identified among some samples.

Consequently, a family-by-family study was conducted to analyze the possibility of existence of a malware ”signature” in these plots, which appeared likely based on the relative uniformity of the zbot and zeroaccess samples.



(a) Moving average of code predictability plot

(b) Cumulative deviation plot

Figure 5.10: Examples of moving average of code predictability and cumulative deviation plots of CLUSTER:85.93.17.123 malware in the Malicia testing data.

As for samples of CLUSTER:85.93.17.123, most displayed easily discernible patterns. The majority followed the pattern shown in Figure 5.10(a), characterized by a specific zone where the probability drops steeply from 100% to below 20% before rising again to 100%, and were, in fact, indistinguishable. This pattern may indicate a typical malicious payload embedded in a benign program.

The corresponding cumulative deviation graphs, shown in Figure 5.10(b), exhibit a much slower growth than all the other malware samples observed so far, bearing a closer resemblance to benign software. Additionally, other samples from this malware family exhibited even shallower curves. Although distinguishable from benign software due to a higher presence of code below 50% and 25% probability, many of these samples might constitute false negatives when classifying based on the ACD.

CLUSTER:newavr, cleaman, and CLUSTER:astaror also exhibit consistent behavior, with signatures resembling those of zbot and zeroaccess, making them easily discernable from benign software.

In contrast, Cridex and Harebot do not exhibit discernible patterns. Nevertheless, they, along with the other smaller malware families, display curves with significantly steeper growth than benign software, which allows for a clear differentiation. However, two samples from CLUSTER:online-police.com, as shown in Figure 5.11, exhibit a shallow curve that makes them difficult to reliably distinguish from benign software.

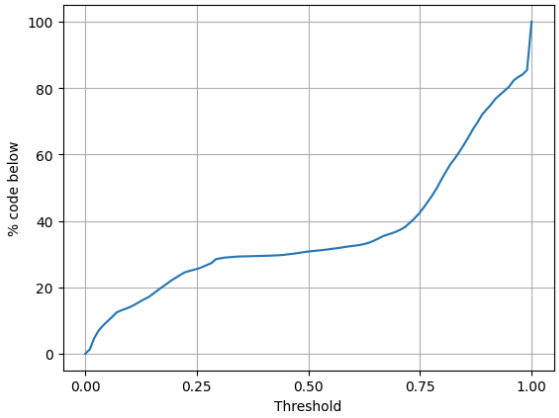


Figure 5.11: Example of cumulative deviation plot of CLUSTER:online-police.com malware in the Malicia testing data.

Although this analysis was not the primary focus of the study, it provided relevant insights into the possibility of using a decoder-based transformer for multi-family classification.

Statistics

The characteristics of the malware families in the dataset having been analyzed, the statistics of all samples are now considered. As with the previous datasets, the area under the curve (ACD) for each sample was approximated, and their mean and SEM were calculated. These results are presented in Table 5.4.

Table 5.4: Comparison of the mean and SEM of Malicia testing data with the benign testing data.

	Mean	*SEM*
Benign 0-50 KiB	0.4009	0.0050
Benign 50-100 KiB	0.5209	0.0076
Benign 100-200 KiB	0.5491	0.0103
Malware 0-50 KiB	0.3340	0.0298
Malware 50-100 KiB	0.7690	0.0112
Malware 100-200 KiB	0.7874	0.0127

Additionally, Figure 5.12 presents the ACD of the samples as boxplots, alongside the benign software data from Section 5.4.1.

By comparing malware to benign software within their respective size intervals, it can be observed that for samples between 50-100 KiB, the minimum and maximum values are almost perfectly separated, although there is significant overlap among the outliers. Thus, classification should proceed relatively smoothly for this interval, albeit with the potential for some false positives and/or false negatives.

On the other hand, significant overlap is observed for samples between 100-200 KiB. Additionally, the boxplot for malware in the 0-50 KiB range falls entirely within the range of the benign software's minimum and maximum values.

These findings suggest that the classification problem may be significantly more challenging than the individual plot analyses initially indicated.

It's also worth noticing that the lack of samples in the 0-50 KiB results in a very small boxplot which is probably not representative of the variance of ACD in this size interval. This might even be the cause for it falling within the range of the minimum and maximum values of benign software in its respective

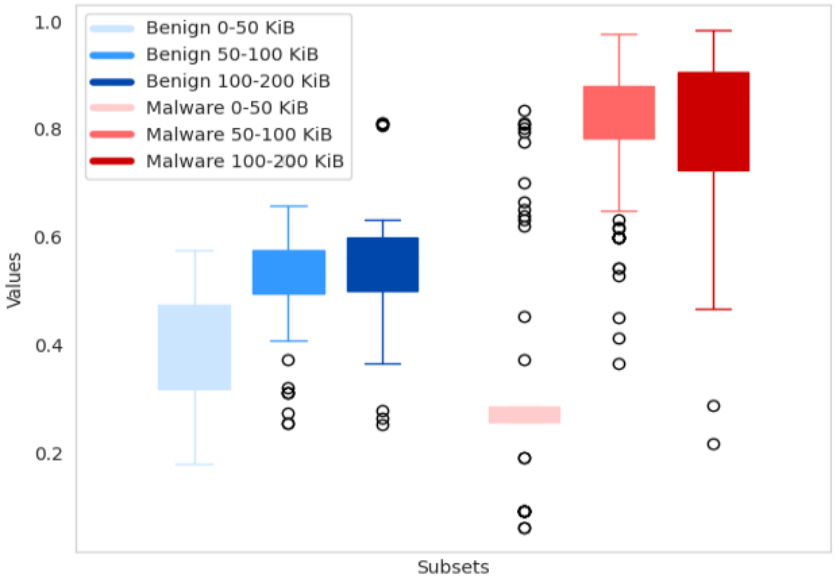


Figure 5.12: Comparison of the boxplots of the malware in the Malicia testing data with the benign testing data.

size interval.

In future work, more data needs to be accrued for malware in this size interval in order to capture the characteristics of these malware samples.

Binary Classification

Threshold Establishment

The next step was to establish the threshold that would enable the classifier to separate malware from benign samples. For this purpose, both the malware and benign programs datasets were plotted as kernel density estimation (KDE) plots[4].

In the confusion zone, where both curves overlap, the threshold was defined as the point where a sample is more likely to be malware than benign software. For instance, in Figure 5.13, which depicts the KDE plot of samples between 50-100 KiB, the threshold is approximately 0.656.

These thresholds were established for each size interval, so that if a new sample falls below the threshold, it is classified as benign software, and if it exceeds the threshold, it is classified as malware.

Despite the apparent differences between the size intervals observed in Figure 5.12, the resulting threshold was 0.6 for both 0-50 KiB and 100-200 KiB size intervals and 0.656 for the 50-100 KiB size interval.

Classification Results

The same dataset was used for classification as a proof of concept, although in a more rigorous approach, separate datasets would be used for establishing classification thresholds and testing. Figure 5.14 shows the obtained confusion matrices and Table 5.5 the respective classification reports.

Precision, defined as the ratio of true positives to total positives predicted, represents the proportion of positive class predictions that were actually correct. For samples with a size below or equal to 50

[4]https://seaborn.pydata.org/generated/seaborn.kdeplot.html

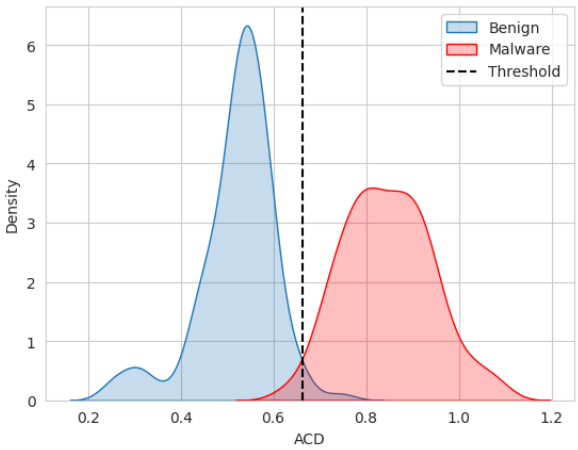
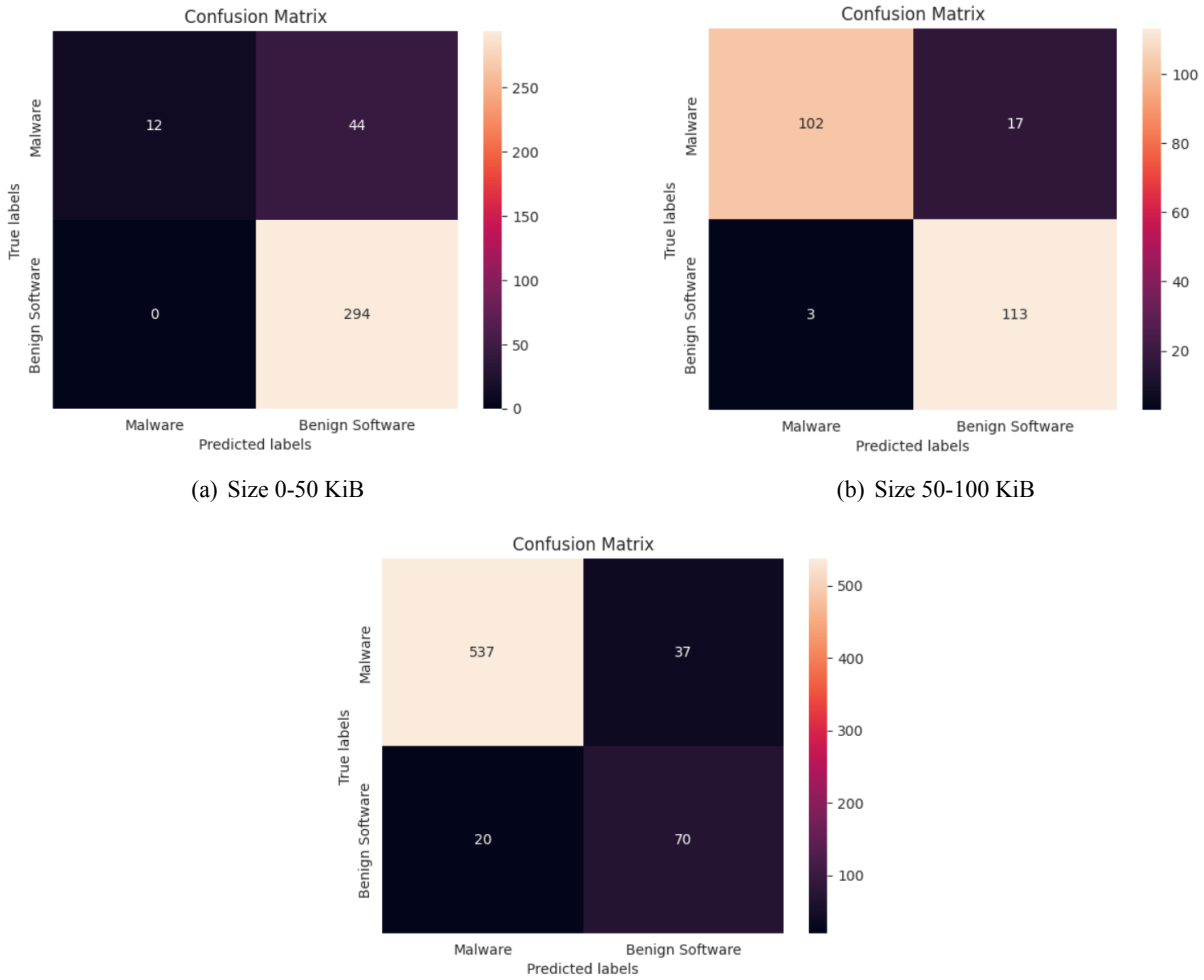


Figure 5.13: Comparison of the KDE plot of malware of the Malicia testing data with the benign testing data (size 50-100 KiB).



(a) Size 0-50 KiB

(b) Size 50-100 KiB

(c) Size 100-200 KiB

Figure 5.14: Confusion matrices for samples split by size interval.

KiB, no benign software samples were misclassified as malware by the model. Consequently, no false positives are present, resulting in a precision of 100%.

Recall, defined as the ratio of true positives to total positives in the ground truth, represents the proportion of true positive class samples that were identified by the model. A low recall of 21% is

Table 5.5: Results of model evaluation of the Malicia testing data and the benign testing data.

	Precision	*Recall*	*F1-score*	*MCC*	*Pos / Neg*
0-50 KiB	1.00	0.21	0.35	0.43	56 / 294
50-100 KiB	0.97	0.86	0.91	0.84	119 / 116
100-200 KiB	0.96	0.94	0.95	0.66	574 / 90

observed, as most malware samples were incorrectly classified as benign.

As a result, the F1-score, the harmonic mean between precision and recall, is significantly low in this size interval.

The final performance metric used in this analysis is the Matthews Correlation Coefficient (MCC), a robust statistic that evaluates the classification performance by incorporating all true and false positives and negatives.

Given the low recall observed, it is not surprising that the MCC for the classifier was suboptimal, reaching only 0.43, a value lower than what would be expected from a random classifier. This underperformance aligns with the overlap observed in the boxplots of Figure 5.12, indicating that the classifier struggled in this specific interval.

It is anticipated that with a more robust model trained on a larger dataset, the separation between malware and benign software distributions would become more distinct, thereby reducing the rate of misclassification. However, if improvements are not observed, it may be beneficial to adjust the decision threshold, provided it does not significantly increase the false positive rate.

For samples in the 50-100 KiB range, a more robust classifier was achieved, correctly identifying the majority of both malware and benign software instances, as evidenced by the confusion matrix in Figure 5.14(b). The classifier attained an F1-score of 91% and an MCC of 0.83, both of which are highly encouraging. These results align with earlier observations that this interval exhibits the highest degree of linear separability between the two classes.

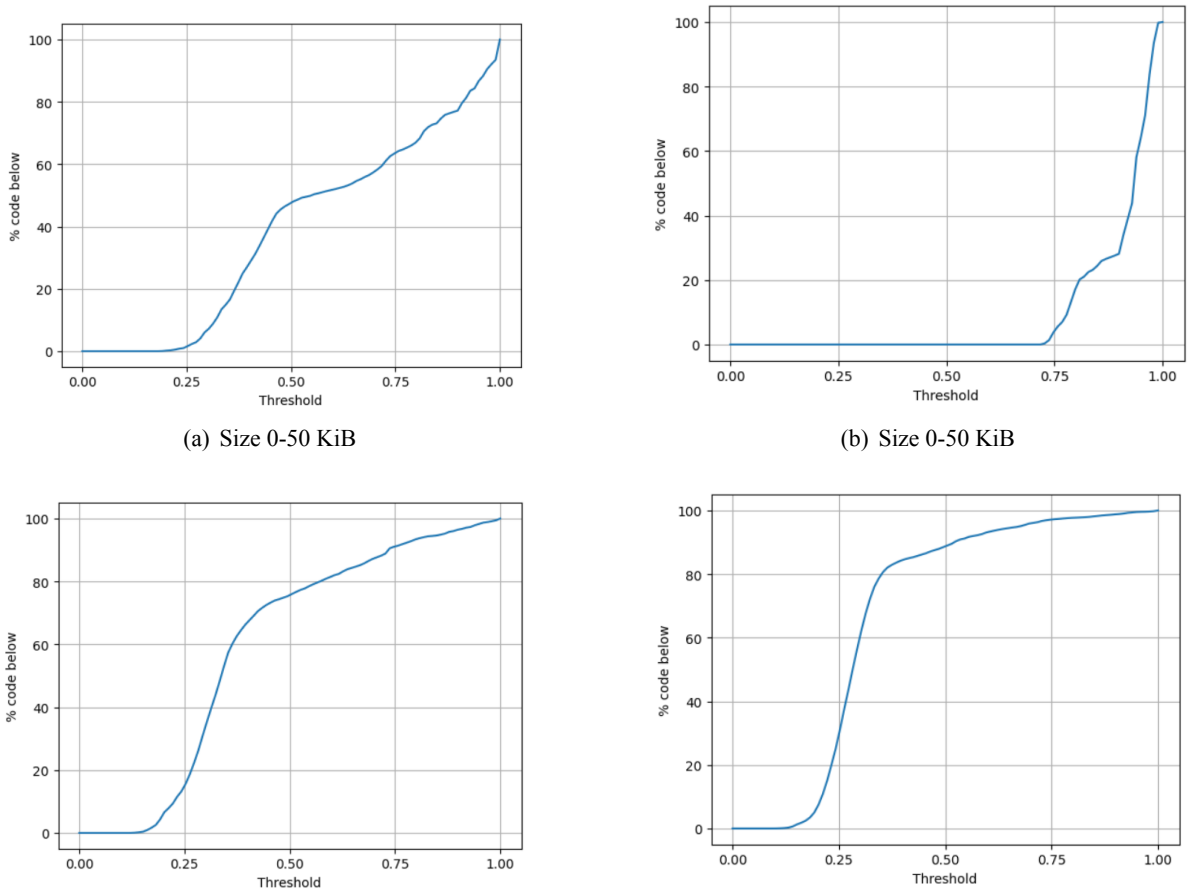
The results for the 100-200 KiB interval may initially seem more favorable, with a recall of 0.94, indicating that 94% of the malware was correctly identified. However, despite this high recall, the MCC was only 0.6641. This lower MCC is likely due to the significant number of false positives, as seen in the confusion matrix in Figure 5.14(c).

5.4.3 Benign DLLs Dataset

Tests were conducted using a benign DLLs dataset to ensure that these files were not misclassified as malware, given that they differ from the executable format used during training. The dataset included 250 samples, with the majority (136 samples) falling within the 0-50 KiB size range.

In the 0-50 KiB range, most samples followed a curve similar to Figure 5.15(a), resembling the previously observed benign software curves. Other samples displayed curves closer to Figure 5.15(b) or somewhere between the two, which are even more distinct from malware patterns.

The programs in the 50-100 KiB range exhibited a similar trend, although with a slightly earlier and more pronounced 'shoulder' in the curve, as shown in Figure 5.15(c). This curve resembles the one observed in the benign programs testing dataset (Figure 5.5(b)), which, as noted earlier, bears some similarity to several malware curves.



(a) Size 0-50 KiB

(b) Size 0-50 KiB

(c) Size 50-100 KiB

(d) Size 100-200 KiB

Figure 5.15: Examples of cumulative deviation plots of benign software in the DLL testing data.

This characteristic becomes even more pronounced in the 100-200 KiB interval, as shown in Figure 5.15(d).

Table 5.6: Comparison of the mean and SEM of the benign software in the DLL testing data with the benign testing data.

	Mean	*SEM*
Benign 0-50 KiB	0.4009	0.0050
DLL 0-50 KiB	0.2685	0.0109
Benign 50-100 KiB	0.5209	0.00756
DLL 50-100 KiB	0.5030	0.0156
Benign 100-200 KiB	0.5492	0.0103
DLL 100-200 KiB	0.5884	0.0093

The area under the curve was approximated at each point and averaged to obtain the ACD, and the mean and SEM of the ACD for the samples were calculated. These results are presented in Table 5.6, with the corresponding boxplots shown in Figure 5.16.

These results show that the distribution of the DLL samples closely resembles that of the executable benign software samples, with the exception of the 0-50 KiB size range, where the mean is significantly lower.

These samples appear to be even more predictable than other benign software, suggesting that they

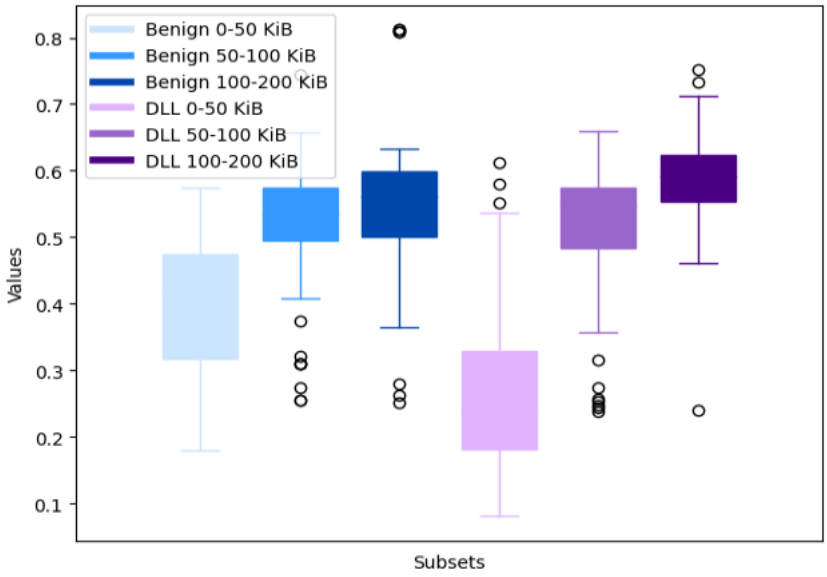


Figure 5.16: Comparison of the boxplots of the benign software in the DLL testing data with the benign testing data.

should pose no challenge to classification. In contrast, the 100-200 KiB interval, with its maximum value exceeding that of other benign software samples, may present some dificulties.

Overall, the classification of benign DLLs is not expected to pose significantly greater challenges than that of other benign software, as their ACD value ranges are largely similar, with the exception of the 0-50 KiB size interval, where classification is expected to be even easier.

Chapter 6

Conclusions

This work introduces a novel methodology for detecting metamorphic malware using machine learning, specifically transformer-based models. The approach is designed to be both efficient and scalable, enabling its application in real-time malware detection for enterprise and endpoint environments.

The methodology is innovative in employing a unidirectional decoder-based transformer to learn patterns in benign software through the static analysis of raw binaries. After training, when provided with a sample, the model outputs the probabilities of occurrence for each byte based on patterns observed in benign software. These probabilities are subsequently used to calculate the Aggregated Cumulative Deviation (ACD), allowing the classification of samples as benign or malicious by comparison to a predefined threshold value.

Various hyperparameter configurations were tested to minimize loss. The methodology was then evaluated using both benign and malware datasets, with testing focused on three size intervals: 0-50 KiB, 50-100 KiB, and 100-200 KiB. Performance metrics, namely F1-score, precision, recall, and Matthews Correlation Coefficient (MCC), were used to assess the results.

The results obtained from these experiments suggest that the methodology presented in this work can be used to detect a wide range of malware, including zero-day threats, as well as polymorphic and metamorphic variants of existing malware. While promising, further research is needed to obtain a robust classifier.

Given the availability of greater computational resources, the classifier could benefit from being trained on a much larger and more diverse dataset, and fine-tuned using more computationally demanding hyperparameters. This would be an important step toward developing a more robust and accurate model. Moreover, by incorporating different types of binaries from various operating systems, the classifier could potentially be generalized to detect malware across multiple platforms (e.g., ELF binaries, office macros, PowerShell scripts).

An important observation from the experiments is the presence of code regions in malware samples where the probability drops to 0% or near 0%, indicating the existence of unexpected code patterns, such as obfuscated or malicious instructions, or encrypted sections that produce ”garbage” zones. An interesting avenue for future research would involve calculating the percentage of code below a low threshold, such as 1%, and incorporating this metric into the classifier, alongside the ACD, to assess whether it improves classification accuracy.

Furthermore, the study utilized three arbitrary size intervals for sample classification, each with its

own threshold. Future work should investigate whether size is an appropriate criterion for separating samples, or if another criterion would yield better results. It is also worth considering whether size-based separation is necessary at all, given the similarity of thresholds across size intervals, which could become less relevant with a larger dataset. If size proves to be an important factor, more precise separation and the expansion of size intervals should be explored to better account for the variety of sample sizes.

An additional focus should be evaluating the model's performance on more representative real-world datasets. The current study may have been influenced by an oversampling of malware, particularly in the 100-200 KiB interval, where malware samples far outnumbered benign ones. In real-world scenarios, benign software is much more common, and a more realistic distribution of samples in the test set could improve the classifier's performance. Despite this limitation, the results indicate that the proposed methodology has potential as a viable approach for malware detection.

The findings related to potential malware signatures in the outputted probabilities suggest that a methodology for multi-family classification could be developed using a decoder-based transformer architecture. This approach could be stand-alone or complement the current methodology by resolving edge cases, such as samples with ACD values near the decision threshold. Alternatively, the transformer could be used to generate signatures for feeding into other models. This could represent a promising direction for future research.

In terms of real-time detection, the proposed methodology offers efficiency and practicality, as it uses unprocessed malware binaries directly. Running large samples through the model was not feasible due to computational limitations. However, several optimizations could be implemented, such as evaluating batches of bytes instead of individual bytes, which were not pursued due to time constraints. Due to the computationally demanding nature of neural networks, the resulting approach could still pose challenges for devices with limited single-core performance. In such cases, remote computation could be utilized, provided the devices are equipped with a fast network connection to a more powerful node, such as a cloud-based infrastructure.

Bibliography

[1] A. Vaswani, G. Brain, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, vol. 30, Long Beach, CA, USA, Dec. 2017.

[2] M. Souppaya and K. Scarfone, *Guide to Malware Incident Prevention and Handling for Desktops and Laptops*, NIST Std. SP 800-83 Rev. 1, Jul. 2013, [accessed Oct. 9, 2024]. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-83r1.pdf

[3] O. Aslan and R. Samet, "A comprehensive review on malware detection approaches," *IEEE Access*, vol. 8, pp. 6249–6271, Jan. 2020.

[4] S. S. Chakkaravarthy, D. Sangeetha, and V. Vaidehi, "A survey on malware analysis and mitigation techniques," *Computer Science Review*, vol. 32, pp. 1–23, May 2019.

[5] M. Willett, "The cyber dimension of the russia–ukraine war," *Survival*, vol. 64, pp. 7–26, Oct. 2022.

[6] "2024 mid-year cyber threat report," SonicWall Inc., 2024, [accessed Oct. 9, 2024]. [Online]. Available: https://www.sonicwall.com/resources/white-papers/mid-year-2024-sonicwall-cyber-threat-report

[7] "2023 internet crime report," Federal Bureau of Investigation (FBI), Mar. 2023, [accessed Nov. 29, 2024]. [Online]. Available: https://www.ic3.gov/AnnualReport/Reports/2023_IC3Report.pdf

[8] "Cost of a data breach report 2024," IBM, [accessed Oct. 9, 2024]. [Online]. Available: https://www.ibm.com/reports/data-breach

[9] S. Morgan, "2023 cybersecurity almanac: 100 facts, figures, predictions, and statistics," Cybersecurity Ventures, May 2023, [accessed Oct. 9, 2024]. [Online]. Available: https://cybersecurityventures.com/cybersecurity-almanac-2023/

[10] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach, "Dynamic malware analysis in the modern era—a state of the art survey," *ACM Computing Surveys*, vol. 52, p. 1–48, Sep. 2019.

[11] F. A. Aboaoja, A. Zainal, F. A. Ghaleb, B. A. S. Al-rimy, T. A. E. Eisa, and A. A. H. Elnour, "Malware detection issues, challenges, and future directions: A survey," *Applied Sciences*, vol. 12, Aug. 2022.

[12] Z. Akhtar. (2021, Jan.) Malware detection and analysis: Challenges and research opportunities. arXiv. [Online]. Available: https://arxiv.org/abs/2101.08429

[13] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. [Online]. Available: https://www.deeplearningbook.org

[14] A. K. Jha, A. Vaish, and S. Patil, "A novel framework for metamorphic malware detection," *SN Computer Science*, vol. 4, Oct. 2023.

[15] A. Johnson, "The devastating effect of polymorphic malware," Cybersecurity Ventures, Apr. 2020, [accessed Oct. 9, 2024]. [Online]. Available: https://cybersecurityventures.com/the-devastating-effect-of-polymorphic-malware/

[16] A. A. Al-Hashmi, F. A. Ghaleb, A. Al-Marghilani, A. E. Yahya, S. A. Ebad, M. S. M. Saqib, and A. A. Darem, "Deep-ensemble and multifaceted behavioral malware variant detection model," *IEEE Access*, vol. 10, pp. 42 762–42 777, Apr. 2022.

[17] A. G. Kakisim, M. Nar, and I. Sogukpinar, "Metamorphic malware identification using engine-specific patterns based on co-opcode graphs," *Computer Standards & Interfaces*, vol. 71, Aug. 2020.

[18] P. Desai and M. Stamp, "A highly metamorphic virus generator," *International Journal of Multimedia Intelligence and Security*, vol. 1, pp. 402–427, 2010.

[19] N. K. Gyamfi, N. Goranin, D. Ceponis, and H. A. Čenys, "Automated system-level malware detection using machine learning: A comprehensive review," *Automated System-Level Malware Detection Using Machine Learning: A Comprehensive Review*, vol. 13, no. 21, Oct. 2023.

[20] C. S. Collberg, C. D. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Technical Report 148, Department of Computer Science, University of Auckland, Tech. Rep., Jul. 1997.

[21] B. Dang, A. Gazet, E. Bachaalany, and S. Josse, *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*. Hoboken, NJ, USA: Wiley, 2014.

[22] I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in *Proceedings of the International Conference on Broadband, Wireless Computing Communication and Applications*, Fukuoka, Japan, Nov. 2010, pp. 297–300.

[23] P. O'Kane, S. Sezer, and K. McLaughlin, "Obfuscation: The hidden malware," *IEEE Security and Privacy*, vol. 9, pp. 41–47, Sep./Oct. 2011.

[24] E. M. Rudd, A. Rozsa, M. Günther, and T. E. Boult, "A survey of stealth malware attacks, mitigation measures, and steps toward autonomous open world solutions," *IEEE Communications Surveys and Tutorials*, vol. 19, pp. 1145–1172, Dec. 2016.

[25] P. Szor, *The Art of Computer Virus Research and Defense*. Upper Saddle River, NJ, USA: Pearson Education, 2005.

[26] K. Hahn, "Robust static analysis of portable executable malware," Master's thesis, Leipzig University of Applied Sciences, Leipzig, Germany, Dec. 2014.

[27] Z. Bazrafshan, H. Hashemi, S. M. H. Fard, and A. Hamzeh, "A survey on heuristic malware detection techniques," in *Proceedings of the 5th Conference on Information and Knowledge Technology*, Shiraz, Iran, May 2013, pp. 113–120.

[28] J. Newsome, B. Karp, and D. Song, "Polygraph: Automatically generating signatures for polymorphic worms," in *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2005, pp. 226–241.

[29] Y. Tang, B. Xiao, and X. Lu, "Using a bioinformatics approach to generate accurate exploit-based signatures for polymorphic worms," *Computers & Security*, vol. 28, pp. 827–842, Nov. 2009.

[30] V. Naidu, J. Whalley, and A. Narayanan, "Exploring the effects of gap-penalties in sequence-alignment approach to polymorphic virus detection," *Journal of Information Security*, vol. 8, pp. 296–327, Nov. 2017.

[31] M. Du, W. Hu, and W. Hewlett, "Autocombo: Automatic malware signature generation through combination rule mining," in *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, Queensland, Australia, Oct. 2021, pp. 3777–3786.

[32] J. L. Alvares, "Malware classification with BERT," Master's thesis, San José State University, San José, CA, USA, May 2021.

[33] A. Chandak, W. Lee, and M. Stamp, "A comparison of Word2Vec, HMM2Vec, and PCA2Vec for malware classification," in *Malware Analysis Using Artificial Intelligence and Deep Learning*, M. Stamp, M. Alazab, and A. Shalaginov, Eds. Cham, Switzerland: Springer International Publishing, Dec. 2020, pp. 287–320.

[34] W. Wong and M. Stamp, "Hunting for metamorphic engines," *Journal in Computer Virology*, vol. 2, pp. 211–229, Dec. 2006.

[35] T. H. Austin, E. Filiol, S. Josse, and M. Stamp, "Exploring hidden markov models for virus analysis: A semantic approach," in *Proceedings of the 46th Hawaii International Conference on System Sciences*, Wailea, HI, USA, Jan. 2013, pp. 5039–5048.

[36] A. H. Toderici and M. Stamp, "Chi-squared distance and metamorphic virus detection," *Journal in Computer Virology*, vol. 9, pp. 1–14, Feb. 2013.

[37] Y. T. Ling, N. F. M. Sani, M. T. Abdullah, and N. A. W. A. Hamid, "Metamorphic malware detection using structural features and nonnegative matrix factorization with hidden markov model," *Journal of Computer Virology and Hacking Techniques*, vol. 18, pp. 183–203, Sep. 2022.

[38] H. D. Menéndez, S. Bhattacharya, D. Clark, and E. T. Barr, "The arms race: Adversarial search defeats entropy used to detect malware," *Expert Systems with Applications*, vol. 118, pp. 246–260, Mar. 2019.

[39] J. Singh and J. Singh, "A survey on machine learning-based malware detection in executable files," *Journal of Systems Architecture*, vol. 112, Jan. 2021.

[40] J. Hegedus, Y. Miche, A. Ilin, and A. Lendasse, "Methodology for behavioral-based malware analysis and detection using random projections and k-nearest neighbors classifiers," in *Proceedings of the 7th International Conference on Computational Intelligence and Security*, Sanya, China, Dec. 2011, pp. 1016–1023.

[41] A. Mohaisen, O. Alrawi, and M. Mohaisen, "Amal: High-fidelity, behavior-based automated malware analysis and classification," *Computers & Security*, vol. 52, pp. 251–266, Jul. 2015.

[42] H. R. Borojerdi and M. Abadi, "Malhunter: Automatic generation of multiple behavioral signatures for polymorphic malware detection," in *Proceedings of the 3rd International Conference on Computer and Knowledge Engineering*, Mashhad, Iran, Oct./Nov. 2013, pp. 430–436.

[43] S. Alqurashi and O. Batarfi, "Automated malware detection based on HMM and genetic K-means," *International Journal of Intelligent Computing Research*, vol. 8, pp. 849–861, Sep. 2017.

[44] P. Maniriho, A. N. Mahmood, and M. J. M. Chowdhury. (2022, Sep.) Maldetconv: Automated behaviour-based malware detection framework based on natural language processing and deep learning techniques. arXiv. [Online]. Available: http://arxiv.org/abs/2209.03547

[45] Y. Ye, T. Li, D. Adjeroh, and S. S. Iyengar, "A survey on malware detection using data mining techniques," *ACM Computing Surveys*, vol. 50, pp. 1–40, Jun. 2017.

[46] D. Komashinskiy and I. Kotenko, "Malware detection by data mining techniques based on positionally dependent features," in *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, Pisa, Italy, Feb. 2010, pp. 617–623.

[47] R. Islam, R. Tian, L. M. Batten, and S. Versteeg, "Classification of malware based on integrated static and dynamic features," *Journal of Network and Computer Applications*, vol. 36, pp. 646–656, Apr. 2013.

[48] J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features," in *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, Fajardo, PR, USA, Oct. 2015, pp. 11–20.

[49] R. Vinayakumar, M. Alazab, K. P. Soman, P. Poornachandran, and S. Venkatraman, "Robust intelligent malware detection using deep learning," *IEEE Access*, vol. 7, pp. 46 717–46 738, Apr. 2019.

[50] P. Shaw, J. Uszkoreit, and A. Vaswani. (2018, Mar.) Self-attention with relative position representations. arXiv. [Online]. Available: http://arxiv.org/abs/1803.02155

[51] J. Cheng, L. Dong, and M. Lapata. (2016, Jan.) Long short-term memory-networks for machine reading. arXiv. [Online]. Available: http://arxiv.org/abs/1601.06733

[52] S. Edunov, M. Ott, M. Auli, and D. Grangier. (2018, Aug.) Understanding back-translation at scale. arXiv. [Online]. Available: http://arxiv.org/abs/1808.09381

[53] X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang, and Q. Liu. (2019, Sep.) Tinybert: Distilling bert for natural language understanding. arXiv. [Online]. Available: http://arxiv.org/abs/1909.10351

[54] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. (2018) Improving language understanding by generative pre-training. OpenAI. [accessed Oct. 9, 2024]. [Online]. Available: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

[55] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. (2018, Oct.) Bert: Pre-training of deep bidirectional transformers for language understanding. arxiv. [Online]. Available: http://arxiv.org/abs/1810.04805

[56] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. Mccandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, vol. 33, Vancouver, BC, Canada, Dec. 2020.

[57] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. (2019, Jul.) Roberta: A robustly optimized bert pretraining approach. arxiv. [Online]. Available: http://arxiv.org/abs/1907.11692

[58] B. Lakha, S. L. Mount, E. Serra, and A. Cuzzocrea, "Anomaly detection in cybersecurity events through graph neural network and transformer based model: A case study with beth dataset," in *Proceedings of the 2022 IEEE International Conference on Big Data*, Osaka, Japan, Dec. 2022, pp. 5756–5764.

[59] V. Sanh, L. Debut, J. Chaumond, and T. Wolf. (2019, Oct.) Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. arxiv. [Online]. Available: http://arxiv.org/abs/1910.01108

[60] M. Gopinath and S. C. Sethuraman, "A comprehensive survey on deep learning based malware detection techniques," *Computer Science Review*, vol. 47, Feb. 2023.

[61] D. Gibert and J. Bejar, "Convolutional neural networks for malware classification," Master's thesis, Universitat Politècnica de Catalunya, Barcelona, Spain, Oct. 2016.

[62] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi. (2018, Feb.) Microsoft malware classification challenge. arxiv. [Online]. Available: http://arxiv.org/abs/1802.05365

[63] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath, "Malware images: Visualization and automatic classification," in *Proceedings of the 8th International Symposium on Visualization for Cyber Security*, Pittsburgh, Pennsylvania, USA, Jul. 2011, pp. 1–7.

[64] Y. Kim. (2014, Sep.) Convolutional neural networks for sentence classification. arxiv. [Online]. Available: https://arxiv.org/abs/1408.5882

[65] X. Xusheng and S. Yang, “An image-inspired and cnn-based android malware detection approach,” in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, San Diego, CA, USA, Nov. 2019, pp. 1259–1261.

[66] S. Jha, D. Prashar, H. V. Long, and D. Taniar, “Recurrent neural network for detecting malware,” *Computers and Security*, vol. 99, Dec. 2020.

[67] M. S. Akhtar and T. Feng, “Detection of malware by deep learning as cnn-lstm machine learning techniques in real time,” *Symmetry*, vol. 14, Nov. 2022.

[68] X. Xiao, S. Zhang, F. Mercaldo, G. Hu, and A. K. Sangaiah, “Android malware detection based on system call sequences and lstm,” *Multimedia Tools and Applications*, vol. 78, pp. 3979–3999, Feb. 2019.

[69] A. Rahali and M. A. Akhloufi. (2021, Mar.) Malbert: Using transformers for cybersecurity and malicious software detection. arxiv. [Online]. Available: http://arxiv.org/abs/2103.03806

[70] A. S. Kale, V. Pandya, F. D. Troia, and M. Stamp, “Malware classification with Word2Vec, HMM2Vec, BERT, and ELMo,” *Journal of Computer Virology and Hacking Techniques*, Apr. 2022.

[71] M. Q. Li, B. C. Fung, P. Charland, and S. H. Ding, “I-mad: Interpretable malware detector using galaxy transformer,” *Computers and Security*, vol. 108, Sep. 2021.

[72] N. Şahin, “Malware detection using transformers-based model gpt-2,” Master’s thesis, Middle East Technical University, Ankara, Turkey, 2021.

[73] D. Demirci, N. Sahin, M. Sirlancis, and C. Acarturk, “Static malware detection using stacked bilstm and gpt-2,” *IEEE Access*, vol. 10, pp. 58 488–58 502, May 2022.

[74] S. Seneviratne, R. Shariffdeen, S. Rasnayaka, and N. Kasthuriarachchi, “Self-supervised vision transformers for malware detection,” *IEEE Access*, vol. 10, Sep. 2022.

[75] K.-W. Park and S.-B. Cho, “A vision transformer enhanced with patch encoding for malware classification,” in *Intelligent Data Engineering and Automated Learning – IDEAL 2022*, Manchester, UK, Nov. 2022, pp. 289–299.

[76] A. Karpathy, Jun. 2023, [accessed Oct. 9, 2024]. [Online]. Available: https://github.com/karpathy/nanoGPT

[77] I. George-Andrei, 2021, [accessed Oct. 9, 2024]. [Online]. Available: https://github.com/iosifache/DikeDataset

[78] A. Nappa, M. Z. Rafique, and J. Caballero, “Driving in the cloud: An analysis of drive-by download operations and abuse reporting,” in *Detection of Intrusions and Malware, and Vulnerability Assessment*, Berlin, Heidelberg, Nov. 2013, pp. 1–20.

[79] A. P. Tuan, A. T. H. Phuong, N. V. Thanh, and T. N. Van, 2018, [accessed Oct. 9, 2024]. [Online]. Available: https://figshare.com/articles/dataset/Malware_Detection_PE-Based_Analysis_Using_Deep_Learning_Algorithm_Dataset/6635642?file=12149696

[80] K. P. Murphy, *Machine learning: a probabilistic perspective*. Cambridge, MA, USA: MIT Press, 2012.