PhD-FSTM-2024-100
The Faculty of Science, Technology and Medicine

Dissertation

Defense held on the 11^{th} of December, 2024 in Luxembourg

to obtain the degree of

Docteur de l'Université du Luxembourg en Informatique

by

Tiezhu SUN
Born on the 19^{th} of September, 1993 in Shandong, China

Boosting Android Malware Learning

Dissertation Defense Committee

Dr. Jacques Klein, Dissertation Supervisor
Professor, University of Luxembourg

Dr. Tegawendé F. Bissyandé, Chairman
Professor, University of Luxembourg

Dr. Kevin Allix, Vice Chairman
Senior Security DevOps, Freelancer

Dr. Djamel Khadraoui, Member
Senior Research Scientist, Luxembourg Institute of Science and Technology

Dr. Daniel Arp, Member
Professor, Technische Universität Wien

Abstract

The ubiquity of Android devices in today's digital ecosystem has made them a prime target for cybercriminals. As Android malware becomes increasingly sophisticated, the need for advanced malware learning techniques to enhance Android malware analysis—including malware detection, malware family classification, and malicious code localization—has become more critical. While recent advancements in machine learning and deep learning have significantly enhanced Android malware analysis, many existing methods still rely on low-level bytecode, handcrafted features, or function call graphs that often fall short in capturing complex malicious behaviors. In this thesis, we propose advanced and original techniques aimed at achieving comprehensive, accurate, and robust Android malware learning, thereby contributing to software security through novel learning methodologies and providing deeper insights into malware behavior.

In the first part of this thesis, building upon the effective image representation method DexRay, we investigate under-explored artifacts beyond the most studied Dalvik bytecode to create more comprehensive representations of Android apps. Our research reveals that in addition to Dalvik bytecode, binary native code and Manifest files also offer rich and relevant information for Android malware detection, with each artifact contributing unique insights. Although combining these three artifacts provides slight improvements in detection performance, Dalvik bytecode alone remains the most efficient and effective option, making it the preferred choice for Android malware detection.

In the second part, recognizing the limitations of existing bytecode learning techniques, such as DexRay's grayscale "vector" images, in fully capturing the semantic information of malware behaviors from Dalvik bytecode, we introduce DexBERT, a pre-trained model built on the BERT architecture. DexBERT processes disassembled Smali code from Dalvik bytecode to produce embeddings that capture nuanced malicious behaviors, enabling fine-grained malware analysis, including class-level maliciousness localization. Remarkably, DexBERT not only excels in this task but also in other class-level Android learning tasks such as defect detection and component type classification, establishing it as a versatile representation learning model for Android bytecode. However, for app-level tasks like Android malware detection, additional embedding aggregation is required.

To address this challenge, our third contribution, LaFiCMIL, tackles the problem of classifying large files from the perspective of correlated multiple instance learning (c-MIL). Rather than designing an Android-specific solution to address this issue, we propose a versatile framework that can be applied to general BERT-like models to handle large file processing efficiently. We initially develop this method using a standard BERT model and validate its effectiveness on typical large file classification

tasks, such as long document classification and code defect detection. Building on this framework, we developed DetectBERT to harness DexBERT's representation learning capabilities for app-level malware detection through c-MIL. DetectBERT analyzes collective class behavior to identify Android malware by recognizing patterns that emerge from the correlations among classes. Evaluations on a large-scale benchmark dataset demonstrate that DetectBERT achieves new state-of-the-art performance in Android malware detection.

In the third part, we explore a critical challenge beyond Android malware detection: Android malware family classification in a real-world temporal-incremental scenario. Specifically, we address concept drift caused by the emergence of new malware families and shifts in the data distribution of existing families. We formulate this problem as Temporal-Incremental Malware Learning (TIML), and adapt state-of-the-art Class-Incremental Learning (CIL) approaches to meet the specific requirements of TIML. We also propose a novel multimodal TIML approach harnessing rich information from multiple malware modalities. Extensive evaluations demonstrate its significant superiority over original adapted CIL approaches. These findings reveal the feasibility of periodically updating malware classifiers at a minimal cost to address concept drift, establishing a new benchmark for robust and adaptive Android malware detection.

In conclusion, the methodologies and models proposed in this thesis not only boost Android malware learning but also open new avenues for research in the broader field of malware detection and classification. The insights gained from this work contribute to a more secure Android ecosystem by enabling more sophisticated and adaptable defenses against evolving threats.

Acknowledgements

I would like to express my deepest gratitude to all the individuals who made this dissertation possible. Many have contributed directly by sharing their invaluable knowledge, advice, and experience, while others provided daily support that helped me throughout this journey.

First and foremost, I extend my heartfelt thanks to my supervisor, Prof. Jacques Klein, for his unwavering trust and for giving me the opportunity to pursue my Ph.D. with renowned researchers. His guidance and belief in me have been instrumental in the success of this endeavor.

I am also deeply thankful to my co-supervisor, Prof. Tegawendé F. Bissyandé, and my daily advisor, Dr. Kevin Allix, for their continuous support and precious insights, which were vital in shaping and conducting my research. Their expertise and encouragement were invaluable throughout my Ph.D. journey.

I express my sincere gratitude to Prof. Lorenzo Cavallaro, who graciously agreed to serve as my CET member. His valuable feedback during each CET meeting has significantly contributed to the development of my work.

I am also grateful to my coauthors, including Prof. David Lo, Prof. Dongsun Kim, Dr. Nadia Daoudi, Dr. Kisub Kim, Weiguo Pian and Xin Zhou, for their productive discussions and valuable collaborations, which have enriched my research experience.

I would like to extend my appreciation to the members of my Ph.D. defense committee: Chairman Prof. Tegawendé F. Bissyandé, Vice-Chairman Dr. Kevin Allix, Prof. Djamel Khadraoui, Prof. Daniel Arp, Dr. Nadia Daoudi, and my supervisor Prof. Jacques Klein. It is a great honor to have such a distinguished panel evaluate my work and I sincerely appreciate their time, effort, and thoughtful examination of my dissertation. Their feedback and evaluation have been invaluable in shaping the final outcome of my Ph.D. work. I would like to express my gratitude to the Luxembourg National Research Fund (FNR), the main financial contributor that funded this thesis through an AFR grant.

With great pleasure, I express my gratitude to all my colleagues and friends in the TruX research group who have made my journey enjoyable and memorable. My heartfelt appreciation also goes out to my friends in Luxembourg who accompanied me during this journey and contributed to many cherished memories.

Words cannot fully capture my gratitude to my wife, Xiaoyan Xie, whose unwavering love, trust, support, and encouragement over the past 15 years have been the foundation of my strength. She has made our family a source of joy, filling our home with warmth, comfort, laughter, and harmony. Her steadfast presence has given me the peace of mind necessary to pursue and achieve this goal. I am also deeply grateful for my wonderful son, Mingjue Sun, whose birth and presence have brought

renewed vitality and purpose to my life.

Lastly, I wish to express my profound gratitude to my father, Shengqiang Sun, my mother, Cuizhi Lu, and my sister, Yang Sun, for their unconditional love and support throughout my life. Their belief in me has been a constant source of inspiration, and without their unwavering support, I would not have been able to achieve such a significant accomplishment.

Tiezhu Sun
University of Luxembourg
December 2024

Contents

List of Figures

List of Tables

1

Introduction

In this chapter, we first provide an overview of Android devices and their susceptibility to malware. Next, we introduce the problems our research addresses and summarize the key challenges in Android malware learning, particularly in combating evolving threats. Finally, we present the contributions of our work and outline the roadmap of this dissertation.

Contents

1.1 Overview of Android Devices

Android, developed by Google, has emerged as the most widely used mobile operating system in the world. As show in Figure 1.1, Android powers approximately 71.67% of smartphones globally as of 2024, with over 3.9 billion active devices [4]. The operating system's success can be attributed to its open-source nature, which has fostered widespread adoption by manufacturers and developers. Android is used in a variety of devices beyond smartphones, including tablets, wearables, smart TVs, and connected devices within the Internet of Things (IoT) ecosystem. This broad reach highlights Android's importance in the global mobile and smart device market.

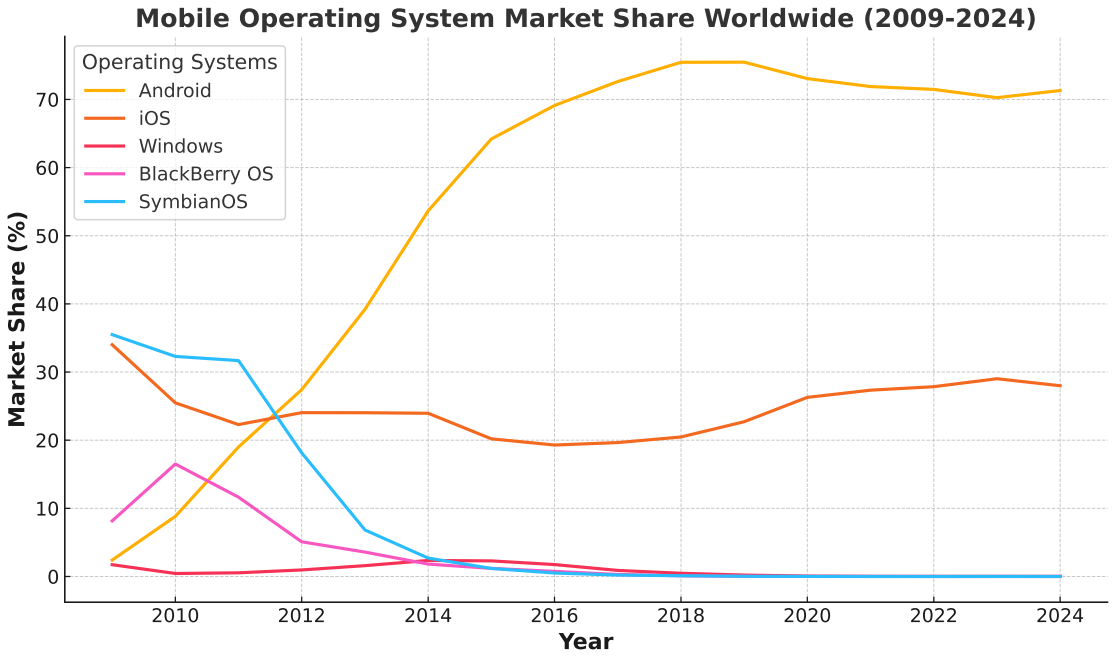

Figure 1.1: Trends in the mobile operating system market share from 2009 to 2024 [1]. The line chart illustrates the market share trends for major mobile operating systems, including Android, iOS, Windows, BlackBerry OS, and SymbianOS, over the period from 2009 to 2024. Android exhibits a significant rise and currently dominates the market.

Despite its widespread use, Android's open ecosystem also makes it a major target for cyberattacks. The platform's flexibility, which allows developers to distribute applications through both the official Google Play Store and third-party app stores, introduces significant security risks. Third-party app stores, in particular, often lack rigorous security checks, making them a common vector for distributing malware. This has resulted in a significantly high number of reported malware incidents. According to AV-ATLAS [2], the total number of Android malware samples surpassed 35 million in 2024, as shown in Figure 1.2, highlighting the growing scale of this threat.

The diversity of Android devices and their integration into various aspects of daily life further exacerbate the security challenges. Android users are exposed to a wide range of malware types [5], including adware, ransomware, spyware, and banking trojans, which often masquerade as legitimate applications. These malicious apps

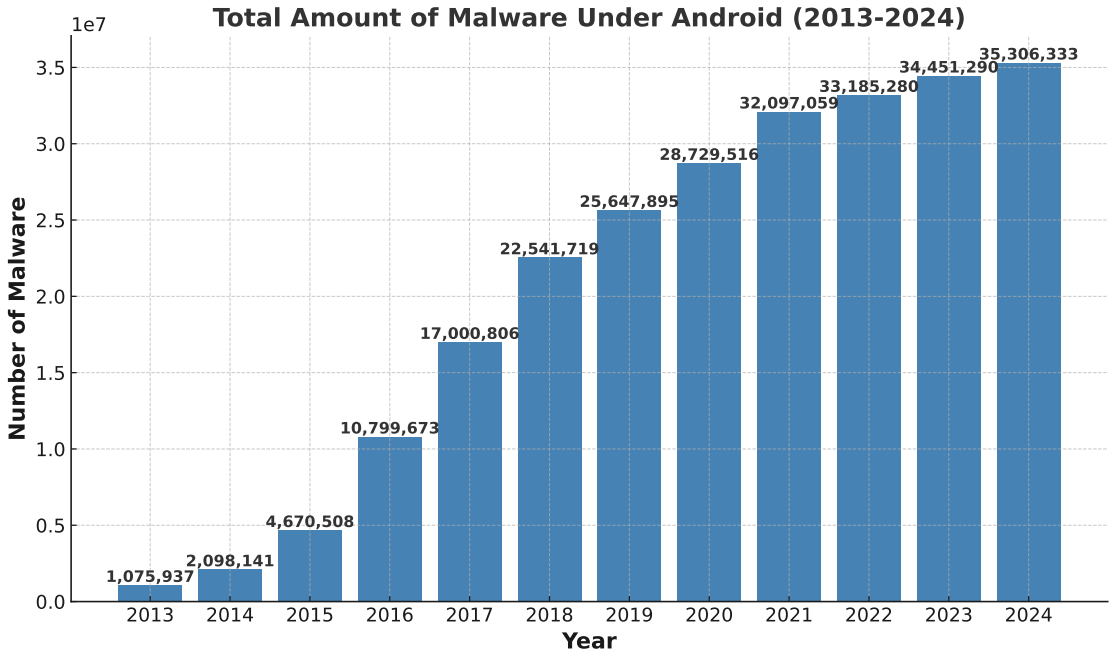

Figure 1.2: Growth of Android Malware (2013-2024) [2]. The bar chart shows the increasing number of malware instances targeting Android devices, rising from over 1 million in 2013 to more than 35 million in 2024.

can request excessive permissions, enabling attackers to gain unauthorized access to sensitive data or take control of the device.

Another factor contributing to Android's security challenges is the issue of fragmentation [6]. The operating system is used across a variety of devices from different manufacturers, many of which run outdated versions of Android. This fragmentation leads to delays in security updates, leaving millions of devices vulnerable to known exploits. While Google has taken steps to mitigate these risks through initiatives like Project Treble [7] and security patches via Google Play Services [8], many devices remain at risk due to slow update cycles from manufacturers.

In addition to the risks posed by mobile applications, Android's growing presence in the IoT ecosystem further expands its attack surface. Android-powered devices such as smart home systems, wearables, and automotive interfaces often lack the same level of security as smartphones, making them attractive targets for cybercriminals [9]. As Android continues to dominate the mobile landscape and expand into new areas, ensuring the security of the platform remains a critical challenge.

In summary, while Android's open-source nature and extensive market presence have driven its success, these same factors have contributed to its susceptibility to malware. The increasing number of malware samples and the evolving sophistication of these threats underline the need for more robust detection and prevention mechanisms. This dissertation addresses these challenges by proposing advanced malware learning techniques to improve Android malware analysis, focusing on malware detection, malware family classification, and the localization of malicious code within applications.

1.2 Problem Statement

Malware, short for "malicious software", refers to any software intentionally designed to cause damage, disrupt operations, or gain unauthorized access to devices, networks, or data [10]. Malware encompasses a wide range of threats [5, 11], including viruses, worms, ransomware, spyware, and trojans, etc. As these threats have evolved, so too have the methods used to detect and mitigate them [12]. In the context of mobile devices, particularly those running the Android operating system, malware poses a significant and growing threat. Android malware is designed to employ specific techniques that make an application perform actions it is not intended to do [13, 7], often with the objective of stealing sensitive information, spying or impersonating users, or taking control of devices.

The rapid proliferation of Android malware [2] has driven the need for more sophisticated detection and mitigation techniques. This dissertation focuses on **Android malware learning**, a broad term that encompasses several key areas of research aimed at improving the security of Android devices. Android malware learning refers to the application of machine learning, deep learning and data-driven approaches to understand, detect, classify, and localize the behavior of malware on Android platforms.

At the core of Android malware learning is **representation learning**, which seeks to transform raw data from Android applications into meaningful features that can be used by learning-based models. Effective representation learning is crucial for capturing the intricate characteristics of Android applications (APKs) and distinguishing between benign and malicious behaviors. Android applications can be represented using different artifacts extracted from the APK [14, 15], including Dalvik bytecode, manifest files, native code, and more. They can also be analyzed in various ways [16, 17, 18, 19], such as through their permissions, API calls, network behavior, automatically learned features, and other characteristics. The challenge lies in developing representations that accurately capture the essence of malicious activity while minimizing noise from benign behaviors and staying adaptive to the constant evolution of malware.

Malware detection is one of the primary goals of Android malware learning. Traditional detection approaches, such as signature-based methods [20, 21], are becoming less effective against modern malware due to the increasing use of obfuscation and the rapid evolution of new threats. As malware variants are continuously developed and deployed, more adaptive and intelligent methods are required. This need gave rise to machine learning-based malware detection [22, 23, 24], which leverages features extracted from Android apps to train models capable of identifying previously unseen malware samples. More recently, deep learning-based techniques [25, 18, 19] have shown great promise, utilizing neural networks to automatically learn complex patterns and extract features from raw data without the need for manual feature engineering. Despite these advancements, designing robust detection systems that can effectively address the diversity and complexity of Android malware remains a significant challenge.

A critical aspect of Android malware learning is **malicious code localization**. It involves identifying and isolating the specific portions of code within an Android application that are responsible for malicious behavior [26]. Given the size and complexity of modern Android apps, this is a non-trivial task. Effective code

localization allows security analysts to pinpoint the exact source of malicious actions within an app, aiding in both detection and mitigation efforts. By narrowing down the problematic sections of code, we can improve the precision of malware analysis and reduce the need for extensive manual inspection. However, this problem remains underexplored due to its inherent difficulty and the lack of comprehensive benchmark datasets for evaluation.

Classification of malware into families or types is another important aspect of malware learning [27, 28]. Once malware is detected, it is often useful to determine its classification to understand its behavior, origin, and potential impact. Android malware classification aims to group malicious samples into categories based on shared characteristics. This helps security professionals identify patterns of attacks, respond more effectively, and mitigate similar threats in the future. However, classifying malware is complicated by the sheer number of malware variants, many of which share overlapping features or attempt to mimic benign applications.

Malware evolves over time, requiring models to be updated regularly to account for new threats. **Temporal-incremental malware learning** involves continuously updating malware detection and classification systems to adapt to new malware samples as they emerge. This is particularly important in the Android ecosystem, where new apps are constantly being developed and new malware strains frequently emerge. Traditional machine learning models often struggle to maintain performance when confronted with new data that was not part of the original training set. Temporal-incremental learning techniques aim to bridge this gap by updating models incrementally, without the need for retraining from scratch. However, this problem is still underexplored, largely due to the challenges of designing efficient algorithms and the scarcity of suitable benchmark datasets.

In summary, the problem addressed in this dissertation is the increasing difficulty of detecting, classifying, and mitigating Android malware due to its rapid evolution and the complexity of the Android ecosystem. To address this, advanced malware learning techniques—including representation learning, detection, classification, malicious code localization, and temporal-incremental learning—are essential. These approaches not only enhance the accuracy and adaptability of malware detection systems but also provide deeper insights into malware behavior, enabling more proactive and scalable defense mechanisms for Android security.

1.3 Challenges in Android Malware Learning

Android malware learning presents several significant challenges that hinder the development of robust and effective detection and classification systems. These challenges span across various aspects, including the availability and quality of datasets, the complexity of representation learning, the issue of concept drift in malware evolution, and the interpretability of detection models. In this section, we will elaborate on these key challenges.

1.3.1 Datasets

A major limitation in Android malware learning is the scarcity of comprehensive datasets with high-quality labels (or ground-truths). The AndroZoo repository [29] has made substantial progress by collecting a large-scale dataset of millions of APKs from various sources, along with corresponding metadata. For each APK, AndroZoo provides a VirusTotal report, which includes the number of antivirus products that

flagged the APK as malicious. Researchers often use this data to assign binary labels by setting a threshold based on the number of antivirus detections. However, this approach poses a risk to label quality, as it may lead to inconsistencies or misclassifications due to false positives or false negatives from the antivirus engines.

Existing malware detection datasets either use a small portion of Androzoo APKs or are derived from other sources [22, 30] that are often smaller in scale and less diverse. This lack of a widely accepted benchmark dataset poses a critical challenge for researchers aiming to compare their methods with others in a consistent manner. For malware classification, the problem is even more pronounced, with fewer available datasets and much of the existing labeled data suffering from poor quality due to limited manual analysis and the ambiguity in defining malware types or families. The quality of these labels directly impacts the reliability of learning-based models trained on such data.

Moreover, the challenges are amplified when it comes to tasks like malicious code localization and temporal-incremental malware learning. For malicious code localization, the difficulty in obtaining granular, manually labeled datasets results in a severe lack of benchmark data. Temporal-incremental learning, which requires datasets that evolve over time, also faces similar constraints, as few datasets capture the time-based evolution of malware in a systematic way. The absence of such benchmark datasets slows progress in both research and practical applications of Android malware learning.

1.3.2 Representation Learning

Representation learning, the process of extracting and selecting useful features from APKs for learning-based models, is another critical challenge. Android applications contain a wide variety of artifacts, such as Dalvik bytecode, manifest files, native code, and more. However, many of these artifacts, especially native code, have not been sufficiently explored in existing research, leading to incomplete representations of the malware. Capturing the full complexity of an APK is particularly difficult due to the sheer size and diversity of modern Android applications.

Manually crafted features [20, 21, 22], which are often tailored to specific aspects of malware behavior, are time-consuming and expensive to develop. Moreover, they may not generalize well to new types of malware, especially as malicious actors employ increasingly sophisticated obfuscation techniques. While recent efforts have shifted towards using learning-based features, such as those based on bytecode images [19] and function call graphs [18], these methods often lack the semantic richness necessary to accurately represent the underlying malicious behaviors.

A major challenge lies in designing representation learning techniques that can capture both the static and dynamic aspects of Android malware. Without effective and comprehensive feature extraction methods, learning-based models may miss important patterns or overfit to irrelevant features, reducing their ability to generalize to unseen malware samples.

1.3.3 Concept Drift

Concept drift [31, 32] is a fundamental challenge in Android malware learning due to the constantly evolving nature of malware. Concept drift refers to the phenomenon where the statistical properties of the target variable change over time, requiring models to adapt continuously. In the context of Android malware, concept drift

manifests in two primary ways:

- **The emergence of new malware families or types**: New malware variants are constantly being created, each with distinct characteristics that may not be captured by models trained on older datasets.
- **Shifts in the data distribution of existing malware families or types**: Even within established malware families, the methods and behaviors employed by attackers can change, leading to shifts in the underlying data distribution. These changes make it difficult for static models to maintain accuracy over time.

Handling concept drift requires models that can not only detect when the data distribution has changed but also adapt incrementally to account for new types of malware without needing to be retrained from scratch. This is especially challenging given the high volume of malware variants that emerge daily and the limited access to continuously updated datasets.

1.3.4 Interpretability

While machine learning and advanced deep learning models have shown impressive results in malware detection, interpretability remains a significant challenge. Interpretability refers to the ability to understand and explain how a model arrives at its decisions. In the context of Android malware learning, interpretability is crucial for several reasons:

- **Security Analysts**: Security professionals need to understand why a particular application was classified as malicious or benign in order to validate the detection results and take appropriate countermeasures.
- **Regulatory Compliance**: In some industries, particularly those governed by data privacy laws, interpretable models are necessary to ensure that decisions made by automated systems are transparent and explainable.
- **Debugging and Improvement**: If a model's decision-making process is opaque, it is difficult to debug errors, improve the model, or build trust in its predictions.

However, models like deep neural networks are often treated as "black boxes", making their decisions difficult to interpret. Striking a balance between performance and interpretability remains a critical challenge. While some methods, such as attention mechanisms or feature importance scores, have been proposed to improve interpretability, they are not always sufficient in the context of highly complex malware datasets.

1.4 Research contributions

This dissertation introduces advanced and original techniques aimed at achieving comprehensive, accurate, and robust Android malware learning. The goal is to enhance software security by developing novel learning methodologies and providing deeper insights into malware behavior.

In the first part, building upon the effective image representation method DexRay, which focuses on the artifact Dalvik bytecode, we explore other under-explored artifacts to create more comprehensive representations of Android apps for malware detection. Specifically, we make the following contributions:

- *Android Malware Detection: Looking beyond Dalvik Bytecode*: We postulate that other artifacts, such as the binary (native) code and metadata/configura-

tion files, can be leveraged at to build more comprehensive representations of Android applications. Through extensive experimental evaluation, we demonstrate that binary code and metadata files also provide relevant information for Android malware detection. In addition, they enable the detection of malware that may not be identified by models built solely on Dalvik bytecode. We further investigate the potential advantages of combining these artifacts into a unified representation to enhance the signal for detecting malicious behavior. Although combining these three artifacts provides slight improvements in detection performance, Dalvik bytecode alone remains the most efficient and effective option, making it the preferred choice for Android malware detection.
This work resulted in a research paper published at the International Workshop on Advances in Mobile App Analysis (A-mobile@ASE) in 2021

Recognizing the limitations of existing bytecode representation techniques, such as DexRay's grayscale "vector" images, in fully capturing the semantic details of malware behaviors from Dalvik bytecode, we introduce advanced representation learning approaches based on BERT-like models. These approaches process disassembled Smali code from Dalvik bytecode to produce embeddings that capture more nuanced and complex malicious behaviors. The corresponding contributions are presented in the second part of the thesis:

- *DexBERT: Effective, Task-Agnostic and Fine-Grained Representation Learning of Android Bytecode:* We introduce DexBERT, a pre-trained model inspired by the BERT architecture, specifically designed to process disassembled Smali code. By generating embeddings that encapsulate fine-grained malicious behaviors, DexBERT enables more precise malware analysis, including localized detection of malicious components at the class level. Additionally, DexBERT proves to be versatile, excelling in other class-level Android learning tasks such as defect detection and component type classification, demonstrating its broad applicability as a representation learning model for Android bytecode.
  This work resulted in a research paper published at the IEEE Transactions on Software Engineering (IEEE TSE) in 2023
- *LaFiCMIL: Rethinking Large File Classification from the Perspective of Correlated Multiple Instance Learning*: While DexBERT has advanced class-level Android analysis tasks, its input size limitations pose challenges for app-level tasks like Android malware detection, requiring additional aggregation of embeddings. To overcome this limitation, we propose LaFiCMIL, a novel approach that addresses the classification of large files from the perspective of correlated multiple instance learning (c-MIL). With the goal of developing an approach that addresses the input constraints of general BERT-like models, not just DexBERT, we implement LaFiCMIL using a standard BERT architecture and validate its effectiveness on typical large file classification tasks such as long document classification and code defect detection. Experimental results show that LaFiCMIL significantly outperforms state-of-the-art baselines across multiple benchmark datasets in both efficiency and accuracy.
  This work resulted in a research paper published at the International Conference on Natural Language and Information Systems (NLBD) in 2024
- *DetectBERT: Towards Full App-Level Representation Learning to Detect Android Malware*: Building on the LaFiCMIL framework, we developed DetectBERT to leverage DexBERT's representation learning capabilities for app-level

malware detection using c-MIL. By treating class-level features as instances within c-MIL bags, DetectBERT aggregates these features into a unified app-level representation. This approach allows DetectBERT to analyze collective class behavior and identify Android malware by capturing patterns that emerge from correlations across different classes. Evaluations on a large-scale benchmark dataset demonstrate that DetectBERT sets a new state-of-the-art in Android malware detection.
This work resulted in a research paper published at the International Symposium on Empirical Software Engineering and Measurement (ESEM) in 2024

In the third part, we investigate another critical problem, i.e., Android malware family classification in a temporal-incremental scenario, addressing concept drifts related to the emergence of new malware families and shifts in the data distribution of old families. Our contributions as as follows:

- *Temporal-Incremental Learning for Android Malware Detection*: We formulate this problem as Temporal-Incremental Malware Learning (TIML), and provide a meticulously restructured million-scale dataset of Android malware classification tailored for the TIML paradigm, serving as a valuable resource for future research endeavors. We adapt state-of-the-art Class-Incremental Learning (CIL) approaches to meet the specific requirements of TIML. We also propose a novel multimodal TIML approach harnessing rich information from multiple malware modalities. Extensive evaluations demonstrate its significant superiority over original adapted CIL approaches. These findings reveal the feasibility of periodically updating malware classifiers at a minimal cost to address concept drift, establishing a new benchmark for robust and adaptive Android malware detection.
  This work resulted in a research paper published at the ACM Transactions on Software Engineering and Methodology (ACM TOSEM) in 2024

1.5 Roadmap

We illustrate in Figure 1.3 the roadmap of this dissertation. In Chapter 2, we review the related work relevant to this manuscript, providing a comprehensive overview of existing approaches in the field.

The contributions of this thesis are organized into three parts:

- Part I: We conduct an assessment of different Android artifacts in malware detection, with the study presented in Chapter 3. This evaluation explores the potential of various Android components to enhance malware detection.
- Part II: We delve into advanced representation learning approaches. In Chapter 4, we introduce our novel bytecode representation learning model, DexBERT. Following this, we present LaFiCMIL in Chapter 5, a method designed to address the input size limitations of BERT-like models, and finally, DetectBERT in Chapter 6, built on the LaFiCMIL framework, which leverages DexBERT's representation learning for app-level malware detection.
- Part III: We explore the problem of Android malware family classification in a temporal-incremental learning scenario, with the details covered in Chapter 7.

Finally, in Chapter 8 and Chapter 9, we conclude this thesis by summarizing our findings and discussing potential future research directions.

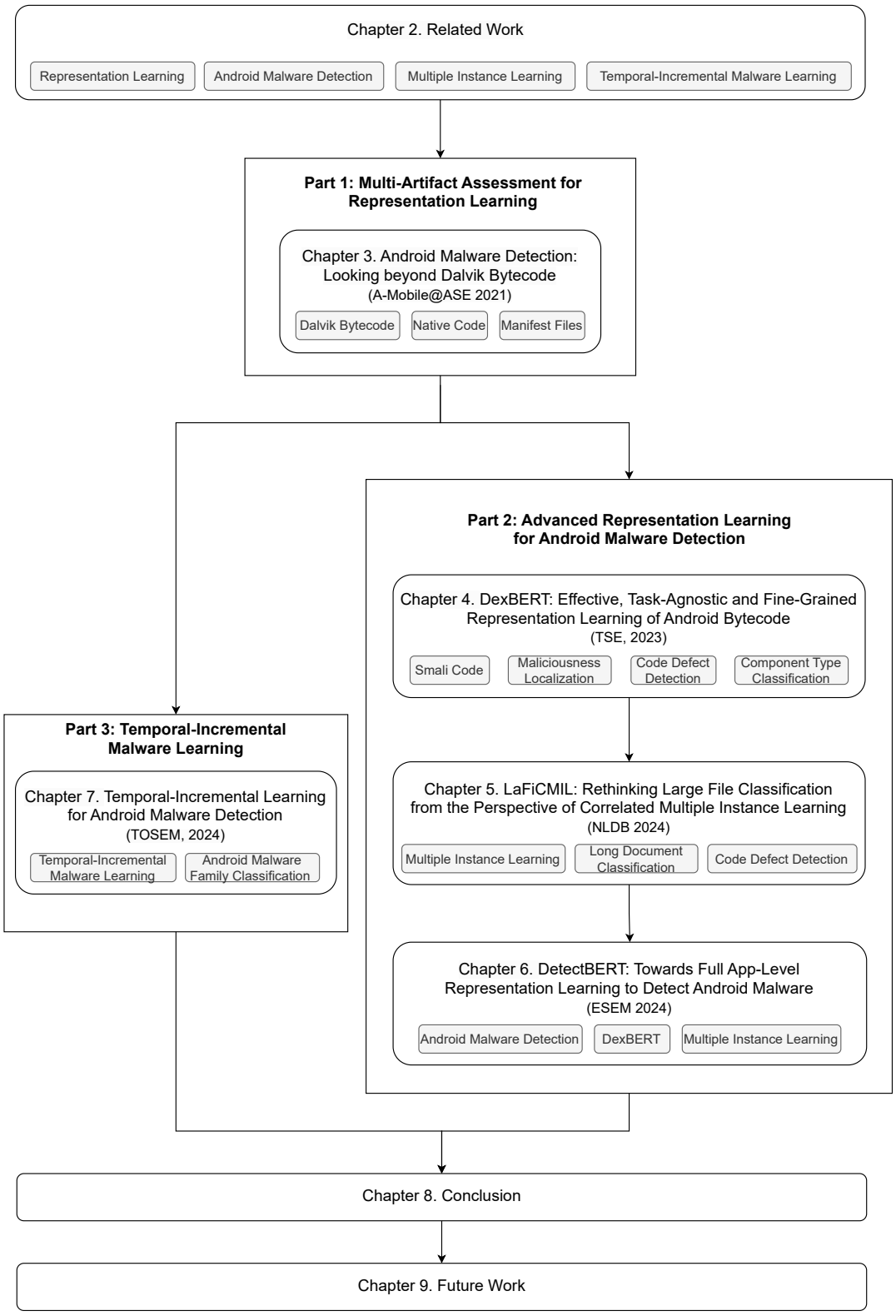

Figure 1.3: Roadmap of this dissertation.

2

Related Work

In this chapter, we review key research areas that are closely related to our work. We begin by examining the foundations of Android malware learning, with a focus on representation learning techniques, covering both general code representation and Android-specific app representation. Next, we explore existing studies on Android malware detection. In the third section, we investigate research on multiple instance learning. Finally, we review studies addressing temporal-incremental malware learning.

Contents

2.1 Representation Learning

The successes in deep learning have attracted increased interest in applying deep learning techniques to learn representations of programming artifacts for a variety of software engineering tasks [33, 34, 35, 36]. Representation learning aims to capture the semantic and syntactic information inherent in source code and other software artifacts, providing meaningful feature vectors that can be used for downstream tasks such as malware detection, bug prediction, and clone detection. By leveraging advanced machine learning models, researchers are increasingly able to automatically extract useful representations from code, leading to better performance in a variety of tasks within software engineering.

2.1.1 Code Representation

Code representation approaches aim to represent source code as feature vectors that contain the semantics and syntactic information of the source code. In general, code representations can be mainly categorized into sequence-based, tree-based, and graph-based representations. The tasks that rely on sequence-based representations consider source code as plain text and use traditional token-based methods to capture lexical information, such as clone detection [37], vulnerability detection [38], and code review [39]. Tree-based representations capture features of source code by traversing the AST of the source code. Code2Vec [35] proposes a path-attention model to aggregate the set of AST paths into a vector. TBCNN [40] learns code representations that capture structural information in the AST. Tree-LSTM [41] employs LSTM in learning the network topology of the input tree structure of AST. Graph-based representation approaches [42, 43] represent code as graphs that are associated with programs, such as control flow graph (CFG), control dependency graph (CDG), and data dependency graph (DDG).

Inspired by the success of transformer-based language models like BERT [44] and RoBERTa [45] in Natural Language Processing, Feng et al. [36] proposed CodeBERT, which is pre-trained both on programming languages and natural languages. Guo et al. [46] proposed GraphCodeBERT to advance CodeBERT by additionally considering data flow information in pre-training. Recently, the emergence and rapid evolution of Large Language Models (LLMs) have significantly revolutionized numerous research areas ,including code representation. The landscape of LLMs spans a broad spectrum, with models such as ChatGPT [47], GPT4 [48], LLaMA [49, 50], and Claude 3 [51] serving general-purpose applications, while others, such as StarCoder [52, 53], Code LLaMA [54], DeepSeek-Coder [55], and Code Gemma [56] are tailored specifically for code-centric tasks.

Although this section has highlighted some of the key studies on code representation, the literature contains a wealth of additional research that we were unable to cover comprehensively. For those interested in a more in-depth exploration, we recommend referring to existing review and survey papers [57, 58, 59, 60, 61].

2.1.2 Android App Representation

Android app representations aim to represent an Android app into feature vectors for various tasks such as malware detection [22] and clone detection [62]. Many works [63, 64, 65, 66, 67, 68, 69] relied on reverse engineering to extract information (features) from APKs and feed the extracted features into traditional ML-based and DL-based approaches to obtain Android representations [70]. Static features

such as permissions, API calls, and control flow graphs are widely used in prior works [71, 72, 73, 74, 63, 64, 75] to generate Android representations. There are several approaches where representation is based on dynamic features. For instance, several Android malware detectors [65, 66, 67, 68, 69] leveraged system calls traces.

The aforementioned features can be represented in different forms: the vectorized representation and the graph-based representation. Features such as permissions or API calls [71, 72, 73, 74, 76, 77, 78, 79], raw [30, 80] or processed [81] opcode sequences, and dynamic behaviors [82, 83] are mainly represented as vectors. Other graph-based features such as control flow graphs [84, 85] and data flow graphs [86] can be directly fed to DL models (e.g., Graph Convolutional Network [87, 88]) or embedded into vectors by graph embedding techniques (e.g., Graph2vec [89]). For readers seeking a more comprehensive exploration of Android app representation, we recommend consulting existing review and survey papers [90, 91, 92].

2.2 Android Malware Detection

Traditional Android malware detection methods have struggled to keep pace with the increasing complexity and diversity of malware [70, 93]. Signature-based approaches, for example, require extensive pre-processing and feature extraction [94, 95, 96] and are effective primarily against known malware types [97]. The integration of machine learning (ML) [22, 98] and deep learning (DL) techniques [99, 100, 19] represents a substantial advancement, offering more adaptable and robust detection capabilities to address evolving threats.

Early ML-based work like DREBIN[22] applied static analysis by training a Linear SVM on features from the DEX and Manifest files. Similarly, RevealDroid[101] and DroidMat[102] use SVM and KNN classifiers, respectively, with features such as API usage, permissions, and intents. MaMaDroid[103] innovatively abstracts API calls into Markov Chains fed into a Random Forest model, while ANASTASIA[104] selects key features with Extra Trees and trains an XGBoost model for enhanced detection. Recently, deep learning has been widely adopted for Android malware detection [105]. DroidDetector[106] utilizes hybrid analysis by feeding extracted permissions, sensitive APIs, and dynamic behaviors into a Deep Belief Network with unsupervised pre-training and supervised fine-tuning. DL-Droid[25] leverages application attributes, events, and permissions, using the Dynalog framework [107] to extract dynamic features and exploring both stateful [108] and stateless[1] input generation methods. Despite these advances, learning-based approaches still face substantial challenges. A primary limitation is the heavy reliance on static analysis [104, 109, 110], low-level bytecode [19, 111], and function call graphs [112, 113]. These approaches often fail to capture the complex and evolving behaviors of modern malware, particularly when dealing with obfuscated or dynamically changing malware patterns.

A comprehensive understanding of malware classification is crucial for advancing Android security, as it involves not only detecting malware but also accurately categorizing it into specific families or types. Over the years, several strategies have emerged to classify Android malware into distinct families. DroidSieve [114] integrates static features such as Certificates and Permissions, utilizing the Extra Tree algorithm [115]. Andro-Simnet [116] delves into hybrid features like API call sequences coupled with Permissions and further enhances it through Social Network

[1] `https://developer.android.com/studio/test/monkey`

analysis. DroidLegacy [117], on the other hand, bases its classification on API call-based signatures. Recently, there has been a shift towards leveraging image representations of bytecode for malware family classification [118, 119]. As an illustrative example, sections of the DEX file have been transformed into images, enabling the extraction of nuanced texture and color features [118]. These features, combined with plain-text traits, are subsequently processed by a multiple kernel learning classifier. In the same direction, MalNet [28] database has been released, which is a collection of over 1.2 million malware images spread across 696 families, which indeed facilitates future research in the realm of malware classification.

While this section has highlighted key studies on Android malware detection, a vast body of additional research exists that we could not cover in full. For readers interested in a deeper exploration, we recommend referring to existing review and survey articles [70, 120, 121, 122, 90].

2.3 Multiple Instance Learning

Multiple Instance Learning (MIL) has gained significant attention in recent years due to its broad applications in various domains [123, 124, 125], including machine learning for Android malware detection. In the context of Android malware learning, MIL presents an opportunity to aggregate features from different parts of an app, which can then be used to make holistic app-level predictions. While MIL is commonly applied in fields such as medical imaging and diagnosis [126, 127], the approach also holds potential in cybersecurity, particularly for handling large and complex datasets such as Android APKs. MIL approaches are broadly divided into two categories. The first group makes bag predictions based on individual instance predictions, typically using average or maximum pooling methods [128, 129, 130, 131]. The second group aggregates instance features to form a high-level bag representation, which is then used for bag-level predictions [132, 133, 134]. Although instance-level pooling is straightforward, aggregating features for bag representation has proven more effective [135, 133].

A fundamental assumption behind Multiple Instance Learning (MIL) is that instances within a bag are independent, but this might not always be true in real world. To tackle this, some research turns to Correlated Multiple Instance Learning (c-MIL), which assumes instances in a bag are correlated [136, 137, 133]. This correlated approach is especially relevant to Android representation learning, where leveraging interactions between components within an app could lead to more accurate malware detection. Despite its potential, the application of c-MIL for Android malware learning and large file classification remains underexplored, opening new avenues for future research.

2.4 Temporal-Incremental Malware Learning

Understanding the principles of incremental learning is fundamental to our study of Temporal-Incremental Malware Learning, particularly as it applies to the rapidly evolving field of Android malware detection. The nature of malware threats, characterized by constant evolution and new variant emergence, necessitates an approach that can adapt without the need for full retraining. This mirrors the incremental learning paradigm, which aims to update models dynamically as new data becomes available, thereby maintaining their relevance and effectiveness over

time.

2.4.1 Incremental Learning

In traditional deep learning, pre-collected datasets are predominantly used. With the emergence of streaming data, there are inherent challenges such as storage constraints [138, 139], privacy issues [140, 141], and elevated re-training costs [142, 143]. These challenges underscore the necessity for the development of Incremental Learning (IL) strategies, which accommodate dynamic and evolving data sources while minimizing the storage and computational overhead. A central challenge in IL is "catastrophic forgetting", where training on only new data might overwrite previous knowledge.

Incremental learning can be broadly categorized into three types, as discussed in various literature sources [141, 144]: Class-Incremental Learning (CIL), Domain-Incremental Learning (DIL), and Task-Incremental Learning (TIL). Among these, CIL has been the most extensively explored type [145, 146, 147, 148, 149, 150]. As a revolutionary CIL method,"Learning without Forgettin" (LwF) [151] addresses catastrophic forgetting through knowledge distillation, allowing for the integration of new classes into models while preserving previously acquired knowledge. This avoids the dependence on the original dataset and mitigates the cost of retraining. Following LwF, iCaRL [145] pioneered the trend towards exemplar-based methods, which soon gained widespread popularity. It introduced an exemplar-based methodology, employing nearest-mean-of-exemplars for classification and representation learning. Following on this direction, PODNet [146] introduced a distillation-based strategy that relies on spatial-based loss to curb representation forgetting. Meanwhile, SS-IL [147] identified that over-reliance on pure softmax can skew predictions in CIL. To address this, they introduced the Separated Softmax layer, which yielded significant improvements in performance. Recently, AFC [148] presented an innovative regularization technique for knowledge distillation. Their method prioritized the preservation of crucial features, ensuring minimal changes while assimilating new classes.

In the context of incremental learning, a "task" refers to a distinct learning objective or problem, often characterized by a unique set of classes or data distributions. TIL shares similarities with CIL in terms of managing newly arriving classes in tasks. The key difference manifests during the inference phase: While CIL requires classification across all classes, TIL restricts classification to the classes within the pertinent task, eliminating the need for cross-task discrimination. This makes TIL a more streamlined version, often seen as a subset of CIL [141, 144]. On the other hand, DIL is tailored for situations characterized by concept drift or distribution changes [142, 152, 153]. In these cases, new tasks come from varied domains but retain a consistent label space. Some DIL strategies [154, 155, 156] take inspiration from CIL methodologies. Since our DexBERT approaches primarily derive from CIL methods, we opt not to delve further into the details of TIL and DIL in this section.

2.4.2 Incremental Learning in Malware Classification

The application of incremental learning to malware classification remains underexplored, with no comprehensive studies addressing this topic. A critical limitation in the existing related studies is their inability to emulate real-world malware evolution, given their simplified and unrealistic assumptions. For instance, some studies relied

on a standard Class-Incremetal Learning (CIL) assumption. Li et al. [157] devised a CIL methodology based on multi-class SVM [158], and tested its efficacy on a small dataset. Compared to TIML, this approach does not utilize deep learning models, which are crucial for effectively implementing knowledge distillation—a key component of incremental learning. Moreover, their evaluation was limited to small datasets, insufficient to fully explore the complexities and evolving nature of malware threats. Qiang et al. [159] introduced a few-shot learning approach under the CIL assumption, facilitating the dynamic adaptation of pre-trained models to previously unknown families with minimal samples. While their method addresses the challenge of learning from limited data, it does not account for the distribution shifts in malware feature vectors within the same malware families, which is a critical aspect of what TIML effectively manages. Renjith et al. [160] investigated incremental learning for on-device Android malware detection. However, their study was limited to binary classification (i.e., predicting whether a given app is malware) and their dataset was not organized in temporal order, failing to capture the real-world evolution of malware.

Rahman et al. [161], in their attempt to explore three different IL scenarios, faced bottlenecks in performance due to various limitations. For instance, their emulation of CIL involved a straightforward addition of new malware classes in successive steps, without taking into account the authentic lifecycle of malware in the real world. Furthermore, their reliance on basic ML features to train DL models on limited datasets (in the order of thousands) inevitably hindered the capabilities of the DL models and the potential of IL techniques. Recently, Chen et al. [162] delved into continuous learning applied to malware detection, addressing the concept drift issue prevalent in Android malware classifiers. They focus on reducing human efforts in the process of labeling emerging malware by selecting representative samples with their techniques. However, they still need to retrain the classifier on the full historical dataset. In contrast, TIML aims to retrain the classifier using only new malware without historical dataset to reduce the training and storage resources.

Part I

Multi-Artifact Assessment for Android Representation Learning

Machine learning, particularly deep learning, has been extensively used in the literature on malware detection, as it scales effectively to analyze large samples of Android applications. Feature engineering has played a crucial role in advancing research in this area. However, most existing works have predominantly focused on Dalvik bytecode and Manifest files as the primary artifacts for extracting features. In this part, we extend the exploration to other under-utilized artifacts beyond these two, conducting a comparative evaluation of their discriminative power in the context of malware detection.

3

Android Malware Detection: Looking beyond Dalvik Bytecode

Recently, a new research direction that builds on the momentum of Deep Learning for computer vision has produced promising results with image representations of Android bytecode. In this chapter, we postulate that other artifacts, such as binary (native) code and metadata/configuration files, could be looked at to build more exhaustive representations of Android apps. We show that binary code and metadata files can also provide relevant information for Android malware detection, i.e., they can detect malware that models built solely on bytecode cannot detect. Furthermore, we investigate the potential benefits of combining all these artifacts into a unique representation with a strong signal for reasoning about maliciousness.

This chapter is based on the work published in the following research paper [111]:

- Sun, T., Daoudi, N., Allix, K. and Bissyandé, T.F., 2021, November. Android malware detection: looking beyond dalvik bytecode. In 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW) (pp. 34-39). IEEE.

Contents

3.1 Overview

Android applications have pervaded virtually all aspects of the lives of hundreds of millions of consumers. Equipping over a billion devices, Android apps are at the center of various research initiatives, notably with respect to security. For example, malware detection research is a never-ending race where approaches quickly become obsolete as malware writers evolve their techniques to hide malicious payload inside different artifacts.

In the early days of Android, malicious behavior patterns were easily detected with simple static [95] and dynamic analysis [163] or based on similarity comparisons [21]. As malware evolved, researchers considered applying machine learning with a large focus on engineering various sets of features based on static or dynamic analysis outputs. While such statistical machine learning approaches [22, 164, 103, 104, 165, 166, 167] achieved good performance on literature benchmarks, the trend today is to investigate neural network architectures towards learning comprehensive representations of Android apps. Such representations exploit lexical information of bytecode [168] and abstract syntax tree representations of code [169], etc. More recently, inspired by the remarkable successes of Deep Learning in the field of computer vision, researchers have started to investigate Android malware detection methods based on image representations of apps [99, 170]. To advance research in this direction, Daoudi et al. [19] developed DexRay as a baseline approach for Android image-based malware detection. Our work is set in the same ambitious research agenda that DexRay laid down: how can we best exploit image representations of Android apps to detect malware?

A very significant reason for the success of deep learning in computer vision is that natural images contain rich semantic information which could provide meaningful features for recognition tasks. If the powerful feature learning capabilities of deep learning are to be used for the task of malware detection, representing apps as images is indeed an appealing prospect. Unfortunately, representing an APK (Android PacKage) as an image is not a straightforward endeavor: In particular, what should be captured in the image has not yet been defined nor studied. A model trained on images presenting a partial view of the objects under study will suffer from blind-spots in its recognition tasks.

Translated into the Android realm, an APK file is itself made of a collection of files, that contain artifacts of different nature. Among the various artifacts in an APK, DexRay and many prior approaches only consider the Dalvik bytecode (.dex files, that we will refer to as `.dex` throughout the chapter). Several other approaches have instead extracted features from the Manifest XML file (that we will refer to as `.xml` throughout the chapter), which contains metadata about an Android App. The native code (.so files *i.e.*, Shared Library files, that will be referring to as `.so`) is seldom considered as a signal source for malware detection. Yet, native code being challenging to analyze, it is a sweet spot to hide malicious behaviors.

In this chapter, we seek to contribute to the systematization of knowledge around Android malware detection, by investigating the value of multiple types of artifacts in representing apps for malware detection. We make the following contributions:

- We evaluate the suitability of three major types of artifacts for Android malware detection;
- We assess whether these artifacts each bring added-value, or whether they are

redundant;
- We investigate four possible methods of combining these artifacts into one Deep Learning approach.
- To facilitate replication, we have made the dataset and source code available at: `https://github.com/Trustworthy-Software/Looking-beyond-Dalvik-Bytecode`

More specifically, we investigate the following research questions:
- **RQ1:** Does each of the major artifacts in Android apps contain relevant information for Malware Detection?
- **RQ2:** How redundant is the information across three considered artifacts?
- **RQ3:** To what extent can the performance of Malware Detection be improved by combining these artifacts?

3.2 Experimental Setup

In this section, we first present a brief description of DexRay. Then, we present the dataset and the experimental setup used to conduct our experiments.

3.2.1 Background on DexRay

DexRay [19] is a recent work that presented and evaluated a simple image-based App representation for Android malware detection. It converts the Dalvik bytecode of Android apps into gray-scale vector image representations. Each byte in the `.dex` file is mapped to a pixel value in the generated image, and all images for apps are resized to one single size, independently of the size of the app or of its bytecode. The features extraction and the classification are both carried out automatically using a simple 1-dimensional convolutional neural network architecture. This approach has been proven to be highly effective in detecting Android malware by reporting an F1-score of 0.96.

3.2.2 Dataset

Table 3.1: Summary of DexRay Dataset

	Number of APKs	Number of APKs with `.so` files
Benign apps	96 858	63 037
Malware apps	61 696	56 678
Total	158 554	119 715

We reuse here the dataset that was used in DexRay[19][1], which was collected from AndroZoo [29]. The APKs (*i.e.*, Android PacKages) in the DexRay dataset are from the period of 2019 to 2020. Benign apps are defined as the apps that have not been detected by any antivirus from VirusTotal[2]. The malware collection contains the apps that have been detected by at least two antivirus engines. We present in Table 3.1 a summary of this dataset. Since not all the APKs contain native code, we also summarize in the same Table the number of apps that include `.so` files.

[1] `https://github.com/Trustworthy-Software/DexRay`
[2] `https://www.virustotal.com/`

3.2.3 Experimental Methodology

In our experiments, we adopt the same experimental setup of DexRay. Specifically, we use the Hold-out strategy [171] by dividing the dataset into 80% for training, 10% for validation and 10% for testing. The detection model is trained on the training dataset, and the model hyper-parameters are optimized based on its performance on the validation dataset. The above process is repeated 10 times by randomly shuffling and splitting the dataset.

We train each model for a maximum of 200 epochs, and we stop the training using an early stopping strategy (we set the patience step to 50). Regarding the convolutional layer, we set the kernel size to 12 and we use `relu` [172] as the activation function. The two convolutional layers contain 64 and 128 channels respectively. The model architecture also contains two dense layers with a `sigmoid` activation function [173]: a first layer with 64 neurons, and a second layer with one neuron for the classification.

As we have presented in Section 3.2.2, some APKs do not contain native code (*i.e.*, `.so` files). Thus, we have set each pixel to zero in the corresponding gray-scale image (*i.e.*, blank image) when the `.so` files are missing in a given APK.

We evaluate the performance of our models using four standard metrics: Accuracy, Precision, Recall and F1 score.

3.3 Empirical Investigation

In this section, we conduct our evaluation by addressing the three research questions outlined in Section 3.1.

3.3.1 RQ1: Does each of the major artifacts in Android apps contain relevant information for Malware Detection?

Table 3.2: Detection performance for each type of artifacts on the DexRay dataset

Source	Accuracy	Precision	Recall	F1-score
`.dex`	0.972	0.974	0.953	0.963
`.so`	0.953	0.985	0.892	0.936
`.xml`	0.94	0.918	0.927	0.923

To investigate this RQ, we train a model for each of the three major types of artifacts: `.dex`, `.xml`, and `.so` files. The `.dex` model is thus trained only on dex files (as in the original DexRay paper). For the `.so` model, we build the images by considering the bytes of the `.so` file(s) of an APK instead of the `.dex` file(s)[3]. Similarly, in the `.xml` model, the bytes of the Manifest file are used as a source for the images.

As recommended in the DexRay paper [19], we choose the size of $(1, 128 \times 128)$, as it is the best trade-off between the model accuracy and computational efficiency. We conduct our experiments using the experimental methodology presented in Section 3.2.3 and we present our results in Table 3.2.

As shown in Table 3.2, the detection models have achieved F1-scores of 0.963, 0.936, and 0.923. From the overall results, we can conclude that `.so` files and `.xml`

[3]In case no `.so` file is present, a blank image is used.

files can indeed provide relevant information for detecting malware, and the detection performance is competitive. We note that the `.so` model obtains a Recall of 0.892, even though over 8% of the malware do not have any `.so` files.

RQ1 Answer: The major artifacts (*i.e.*, `.dex` files, `.so` files and `.xml` files) in Android apps contain relevant information for malware detection. Although the `.so` and the `.xml` models do not reach the same performance as the `.dex` model, they still perform relatively well.

3.3.2 RQ2: How redundant is the information across three considered artifacts?

Table 3.3: Overlap and Differences of predictions made by `.dex`, `.so`, and `.xml` models

		Detected by			Missed by		
		`.dex`	`.so`	`.xml`	`.dex`	`.so`	`.xml`
Detected by	`.dex`	na	14 917	14 655	na	476	738
	`.so`	14 917	na	14 415	186	na	688
	`.xml`	14 655	14 415	na	242	482	na

The models that are trained on `.so` and `.xml` files have shown promising scores for malware detection, but have a lower Recall than the model built on `.dex`. Hence it is possible that the malware detected by the `.so` and `.xml` models are just a subset of the malware detected by the `.dex` model. An explanation for such a case could be that the `.so` and `.xml` artifacts do not bring additional information compared to the `.dex` artifacts. The goal of RQ2 is to investigate whether `.so` and `.xml` could provide complementary information to `.dex` models. Therefore, we examine the overlap between the samples that are detected by each of the three models, as well as the samples that are detected by a model and missed by another. We present our results in Table 3.3.

As shown in Table 3.3, most of the samples can be correctly classified (*i.e.*, malware or benign apps) by the three models. At the same time, there are malware samples that the `.dex` model fails to detect but the `.so` model and/or the `.xml` model detect and vice versa. Specifically, in all 15 855 APKs in the test dataset, `.so` model can detect 186 APKs that escape the detection of `.dex` model. Also, the `.xml` model successfully detects 242 APKs that the `.dex` model fails to detect. This observation suggests that there is knowledge that could be harnessed from the three different sources of information to enhance the detection performance of the `.dex` model.

RQ2 Answer: Most of the apps are detected by the three models, indicating that most of the information contained in the different artifacts is redundant in the context of malware detection. Nevertheless, each model detects malware that the other models do not. Therefore, it is expected that a model built on all three sources of artifacts could outperform a single-source model.

3.3.3 RQ3: To what extent can the performance of Malware Detection be improved by combining these artifacts?

As we have concluded in Section 3.3.2, the three types of artifacts can complement each other. In this section, we investigate different methods to combine the information provided by the `.dex`, the `.so`, and the `.xml` files.

3.3.3.1 Aggregation Methods

In the following, we present a brief description of four malware detection approaches that consider information from the three types of artifacts.

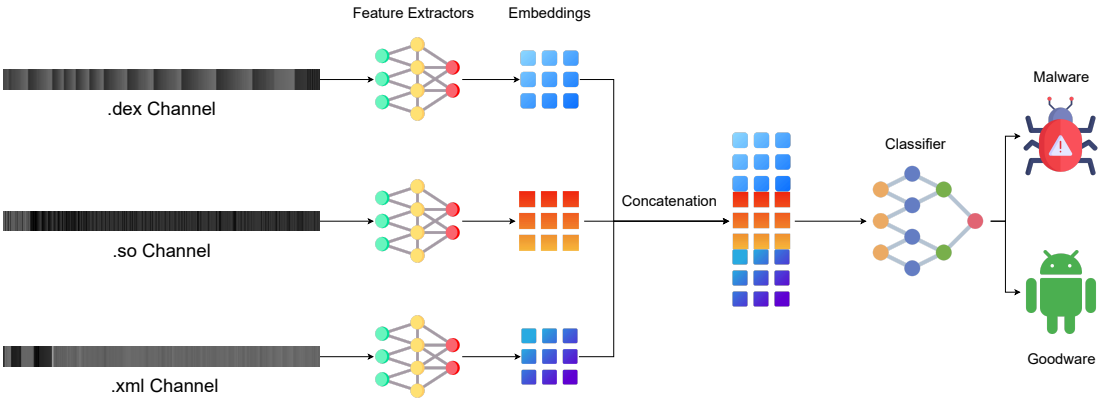

Figure 3.1: Architecture of Ensemble Model for Color-scale Images

- **Long-vector images** For this model, we generate one image that contains the concatenation of the gray-scale 1-d images horizontally, i.e., the resulting image has a $(1, 128 \times 128 \times 3)$ dimension. Since the Long-vector images have the same form of the gray-scale 1-d images (They only differ in the width), we rely on the same model architecture of DexRay.
- **Rectangular images** For the *rectangle images*, the three gray-scale 1-d images are stacked vertically, i.e., the resulting image has a $(3, 128 \times 128)$ dimension. The same model architecture of DexRay is used with the rectangular images.
- **Color images** As a potential method to combine the three sources of artifacts, we investigate a "color" image, where each color ("channel") of the resulting image is built from the gray-scale image for one artifact source. We use the same model architecture of DexRay, and we apply minor modifications in order to make it compatible with the 3-channel image. Specifically, we use MaxPooling2D with $pool\_size = (12, 1)$, instead of MaxPooling1D with $pool\_size = (12)$. As for the convolutional layers, we have not made any change since the Keras API[4] automatically adapts to inputs with different dimensions. In our case, the 1-d convolutional layer thus has 3 kernels for the 3-channel input image.
- **Ensemble Model** In the ensemble version (as shown in Figure 3.1), we separately learn the features from each gray-scale 1-d image. Similar to the model for gray-scale image, each type of image undergoes its corresponding model branch which consists of two 1-d convolutional layers and two 1-d pooling layers. Next, we use a Concatenation layer to combine the feature

[4] `https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv1D`

maps extracted from the three gray-scale images. Then, we add another 1-d convolutional layer to learn and combine features from the three images. For the classification, we use the same two dense layers of DexRay.

3.3.3.2 Evaluation Results

Table 3.4: Detection Results of combined sources compared to single `.dex` source on DexRay Dataset

Source	Accuracy	Precision	Recall	F1-score
`.dex`	0.972	0.974	0.953	0.963
long vector	0.969	0.97	0.949	0.959
rectangle	0.972	0.972	0.956	0.964
color	0.972	0.973	0.953	0.963
ensemble	0.974	0.973	0.959	0.966

We evaluate the detection performance of each of the previous models and we report our results in Table 3.4. Overall, we notice that the long-vector and the color image-based models do not improve the detection performance compared to the `.dex` image-based model. As for the rectangular and the ensemble models, they bring a slight improvement. In our experiments, the same dataset splits are used for the evaluation of all the model, which means that the ensemble model (i.e., the best combination method) outperforms the `.dex` model on the same exact applications. Although the ensemble model fails to detect 165 apps that are correctly predicted by the `.dex` model, it manages to detect 201 samples that have escaped the detection of the `.dex` model.

RQ3 Answer: Some combination methods can bring some improvements—although very minor—to the detection performance. None of the four proposed methods seem to fully benefit from the additional information brought by the different types of artifacts.

3.4 Discussion

In this section, we first describe some insights about the image representations of Android apps for malware detection in Section 3.4.1. Then, we discuss some potential threats to validity of the proposed methods in Section 3.4.2.

3.4.1 Some Insights

In this chapter, we find that in addition to Dalvik bytecode, binary native code and Manifest files can also provide rich and relevant information for Android malware detection. Both the Dalvik bytecode and the Manifest files have been heavily used in the literature. However, to the best of our knowledge, there has been no comparative evaluation of their relative discriminating power in the context of malware detection. Several ready-to-use tools have been developed to analyze `.dex` files. In contrast, analyzing Android apps' native code is not as streamlined a task. This could potentially explain why so few prior works leverage native code. One contribution of the this chapter is to demonstrate that native code can indeed be leveraged without complex analysis, using a simple and straightforward representation.

An important point about Android Malware detection is that the performance exhibited by state-of-the-art approaches—and even by a single source `.dex` model—is already very high. Therefore, the potential gain in performance is very small, but nonetheless highly sought after: Even a mere 1% performance improvement could mean *thousands* of additional malware uncovered each month. The present work offers a perspective to a simple yet important question: In the field of Android malware detection, where can improvements come from?

3.4.2 Threats to Validity

Our experiments and conclusions face some threats to validity. First, our experiments were conducted on a single dataset, thus the generalizability of our conclusions needs further verification. Nevertheless, the dataset we use is large, recent, and likely representative of what can be found on Android apps markets. It is furthermore made available to researchers for further experiments.

Second, the results we obtain may be specific to the image representation that we used. Indeed, how to best represent an Android app (or part of it) as an image is still an open question. By reusing the representation introduced in DexRay, our results are directly comparable to those of the original DexRay. Furthermore, we note that all the models we investigate in this chapter achieve an f1-score above 0.9, suggesting that the representations used are adequate for Malware detection.

3.5 Summary

In this chapter, we demonstrate that the image representations of the three major types of Android artifacts considered—Dalvik bytecode, XML (Manifest files), and native code (shared objects)—each contain relevant information for malware detection. We show that each artifact contributes unique and valuable insights that are not present in the others. For instance, models based on `.xml` and `.so` artifacts successfully detect Android apps that evade detection by the `.dex` (Dalvik bytecode) model. We also explore four approaches to combine these artifacts into a unified image representation that encompasses all three sources. Some of these approaches yield slight improvements in performance, demonstrating the potential of multi-artifact analysis.

However, despite the incremental gains achieved through artifact combination, our results indicate that Dalvik bytecode alone remains the most efficient and effective artifact for Android malware detection. This suggests that while combining artifacts may enhance detection performance marginally, Dalvik bytecode remains the preferred choice due to its strong standalone capabilities.

Part II

Advanced Representation Learning for Android Malware Detection

From Part I, we found that although combining multiple artifacts provides slight improvements in Android malware detection, Dalvik bytecode alone remains the most efficient and effective option, making it the preferred choice for Android malware detection. However, existing bytecode representation techniques, such as DexRay's grayscale "vector" images, have limitations in fully capturing the semantic information of malware behaviors from Dalvik bytecode. In this part, we explore more advanced representation learning methods specifically designed to overcome these limitations and enhance Android malware detection.

4

DexBERT: Effective, Task-Agnostic and Fine-Grained Representation Learning of Android Bytecode

Machine Learning (ML) is increasingly automating software engineering tasks by converting software artifacts, such as source or executable code, into representations suitable for learning. Traditional approaches rely on manually crafted features, which can be imprecise and incomplete. Representation learning offers a more effective solution by automatically extracting relevant features. In Android-related tasks, models like apk2vec and DexRay focus on whole-app representations, while others, like smali2vec, target specific tasks, limiting their general applicability and failing to capture the full semantic information of Dalvik bytecode. This chapter introduces DexBERT, a BERT-like model for task-agnostic and fine-grained representation learning of Dalvik bytecode. DexBERT captures information at a class-level granularity and is evaluated on three class-level tasks: Malicious Code Localization, Defect Prediction, and Component Type Classification. We also explore strategies to handle the varying sizes of apps, demonstrating how DexBERT can effectively capture meaningful information across different contexts. While this chapter focuses on the class-level representation capabilities of DexBERT, its potential extends to app-level tasks like Android malware detection, which will be explored in the subsequent chapters.

This chapter is based on the work published in the following research paper [174]:

- Sun, T., Allix, K., Kim, K., Zhou, X., Kim, D., Lo, D., Bissyandé, T.F. and Klein, J., 2023. Dexbert: Effective, task-agnostic and fine-grained representation learning of android bytecode. IEEE Transactions on Software Engineering.

Contents

4.1 Overview

Pre-trained models yielding general-purpose embeddings have been a recent highlight in AI advances, notably in the research and practice of Natural Language Processing (*e.g.*, with BERT [175]). Building on these ideas, the programming language and software engineering communities have attempted similar ideas of learning vector representations for code (*e.g.*, CODE2VEC [35]) and other programming artifacts (*e.g.*, bug reports [176]). Unfortunately, these pre-trained models for code embedding often do not generalize beyond the task they have been trained on [177].

In the Android research landscape, many techniques have been proposed to address app classification problems [66, 75, 68, 67, 65, 63, 178, 179], most of which, however, can only handle coarse-grained tasks (*i.e.*, at the whole-app level). Although there are a few works [26, 180] targeting some fine-grained tasks at the class level, their learned representations have also not been shown to generalize to other class-level tasks. Therefore, despite the good performance exhibited by existing approaches, there is still a research gap to be filled with the investigation of simultaneously fine-grained and task-agnostic representation learning for Android applications. Indeed, advances in this direction will help researchers and practitioners who are conducting class-level tasks, such as malicious code localization or app defect prediction, to achieve state-of-the-art performance while reducing costs due to manual feature engineering or repetitive pre-training computations for multiple representation models.

Building a fine-grained task-agnostic model is, however, challenging since it essentially requires to capture knowledge relevant to a variety of tasks altogether and at the low granularity of the representation. A few studies have investigated and built task-agnostic models in the field of software engineering. CodeBERT [36] and `apk2vec` [181] are key representatives of these models. On the one hand, while CodeBERT brings significant improvements, it cannot be directly used for representing Android apps for two main reasons: lack of source code in apps and limit of input elements. Even though it is technically feasible to apply it to the assembly language `Smali`, the performance is unsatisfactory, as evidenced by our experiments in Section 4.5.3. On the other hand, `apk2vec` successfully constructs the behavior profiles of apps and achieves significant accuracy improvements while maintaining comparable efficiency. However, despite `apk2vec`'s success in app representation, its granularity, and graph-based design still have limitations [182, 183, 184, 185, 186]. Notably, `apk2vec` is designed to handle only app-level tasks, while our proposed approach is targeted at fine-grained tasks at the class-level.

Towards addressing limitations of existing representation techniques (*i.e.*, lack of a universal model for Android bytecode at low granularity), we propose in this chapter DexBERT, a fine-grained and task-agnostic representation model for the bytecode in Android app packages. The result of DexBERT can be applied across various class-level downstream tasks (*e.g.*, malicious code localization, defect prediction, *etc.*). Our approach first extracts features from `Smali` instructions (*i.e.*, an assembly language for the *Dalvik* bytecode used by Android's Dalvik virtual machine.). It then combines embeddings of code fragments to build a model that can address various class-level problems. Our pre-trained model can capture the essential features and knowledge by first learning an accurate general model of Android apps' bytecode. The proposed aggregation methods allow DexBERT to handle fine-grained class-level tasks, making DexBERT capable of operating on lower granularity of Android

artifacts than other state-of-the-art representation models.

To evaluate the effectiveness of DexBERT, we first conduct a preliminary experiment to observe the feasibility of building a general model of Android app code. This experiment mainly focuses on pre-training BERT on `Smali` instructions to ensure that the generated embeddings contain meaningful features for various tasks. We observed clearly converging loss curves on all the pre-training tasks, with the pre-trained model achieving 95.30% and 99.35% accuracy on the masked language model and next sentence prediction tasks, respectively. Such performances demonstrate that our model indeed learned meaningful features and can be generalized to a variety of different tasks.

We further perform a comprehensive empirical evaluation to measure the overall performance of DexBERT on three class-level downstream tasks: Android malicious code localization, Android defect detection, and component type classification. Android malicious code localization is a task whose goal is to identify the malicious parts of Android apps. Android defect detection locates defective code to help developers improve the security and robustness of Android apps. component type classification is a multi-class classification problem that has a distinct character compared to the two tasks mentioned above. This classification problem is introduced to provide a more comprehensive evaluation of DexBERT's universality. Our experimental results show that DexBERT can localize malicious code and detect defects with significant improvement over current state-of-the-art approaches (74.93 and 6.33 percentage points improvement for malicious code localization and defect prediction, respectively) in terms of accuracy. DexBERT also significantly outperform other BERT-like baselines on the task of component type classification by a roughly 20 percentage point increase in terms of F1 Score.

The contributions of our study are as follows:

- We propose a novel BERT-based pre-trained representation learning model for Android bytecode representation, named DexBERT. It can be used directly on various class-level (*i.e.*, finer-grained than existing app-level approaches) downstream analysis tasks by freezing parameters of the pre-trained representation model when tuning the prediction model for a specific downstream task.
- We propose aggregation techniques that overcome the limitations with the size of the input of BERT.
- We conduct a comprehensive evaluation that shows that DexBERT achieves promising performance on multiple pre-training tasks and class-level Android downstream tasks.
- To ease replication, we share the dataset and source code to the community at the following address:
  `https://github.com/Trustworthy-Software/DexBERT`.

4.2 Approach

In this section, we first present the overview of DexBERT workflow in Section 4.2.1. Then, we illustrate details of DexBERT in Section 4.2.2, and we describe the applications of representations learned by DexBERT on downstream tasks in Section 4.2.3.

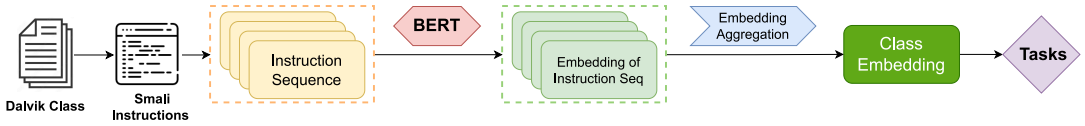

Figure 4.1: Overview of a class embedding by DexBERT.

4.2.1 Overview

DexBERT focuses on extracting features from Android application Dalvik bytecode and targets fine-grained Android tasks (*i.e.*, at class-level). Our approach is clearly different from `apk2vec` [181], which is a static analysis based multi-view graph embedding framework for app-level Android tasks, and CodeBERT, which is a bimodal pre-trained model for programming language (PL) and natural language (NL) tasks. Specifically, DexBERT takes disassembled `Smali` code from Dalvik bytecode as input and learns to extract corresponding representations (*e.g.*, the embedding of a class). `Smali` is a text representation of Dalvik bytecode, in the same way, that assembly code is a text representation of compiled code. Because DexBERT supports various class-level tasks (while the original BERT is limited to relatively small inputs), there is a need to combine, or aggregate the representations of several sequences of instructions, or *chunks*, into one single representation that covers the chosen input.

As shown in Figure 4.1, representation (or embedding) of the `Smali` Bytecode is learned by BERT during the pre-training phase. This learned embedding can then be applied to class-level Android downstream tasks. Specifically, we can easily extract the Bytecode of an Android application. After disassembling each Dalvik class, we obtain the `Smali` instructions. This flow in `Smali` instructions (grouped in chunks) is fed to the BERT model in order to pre-train it, *i.e.*, to learn how to represent `Smali` code. Then, to obtain the embedding of a `Smali` class, we aggregate the learned DexBERT representation of each `Smali` instruction sequence present in the class.

4.2.2 DexBERT

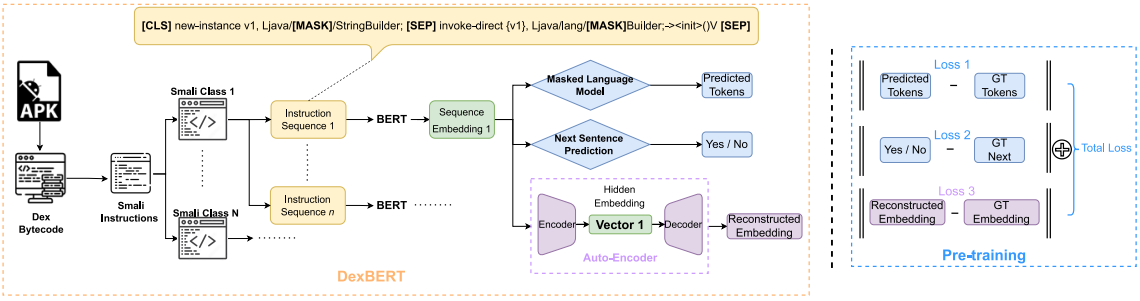

Figure 4.2: Illustrations of DexBERT and Pre-training Loss Function. "GT" is an abbreviation for "ground-truth".

We introduce the pipeline of DexBERT in Section 4.2.2.1. The mechanism of DexBERT pre-training is presented in Section 4.2.2.2. In Section 4.2.2.3, we introduce the Auto-Encoder, which is designed to reduce the dimensionality of the learned representation while keeping the key information.

4.2.2.1 DexBERT

As an assembly language, the `Smali` code disassembled from *Dalvik* bytecode is a sequence of instructions. Similar to the original BERT, we pre-train our model on multiple pre-training tasks to force DexBERT to capture general-purpose representations that can be used in various downstream tasks.

Specifically, as shown in the left box in Figure 4.2, regarding each `Smali` instruction in each class as a text snippet we are able to create instruction pairs (similar to sentence pairs in original BERT) as input sequences. In each pair, two instructions are separated by a reserved token `[SEP]`, and several randomly selected tokens (or "word") are masked. For the masked language model, one of the two pre-training tasks of the original BERT, the goal is to correctly predict the tokens that are masked. The second original BERT pre-training task, Next Sentence Prediction, is turned here into Next Instruction Prediction, leveraging the pairs of instructions. In essence, this task's goal is to predict, given a pair of instructions, whether or not the second instruction follows the first one.

4.2.2.2 Pre-training

Pre-training plays a vital role in helping DexBERT learn to generate meaningful embeddings. To ensure that the learned embeddings have general-purpose, the pre-training is supposed to be performed on multiple tasks simultaneously. As described, we adopt and adapt the two pre-training tasks of the original BERT: ① masked language model (*i.e.*, masked words prediction) and ② next sentence prediction as the main pre-training tasks.

In each pre-training iteration, the input sequence of instructions is fed into the BERT model, which generates a corresponding sequence embedding (as shown in the left box in Figure 4.2). The sequence embedding is then taken as input for the pre-training tasks. Each task is a simple neural network with a single fully connected layer. As shown in the right box in Figure 4.2, a loss value is then calculated by comparing the output of each task head to the automatically created ground-truth (*i.e.*, randomly masked tokens or binary label indicating whether the second statement indeed follows the first one or not). The model weights of connections between neurons are adjusted to minimize the total loss value (*i.e.*, the sum of all loss values for pre-training tasks) based on the back-propagation algorithm [187]. Note that the pre-training tasks are only designed to help the BERT model learn meaningful features of input sequences and have little practical use in the real world.

4.2.2.3 Auto-Encoder

Even though the two aforementioned pre-training tasks could work well on `Smali` instructions, the dimensionality (*i.e.*, 512×768, which is defined as the multiplication of token number in each input sequence N and the dimension of the learned embedding vector H) of the generated representation for each sequence is quite large. Since there are usually hundreds or thousands of statements in a `Smali` class that need to be embedded, the dimensionality of the learned embeddings should be reduced before deployment to downstream tasks while preserving their key information. In common practice for BERT-like models, the first state vector (of size 768) of the learned embedding is often used for this purpose. In our case, we add a third pre-training task, an Auto-Encoder, whose goal is to find a smaller, more efficient representation.

A smaller representation is necessary primarily because APKs consist of a significantly larger number of tokens compared to typical textual documents and code files. We provide a comparative analysis on token sequence length across three different data formats - textual documents (Paired CMU Book Summary [188]), code files (Devign [189]), and APKs (DexRay [19]), the number of BERT tokens in each dataset, denoted as [Mean]±[Deviation]. Specifically, the Paired CMU Book Summary has 1148.62±933.97 tokens, the Devign dataset contains 615.46±41 917.54 tokens, while the DexRay dataset contains a considerable 929.39K±11.50M tokens. This substantial quantity difference in tokens for APKs necessitates the effort to achieve as compact an embedding as possible for a given token sequence of Smali instructions.

AutoEncoder [190] is an artificial neural network that can learn efficient small-sized embedding of unlabeled data. It is typically used for dimensionality reduction by training the network to ignore the "noise". The basic architecture of an Auto-Encoder usually consists of an Encoder for embedding learning and a Decoder for input regenerating (as shown in the left box in Figure 4.2). During the training process, the embedding is validated and refined by attempting to regenerate the input from the embedding, essentially trying to build the smallest complete representation of its input.

The encoder in our approach consists of two fully connected layers with 512 and 128 neurons, respectively. Symmetrically, the decoder consists of two fully connected layers with 128 and 512 neurons, respectively. The sequence embedding (with the size of 512×768) learned from BERT is both the input of the Encoder and the output target of the Decoder in the pre-training process. After comparative experiments, we opted for a hidden embedding with size 128 as the final representation of the original instruction sequence. We provide an analysis of different embedding sizes in Section 4.5.1. Compared with the raw BERT embedding, the dimension of the final sequence embedding is 3072 times smaller, from 512×768 to 128. Consequently, in Figure 4.2, the size of *Vector 1*, which is the embedding of *Instruction Sequence 1* yielded by the Encoder, is 128.

4.2.3 Class-level Prediction Model

In order to validate the effectiveness of learned DexBERT representation, we apply it to three class-level Android analysis tasks. To perform these class-level tasks, we need an efficient representation for each class, and we thus need a method to aggregate the embeddings of the instruction sequences of each class into one single embedding. We introduce this method in Section 4.2.3.1. We then present the details of the prediction model in Section 4.2.3.2.

4.2.3.1 Aggregation of Instruction Embedding

The representations learned from DexBERT are for instruction sequences. Usually, each `Smali` class consists of many methods, each of which contains a certain, often large, number of statements. The number of sequence vectors in each class is indeterminate, while the shape of the input to the class-level prediction model (neural network) is supposed to be fixed. Specifically, within each class, there are many sequence embedding vectors with a size of 128, which are expected to be combined into one single vector. Thus, embedding aggregation is required to obtain the final class-level representation.

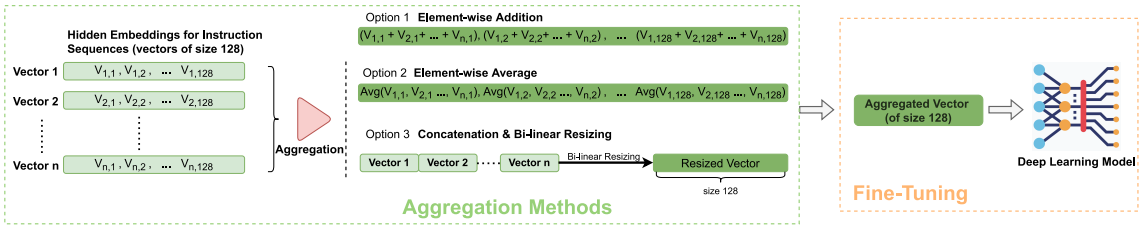

Figure 4.3: Illustrations of Three Embedding Aggregation Methods and Fine-tuning of Downstream Tasks.

We primarily have three reasons to solve the long instruction sequence problem by splitting a `Smali` class into snippets and aggregate the learned snippet embedding into one class embedding. First, `Smali` instructions are individual commands performing specific operations in the Android run-time environment. Although they often appear in sequences, each `Smali` instruction operates largely independently, not necessarily bearing the kind of interdependence seen in words within natural language sentences. Second, the class representation, which aggregates the instruction embeddings, retains a sense of the overall structure and function of the class while being context-aware. Last, while there might be some loss of context when splitting long sequences, we believe the trade-off between computational efficiency and a minor potential loss of context is justified. To give a quantified measure of this trade-off, let's consider the GPU memory required. Doubling the input length limit for a BERT model would necessitate four times the initial GPU memory, a demand that most standard devices cannot meet, particularly for even longer sequences. However, the high memory cost can be avoided without significant performance loss if we perform an average of 2.69 splits on long sequences in the adopted datasets, as shown in our experiments.

While it would be possible to leverage another step of representation learning to devise a strategy to aggregate several representations, we instead opt for less computationally expensive approaches. In order to adapt to the variability of vector numbers in different `Smali` classes, as shown in Figure 4.3, we propose to aggregate these sequence embedding vectors of size 128 into one single vector of size 128 by performing simple element-wise average, element-wise addition, or random selection. Another method we investigate is a general approach to vector reshaping in computer vision: concatenation & bilinear resizing. Specifically, we first concatenate the embedding vectors into a long vector and resize it with the bilinear interpolation algorithm [191]. We investigate these four methods in Section 4.4.5 and show that these simple methods are effective.

4.2.3.2 Prediction Model

As shown in the right box in Figure 4.3, after embedding aggregation, we finally obtain one embedding vector for each `Smali` class. The last step is to apply the learned embedding to downstream tasks. Typically, this can be done by feeding the embeddings to another independent neural network that can be trained to a specific task. In the original BERT, they only add one additional layer following the pre-trained model for each downstream task.

In our approach, we design a simple model architecture with only three fully connected layers of neural network as the task-specific model and freeze the parameters of the pre-trained DexBERT model when tuning for specific tasks. The computational cost in training downstream tasks is significantly decreased by only tuning parameters

of the task-specific model. Specifically, the parameter number of the task-specific model is 10.4K, which is almost negligible compared with 459.35M of the pre-trained DexBERT model.

The input of the task-specific model is the aggregated vector of class embedding, as shown in Figure 4.3. The task-specific model for malicious code localization predicts whether what a given class contains is malicious or not. Similarly, for defect detection, the task-specific model predicts if a given class contains defective code. For component type classification, the task-specific model predicts to which component type the given class belongs. The details of each task-specific setup are presented in Section 4.3.3.

4.3 Study Design

In this section, we first overview the research questions to investigate in Section 4.3.1. Then, details about dataset and empirical setup are presented in Section 4.3.2 and Section 4.3.3. We provide evaluation results and answer the research questions in Section 4.4.

4.3.1 Research Questions

In this chapter, we consider the following six main research questions:

- **RQ1:** Can DexBERT accurately model Smali bytecode?
- **RQ2:** How effective is the DexBERT representation for the task of Malicious Code Localization?
- **RQ3:** How effective is the DexBERT representation for the task of Defect Detection?
- **RQ4:** How effective is the DexBERT representation for the task of Component Type Classification?
- **RQ5:** What are the impacts of different aggregation methods of instruction embeddings?
- **RQ6:** Can DexBERT work with subsets of instructions?

4.3.2 Dataset

In this work, we rely on four different datasets and present them in this section.

4.3.2.1 Dataset for Pre-training

With DexBERT, we target a general-purpose representation for various Android analysis tasks. To obtain a representative sampling of the diverse landscape of android apps, we opted to leverage the dataset of a recent work in Android malware detection. One thousand apps—malware or benign—are randomly selected from the dataset used to evaluate DexRay [19], a work that collected more than 158 000 apps from the AndroZoo dataset [29]. Class de-duplication was performed in order to include as much diversity as possible without letting the total number of instructions explode. After removing duplicate classes, our selection of APKs results in over **35 million** `Smali` instructions, from which we obtained **556 million tokens**. This is comparable to the scale of BooksCorpus [192], one of the pretraining datasets used in the original BERT, which consists of 800 million tokens.

Despite the pre-training dataset of DexBERT being smaller than the original BERT, its sufficiency is supported by two factors. Firstly, `Smali`, being an assembly language, possesses a simpler structure and a significantly smaller set of tokens

compared to natural languages and high-level programming languages, implying that a smaller dataset is enough to capture its essential features. Secondly, the efficiency of the dataset is evaluated based on DexBERT's performance in downstream tasks which use APKs from distinct sources than the pre-training dataset. The superior performance of DexBERT over baseline models in these tasks confirms the adequacy of the pre-training dataset.

Based on the `Smali` instructions in the dataset, we generated a WordPiece [193] vocabulary of 10 000 tokens for DexBERT, which is only one-third the size of the original BERT vocabulary. The WordPiece model employs a subword tokenization method to manage extensive vocabularies and handle rare and unknown words. It breaks down words into smaller units, effectively addressing out-of-vocabulary words.

4.3.2.2 Dataset for Malicious Code Localization

RQ2 deals with Malicious Code Localization, *i.e.*, finding what part(s) of a given malware contains malicious code. At least two existing works have tackled this challenging problem for Android Malware and thus have acquired a suitable dataset with ground-truth labels. In Mystique [194], Meng *et al.* constructed a dataset of 10 000 auto-generated malware, with malicious/benign labels for each class. However, almost all of the code in these generated APKs is either malicious or from commonly used libraries (such as `android.support`), and thus may not be representative of existing apps, nor of the diversity of Android apps. More recently, in MKLDroid [26], Narayanan *et al.* randomly selected 3000 apps from the Mystique dataset and *piggybacked* the malicious parts into existing, real-world benign apps from Google Play, resulting in a dataset they named MYST. Although still a little far from the real-world scenario, the repackaged malware in the MYST dataset contains both malicious and benign classes, which can support a class-level malicious code localization task. We thus decide to rely on the MYST dataset to conduct our experiments related to RQ2. Note that, to the best of our knowledge, no fully labeled dataset of real-world malware exists for the task of malicious code localization for Android. Note also that in our work, we choose MKLDroid as a baseline work, enabling us to directly compare DexBERT against MKLDroid.

Despite the challenges in acquiring labeled real-world malware, we remained determined to evaluate DexBERT's performance in real-world scenarios. Ultimately, we succeeded in constructing a dataset that, albeit not extensive, contains labeled real-world malicious classes, thus broadening our evaluation scope. Specifically, we found 46 apps in the Difuzer [195] dataset, where the locations of a specific malicious behavior, namely the logic bomb, have been manually labeled. This allows us to obtain labels of malicious classes and thereby assess DexBERT's ability to localize malicious code in real-world applications. In addition, the authors of Difuzer provided more apps from their subsequent work, and we were able to successfully download and process 88 apks in total.

Given that each APK in the Difuzer dataset only has one class labeled as containing a logic bomb, and the malicious or benign nature of other classes is unknown, we are only able to utilize 88 malicious classes from these 88 real-world APKs for our extended evaluation. To facilitate a more comprehensive evaluation process, we constructed a dataset with additional APKs. The training set comprises three sources: 50 Difuzer APKs with logic bombs, 50 benign APKs from the DexRay dataset [19], and 100 APKs from MYST dataset to augment the dataset size. Please

note that we selectively choose a portion of the benign classes at random to prevent a significant data imbalance, as the initial number of benign classes is much larger than that of malicious classes. As a result, we acquired 1929 benign classes and 425 malicious classes, including 50 from Difuzer APKs, for fine-tuning the classifier. The evaluation set, aimed at testing DexBERT on real-world APKs, consists solely of the remaining 38 APKs with logic bombs from Difuzer and 50 benign APKs from DexRay. We ended up with 95 benign classes (almost two per benign apks) and 38 malicious classes containing logic bombs. Please be aware that all the aforementioned designs aim to make the constructed data suitable for deep learning model training, while ensuring that there is no overlap between the training and evaluation sets.

4.3.2.3 Dataset for App Defect Detection

As another important class-level Android analysis task, app defect detection is the subject of **RQ3**. Dong *et al.* proposed `smali2vec` as a deep neural network based approach to detect application defects and released a dataset containing more than 92K Smali class files collected from ten Android app projects in over fifty versions. For the convenience of labeling, they collected these APKs from GitHub based on three project selection criteria: 1) the number of versions is greater than 20; 2) the package size is greater than 500 KB; and 3) a large number of commits and of contributors. The defective `Smali` files are located and labeled with Checkmarx [196], a widely used commercial static source code analysis tool. Finally, each `Smali` class in the dataset has a label indicating whether it is defective or not. We choose `smali2vec` as our dataset for app defect detection to enable comparison with their `smali2vec` approach built specifically for defect detection.

4.3.2.4 Dataset for Component Type Classification

To further evaluate the universality of DexBERT, we introduced a third class-level task called Component Type Classification. This task, distinctly different from the previous two, is designed to provide a comprehensive assessment of DexBERT's applicability across various scenarios. In the Android framework, four primary components exist, namely Activities, Services, Broadcast Receivers, and Content Providers. These are fundamental building blocks of an Android application and are declared in an application's manifest file (`AndroidManifest.xml`). Therefore, we can readily obtain labels for these four types of component classes and formulate a high-quality dataset for this task. Note that this task was designed solely to demonstrate the universality of DexBERT; It is selected due to the different nature of this task from other two downstream tasks and the ease and speed with which ground truth can be obtained. We randomly selected 1000 real-world APKs from the AndroZoo repository [29], from which we extracted 3406 component classes with accurate labels. We used 75% of this data for training and the remaining 25% for testing.

4.3.3 Empirical Setup

In this section, we introduce the empirical settings adopted in the pre-training task and three downstream tasks.

4.3.3.1 Pre-training

Based on the typical BERT [175] design, we simplify the model architecture of DexBERT to a certain extent to reduce the computational cost. Indeed, while

the dimension of intermediate layers in the position-wise feed-forward network was originally defined as $H \times 4$, where H is set to 768 by default, as mentioned in Section 4.2.2.3, we reduce this dimension to $H \times 3$, *i.e.*, from 3072 to 2304. The number of hidden layers and heads in the multi-head attention layers are set to 8 instead of 12. With these simplifications, the number of floating-point operations (FLOPs, indicating the computational complexity of the model) is reduced by 43.9%, from 44.05G to 24.72G. Meanwhile, the number of model parameters is only decreased by 7.7%, from 497.45M to 459.35M. Thus we keep as many as possible of the model parameters while reducing the computational cost, with the goal of preserving the learning ability of the model as much as possible.

The batch size is set to 72, and the learning rate is set to $1e^{-4}$. Following the reference implementation[1] we leveraged, we select the Adam optimizer [197]. We adopt the Cross-Entropy loss function[2] for both the *masked words prediction* task and the *next sentence prediction* task, and the Mean Squared Error (MSE) loss function[3] for the Auto-Encoder task. Particularly, the MSE Loss is a criterion that measures the squared L2 norm between each element in the input x and target y. Thus, the loss value of this task could also be regarded as an evaluation metric for the Auto-Encoder task.

The pre-training of DexBERT on 556 million tokens for 40 epochs took about 10 days on 2 Tesla V100 GPUs (each with 32G memory). However, it is important to note that DexBERT users do not need to pre-train the model from scratch. They can directly use the pre-trained model we provide for their own Android analysis tasks.

4.3.3.2 Malicious Code Localization

As discussed, malicious code localization is a difficult and under-explored problem. Therefore, there are few widely recognized approaches and benchmark datasets available. To the best of our knowledge, Narayanan *et al.* provided the first well-labeled dataset that comes close to real-world practical needs, to evaluate their multi-view context-aware approach MKLDroid [26] for malicious code localization.

In order to fairly compare with this baseline work, we follow the same validation strategy, *i.e.*, 2000 APKs for fine-tuning and the remaining 1000 APKs for evaluation. We use 3-fold cross validation to ensure the reliability of the results. The prediction model consists of 3 fully connected layers, of with 128, 64, and 32 neurons, respectively. The output layer consists of two neurons that are used to predict the probabilities for a class being either malicious or benign. We leverage the best aggregation method (*i.e.*, element-wise addition) found in **RQ5**. When fine-tuning for this task, we use the Cross-Entropy loss function and the Adam optimizer. With a batch size of 256, we train the prediction model for 40 epochs. For evaluation metrics, we adopt Precision, Recall, False Positive Rate (FPR), and False Negative Rate (FNR), the same as the baseline work MKLDroid for easy comparison. We further report F1-Scores as the overall metric.

4.3.3.3 App Defect Detection

Similar to malicious code localization, app defect detection is a relatively new challenge, only addressed by a small number of works in the literature. Particularly,

[1] `https://github.com/dhlee347/pytorchic-bert`

[2] `https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html`

[3] `https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html`

Dong *et al.* proposed a DNN-based approach `smali2vec` [180] targeting Android applications and released a benchmark dataset containing more than 92K Smali class files. The classifier of `smali2vec` has 810,600 weight parameters across 10 layers, each of which has 300 neurons. In contrast, our model only comprises a total of 10,304 weight parameters spanning three simple layers with sizes of 128, 64, and 32.

In this work, we follow their *Within-Project Defect Prediction* (WPDP) strategy using 5-fold cross-validation to compare their approach with ours. They provide an AUC score for each project, and report their mean value as the final evaluation metric. In addition, we also report the weighted average score to cater to the significant size variations among projects. For this task, we adopt the same model architecture, aggregation method, loss function, and training strategy as for the malicious code localization task.

4.3.3.4 Component Type Classification

The majority of the empirical settings for the Component Type Classification task are identical to those used in Malicious Code Localization, with the exception of the number of neurons in the output layer of the prediction model. In this task, the classifier contains four neurons, instead of two, to output the probabilities for the four different component types.

4.4 Experimental Results

In this section, we present the evaluation results of DexBERT, and we answer our five research questions.

4.4.1 RQ1: Can DexBERT accurately model Smali bytecode?

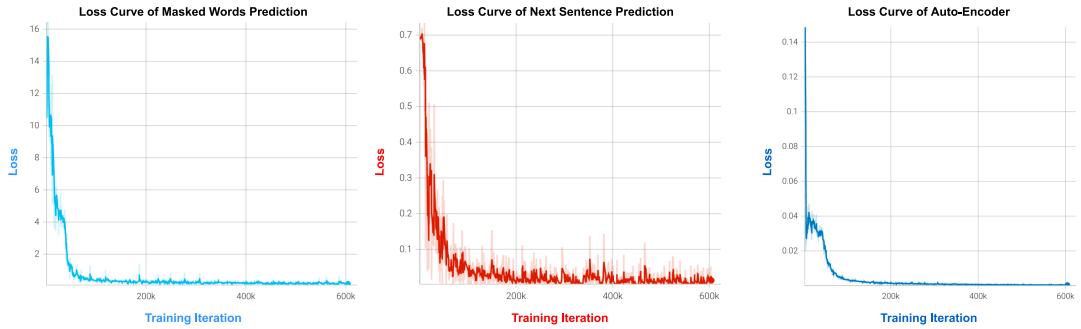

Figure 4.4: Loss Curves of Three Pre-training Tasks. The X axis represents the training iteration index and Y axis represents the loss value.

With this first RQ, we assess whether or not DexBERT is able to learn from `Smali` bytecode and build an accurate model of `Smali` bytecode as used in Android apps. To that end, we report the loss curves of the three pre-training tasks, presented in Sections 4.2.2.2 and 4.2.2.3. These loss curves are shown in Figure 4.4, where the X axis represents the training iterations (i.e., batches).

Two elements allow us to confidently conclude that DexBERT indeed can learn an accurate model of `Smali` bytecode. First, the loss for all three pre-training tasks rapidly drops and is already very low after being fed just a small portion of our pre-training dataset, suggesting that our dataset is more than large enough for our

purpose. Second, continuing the pre-training process results in even lower loss and does not generate random fluctuations, suggesting that the model learned is not contradicted by new pieces of `Smali` bytecode and indeed converges.

As mentioned in Section 4.3.3.1, the MSE loss could also be regarded as an evaluation metric, and its loss value in the third curve in Figure 4.4 approaches zero late in the training process, indicating that the Auto-Encoder is able to reconstruct the given input vector of DexBERT embedding with minimal error. Therefore, the output vector of Encoder (*i.e.*, Hidden Embedding in Figure 4.2) preserves the key information of the original DexBERT representation, which is required for the representation reconstruction by Decoder.

Table 4.1: Evaluation of Pre-training Tasks. The masked words prediction task is evaluated on 2 037 400 tokens and the next sentence prediction task is evaluated on 101 870 instruction pairs.

Task	Accuracy	# of Samples
Masked Words Prediction	95.30%	2 037 400
Next Sentence Prediction	99.35%	101 870

In order to further evaluate the other two pre-training tasks, we created an **evaluation set** containing 2 037 400 masked tokens for the masked words prediction task and 101 870 instruction pairs for next sentence prediction task. We calculate accuracy on the evaluation set as a metric to further evaluate the performance of DexBERT on these two tasks.

Given the initial imbalanced distribution of each token and the randomness of our selection process, the distribution of masked tokens was imbalanced. Common characters, strings, or variables such as slash "/" (15.47%), comma "," (6.32%), "v0" (1.83%), "object" (1.73%), and "lcom" (1.67%) were most frequently masked. Nevertheless, less common characters ($< 1.00\%$) still accounted for 42.69% of the masked tokens. As shown in Table 4.1, with an accuracy of 95.30% on over two million predictions, we believe DexBERT's performance in the MLM task is robust. Regarding the NSP task, the distribution of positive and negative samples was well balanced; positive samples constituted 49.81% of the total samples, and negative samples made up the remaining 50.19%. DexBERT achieved a near-perfect accuracy of 99.35% on over 100K instruction pairs. This suggests that DexBERT was able to learn accurate features for the NSP task. The high accuracy of these two tasks demonstrates that the learned representations contain key features of the input instruction sequences leading to correct predictions.

RQ1 Answer: DexBERT can learn an accurate model of Smali bytecode.

4.4.2 RQ2: How effective is the DexBERT representation for the task of Malicious Code Localization?

In this section, we investigate the performance of DexBERT on the malicious code localization task and compare it with the MKLDroid baseline work [26] on their evaluation dataset. Following their experimental setup, we fine-tune DexBERT to

output, for each class of a given app, a *maliciousness score*, or *m-score* for short. MKLDroid was evaluated with a beam search strategy, with a width of 10. In Table 4.2, we show MKLDroid performance metrics as reported in the MKLDroid paper [26], followed by the performance metrics of DexBERT (Line *DexBERT-m*), also computed with a beam search of width 10.

Table 4.2: Performance of Malicious Code Localization on the MYST dataset.

Approach	F1-Score	Precision	Recall	FNR	FPR
MKLDroid	0.2488	0.1434	0.9400	0.0500	0.1700
smali2vec	0.9916	0.9880	0.9954	0.0046	0.0046
DexBERT-m	0.5749	0.4034	**1.0000**	**0.0000**	0.4847
DexBERT	**0.9981**	**0.9983**	0.9979	0.0021	**0.0006**

Additionally, we perform an experiment where we evaluate DexBERT without beam search, where each class is predicted as malicious if the m-score is above a threshold of 0.5 or benign otherwise. The performance metrics for this experience are reported in the last line of Table 4.2. In effect, when evaluated in the same conditions as MKLDroid, DexBERT significantly outperforms MKLDroid with an F1 score 0.9981 on the MYST dataset. Therefore, DexBERT does not need beam-search at all and achieves excellent performance when classifying each class independently. Furthermore, we also include `smali2vec` [180] as an additional baseline, which, although it achieves fairly good performance, fails to outperform our proposed DexBERT.

As noted in Section 4.3.2.2, we expanded our evaluation of DexBERT on real-world Android applications. Employing our dataset constructed from Difuzer apps, i.e., Difuzer Extension dataset, DexBERT achieved a notable F1 score of 0.9048 in identifying malicious classes within real-world APKs. Further, it achieved a commendable F1 score of 0.9560 in predicting benign classes, thereby eliminating our concerns that data imbalance might negatively impact the evaluation.

RQ2 Answer: DexBERT demonstrates significant performance improvements over MKLDroid and achieves a slight edge over `smali2vec` in the task of malicious code localization. Moreover, DexBERT also show its potential on localizing malicious classes within real-world Android apps.

4.4.3 RQ3: How effective is the DexBERT representation for the task of Defect Detection?

Table 4.3: Performance of App Defect Detection

Project # of classes	AnkiDroid 14 767	BankDroid 12 372	BoardGame 1634	Chess 5005	ConnectBot 3865	Andlytics 5305	FBreader 9883	K9Mail 11 857	Wikipedia 18 883	Yaaic 974	Average Score	Weighted Average AUC Score
`smali2vec`	0.7914	0.7967	**0.8887**	0.8481	**0.9516**	0.834	0.8932	0.7655	**0.8922**	**0.9371**	0.8598	0.8399
DexBERT	**0.9572**	**0.9363**	0.7691	**0.9125**	0.8517	**0.9248**	**0.9378**	**0.8674**	0.8587	0.8764	**0.8892**	**0.9032**

In this section, we investigate the performance of DexBERT on the task of app defect detection, and we compare it against the baseline work `smali2vec` [180]. The performance of `smali2vec`[4] on 10 Android projects is shown in Table 4.3, where the

[4]as reported in the `smali2vec` paper [180]

`# of classes` represents the number of `Smali` classes in each project.

Using our DexBERT representation, we fine-tune a model to predict the likelihood that a given class is defective.

As shown in Table 4.3, DexBERT outperforms `smali2vec` on 6 out of 10 projects and achieves a weighted AUC score of 90.32%, which is a 6.33 percentage points improvement over `smali2vec`.

RQ3 Answer: DexBERT slightly outperforms `smali2vec` for the task of app defect detection.

4.4.4 RQ 4: How effective is the DexBERT representation for the task of Component Type Classification?

Table 4.4: Comparison of F1 Score Among Various BERT-like Baselines for Four Component Classes.

Method	Activity	Service	BroadcastReceiver	ContentProvider	Average
BERT	0.8272	0.7642	0.5673	0.9091	0.7669
CodeBERT	0.917	0.5381	0.8756	0.8468	0.7943
DexBERT(woPT)	0.7402	0.5850	0.7660	0.8947	0.7465
DexBERT	**0.9780**	**0.9117**	**0.9600**	**0.9756**	**0.9563**

In this section, we explore DexBERT's performance on the Component Type Classification task, aiming to further examine its universal applicability across diverse application scenarios. We contrast DexBERT's performance against other BERT-like models, specifically BERT [175], CodeBERT [36], and DexBERT without pre-training. Similar to the settings for the Malicious Code Localization task, we fine-tune a classifier to predict a given class's component type. Given that other BERT-like baselines lack an AutoEncoder module for further reduction of embedding dimensionality, we use the first state vector (size 768) of the embedding for all comparative experiments.

As Table 4.4 illustrates, DexBERT excels in predicting all four types of component classes. On average, DexBERT's performance surpasses all baselines by a significant margin, exhibiting a roughly 20 percentage point increase in terms of F1 Score. This reiterates DexBERT's effectiveness in representing Smali instructions in Android and, crucially, validates the universality of DexBERT.

RQ4 Answer: DexBERT significantly outperforms baselines for the task of component type classification which differs from first two tasks that focused on security-related properties, demonstrating its versatility across various application scenarios.

4.4.5 RQ 5: What are the impacts of different aggregation methods of instruction embeddings?

In this section, we investigate the impact of the four embedding aggregation methods (*i.e.*, element-wise addition, random selection, averaging and concatenation & bilinear resizing, cf. Section 4.2.3.1). These techniques were initially leveraged to aggregate token embeddings in BERT [175] and other deep learning approaches [19].

Table 4.5: Comparison of Different Aggregation Methods on Three Downstream Tasks: Malicious Code Localization (MCL), Defect Detection (DD), and Component Type Classification (CTC)

Method	MCL@F1 Score	DD@AUC Score	CTC@F1 Score
Addition	**0.9989**	**0.9064**	**0.9563**
Random	0.9982	0.8553	0.8898
Average	0.9916	0.8712	0.9442
Concat&Resize	0.9979	0.8508	0.7491

From our point of view, the output features from BERT and the AutoEncoder of DexBERT are essentially similar in nature. Each state vector of BERT embedding is a high-level abstract feature of the corresponding token. Similarly, each output vector of Auto-Encoder is a high-level abstract feature vector of the corresponding token sequence. Therefore, if it's plausible to aggregate token embeddings by addition or the other three techniques, it should also be plausible to aggregate sequence embeddings in a similar manner.

We conduct comparative experiments based on three downstream tasks to evaluate to what extent DexBERT is sensitive to the aggregation method. As shown in Table 4.5, on the malicious code localization task, performance metrics for all four methods are very close, with no significant differences between methods. For the other two downstream tasks, we note that the differences are more significant than for the task of malicious code localization. Despite all four aggregation methods yielding an acceptable performance, *addition* is the best performer, achieving the highest metric scores on both tasks.

RQ5 Answer: All four proposed aggregation methods are effective on all three downstream tasks. Element-wise addition achieves the best performance on both tasks.

4.4.6 RQ6: Can DexBERT work with subsets of instructions?

Table 4.6: Comparative Analysis of Full Instructions vs API Calls for Malicious Code Localization (MCL) and Component Type Classification (CTC).“Avg Time” means the average inference time per class.

Method	MCL@F1 Score	CTC@F1 Score	Avg Time
Full Instructions	**0.9981**	**0.9563**	0.00768s
API Call	0.9932	0.8779	**0.00073s**

In the previous RQs, we demonstrated the effectiveness of DexBERT when using the entire `Smali` bytecode. Representing `Smali` bytecode with DexBERT can be computationally expensive, given the very large number of instructions an app (or even a class) can contain. With this RQ, we investigate the ability of DexBERT to work with subsets of instructions, hence reducing the number of pieces of code to represent and reducing the need for aggregation.

We postulate that API invocations are the instructions that carry the most

semantics information, and thus conduct an experiment where we pre-filter the flow of `Smali` bytecode to keep only API calls. Based on the statistics on our pre-training dataset, API instructions constitute approximately 17.27% of the total instructions. We re-use the pre-trained model built with the complete flow of instructions (Cf. RQ 1), but we fine-tune a dedicated model with filtered instructions only. Since many classes in some projects in the dataset for defect detection do not have API calls (some concrete examples are included in appendix), which would result in empty representations, we only consider the tasks of malicious code localization and Component Type Classification here.

As shown in Table 4.6, while the performance of DexBERT is slightly higher with all instructions, DexBERT still performs very well with API calls only. Computationally, however, working with only API calls is one order of magnitude faster. As for the total execution time, we take the evaluation of malicious code localization task as an example here. With full instructions, we require approximately 1.9 hours to generate the DexBERT features for all 911,724 classes. Conversely, with only API calls, we need only about 11 minutes to generate all feature vectors.

RQ6 Answer: When compatible with the downstream task, DexBERT is also very effective and fast when considering API calls only.

4.5 Discussion

In this section, we begin with an ablation study examining the impact of DexBERT's embedding size on downstream tasks, and an ablation study assessing the effectiveness of the two pre-training tasks. Following that, we compare the performance of various BERT-like baselines across three different downstream tasks. Then, we share some insights about the proposed DexBERT for Android representation. Next, we discuss some potential threats to validity of the proposed approach. Finally, we discuss some future works which are worth studying next.

4.5.1 Ablation Study on DexBERT Embedding Size

Table 4.7: Ablation Study on the Impact of DexBERT Embedding Size

Size	MCL@F1 Score	DD@AUC Score	DD@F1 Score	CTC@F1 Score
768	0.9999	0.9699	0.8887	0.9563
256	0.9995	0.9336	0.8029	0.9246
128	0.9981	0.9032	0.7542	0.9202
64	0.9813	0.8472	0.6693	0.9007

As detailed in Section 4.2.2.3, Android application scenarios require a smaller embedding due to the considerably larger token quantities compared to typical textual documents and code files. To find a reasonable trade-off between model computation cost and performance, we conducted an ablation study exploring the impact of DexBERT embedding size on the three downstream tasks. The experiments contain three different sizes for the hidden embedding of the AutoEncoder (AE), specifically 256, 128, and 64. Additionally, we evaluated the performance by directly utilizing the first state vector of the raw DexBERT embedding, which has a size of 768, without applying any dimension reduction from the AutoEncoder.

Table 4.7 reveals that in the task of Malicious Code Localization, a decrease in vector size does not lead to a significant loss in the performance, until the size is reduced to 128. Hence, we concluded that 128 is the optimal size for this task.

As for the tasks of Defect Detection and Component Type Classification, the experimental results demonstrate that a larger embedding size resulted in a considerable improvement in performance. However, a size of 128 also offered a solid trade-off for these two tasks, supporting satisfactory performance with a metric score exceeding 0.9. Please be aware that the choice to use the AUC score for defect detection was made in order to maintain consistency with the metric employed by the primary baseline for this task, namely, `smali2vec` [180]. However, to be consistent with the other two tasks, we have also included the F1 score for this task in Table 4.7.

4.5.2 Ablation Study on Pre-training Tasks

Table 4.8: Comparison of F1 scores on Three Downstream Tasks based on Different Pre-training Task Designs for : Malicious Code Localization (MCL), Defect Detection (DD), and Component Type Classification (CTC).

Pre-training Designs	MCL	DD	CTC
Only MLM	0.9987	0.8055	0.8827
Only NSP	0.6547	0.5331	0.5491
MLM&NSP	**0.9999**	**0.8725**	**0.9563**

To better understand the pre-training process, we conducted an ablation to confirm the necessity and effectiveness of the two pre-training designs, i.e., MLM and NSP, on the final improvement of the model.

In models like BERT and DexBERT, multi-task learning with MLM and NSP is designed to generate universal features for a variety of tasks. Removing either task diminishes the model's representational power. As demonstrated in Table 4.8, while MLM alone can achieve relatively good performance, the combination of both pre-training tasks significantly improves the model's performance, reinforcing their mutual importance for capturing the smali bytecode structure and semantics effectively. The results on all three downstream tasks, especially on Defect Detection (DD) and Component Type Classification (CTC) demonstrate the importance of both MLM and NSP pre-training tasks.

4.5.3 Comparative Study with other BERT-like Baselines

Table 4.9: Comparison of F1 Scores among Various BERT-Like Baselines for Three Tasks: Malicious Code Localization (MCL), Defect Detection (DD), and Component Type Classification (CTC). DexBERT(woPT) indicates DexBERT without Pre-training.

Models	MCL	DD	CC
BERT	0.9182	0.66851	0.7669
CodeBERT	0.9985	0.64775	0.7943
DexBERT(wo-PreT)	0.9961	0.74381	0.3028
DexBERT	**0.9999**	**0.8725**	**0.9563**

To better understand the necessity and effectiveness of the pre-training process on `Smali` code, in this section, we conduct a comparative study to assess the performance of existing BERT-like models that can be directly applied to all three Android downstream tasks without any technical barriers. Specifically, the baselines include BERT [175], CodeBERT [36], and DexBERT without pre-training. With the same reason in Section 4.4.4, we use the first state vector (size 768) of the embedding for all comparative experiments.

The outcomes are shown in Table 4.9. Interestingly, each baseline model performed remarkably well for Malicious Code Localization. This can be attributed to the fact that the dataset is artificially generated by inserting malicious code into real-world apps, resulting in a clear separation between positive and negative samples, making them easy to learn from. However, the pre-trained DexBERT model outperformed the baselines with an impressive F1 score of 0.9999, approaching perfection.

For the other two tasks, Defect Detection and Component Type Classification, where datasets were collected from real-world APKs, the pre-trained DexBERT clearly surpassed BERT and CodeBERT by approximately 20 percentage points on both tasks. Furthermore, the performance of DexBERT without pre-training exhibited low performance and instability across the three tasks, which was somewhat expected as it lacked prior knowledge before being fine-tuned on downstream tasks. Overall, the comparative results shown in Table 4.9 strongly demonstrate the necessity and effectiveness of the DexBERT pre-training process on `Smali` code.

4.5.4 Insights

In this chapter, we find that the popular NLP representation learning model BERT can be used for Android bytecode without much modification by regarding disassembled `Smali` instructions as natural language sentences. While it had already been shown that BERT-like models could be used for Source Code, our work shows it can also work directly with raw apps in the absence of original source code.

Still, there are some gaps between natural language and `Smali` code to mitigate. In particular, NLP problems and Android analysis problems have significantly different application scenarios. NLP problems are usually at the text snippet level, where the base unit is a (short) paragraph (*e.g.*, sentence translation or text classification). However, in Software Engineering and Security for Android applications, there are problems both at the class level and the whole app level, *i.e.*, ranging from a few instructions to millions of instructions. Therefore, embedding aggregation is required when applying instruction embeddings to Android analysis problems.

Besides, while this work is only evaluated on class-level tasks, it is expected to work at other levels (lower or higher). There are no significant technical barriers for lower-level tasks (*i.e.*, method level or statement level) except for the absence of well-labeled datasets. For higher-level tasks (*i.e.*, APK level), further embedding aggregation would be required to support whole-application tasks.

4.5.5 Threats to Validity

Our experiments and conclusions may face threats to validity. First, the MYST dataset used for malicious code localization is artificially created and, therefore may not be representative of the real-world landscape of apps. To mitigate this concern, we expanded our evaluation to include real-world apps from Difuzer [195]. Moreover, the utility of our malicious code localization model in real-world scenarios

could be further assessed by extending its application to a wider variety of malicious behaviours beyond logic bombs. This extension could be an avenue for future work, as it would necessitate significant manual analysis efforts.

Second, the dataset for Android app defect detection was created several years ago. Android applications are constantly evolving over time. How well DexBERT performs on today's apps requires further validation. Therefore, it may be necessary to create new datasets and conduct more comprehensive evaluations on Android app defect detection.

4.6 Summary

In this chapter, we introduced DexBERT, a pre-trained representation learning model designed to address various fine-grained Android analysis tasks. To overcome the input length limitations inherent in standard BERT models, we developed an aggregation method based on Auto-Encoder techniques. By freezing the parameters of the pre-trained DexBERT model, the learned representations can be effectively applied to a range of class-level downstream tasks.

Extensive experimental results demonstrated the superior performance of DexBERT in tasks such as malicious code localization, Android application defect detection, and component type classification, outperforming baseline methods. Additionally, DexBERT has the potential to be adapted for app-level tasks, such as Android malware detection, which will be explored in the upcoming chapters.

5

LaFiCMIL: Rethinking Large File Classification from the Perspective of Correlated Multiple Instance Learning

While DexBERT has advanced class-level Android analysis, its input size limitations pose challenges for app-level tasks like Android malware detection, requiring additional embedding aggregation. Similar limitations are common in BERT-like models in Natural Language Processing (NLP), which struggle with long input sequences, typically processing only partial information from lengthy texts at high computational cost. To address this, we introduce LaFiCMIL, a method designed specifically for large file classification through correlated multiple instance learning (c-MIL). Optimized for efficient training on a single GPU, LaFiCMIL is versatile across binary, multi-class, and multi-label tasks. In this chapter, we implement it using standard BERT models and evaluate its performance across seven benchmark datasets. The results demonstrate LaFiCMIL's efficiency and state-of-the-art performance, marking it as a significant advancement in large file classification. Instead of designing an Android-specific solution, LaFiCMIL is intended to be a versatile framework that can be applied to general BERT-like models, including DexBERT, which will be explored in the following chapter.

This chapter is based on the work published in the following research paper [198]:

- Sun, T., Pian, W., Daoudi, N., Allix, K., F. Bissyandé, T. and Klein, J., 2024, June. LaFiCMIL: Rethinking Large File Classification from the Perspective of Correlated Multiple Instance Learning. In International Conference on Applications of Natural Language to Information Systems (pp. 62-77). Cham: Springer Nature Switzerland.

Contents

5.1 Overview

With the initial goal of enabling DexBERT to process bytecode from an entire app and achieve full app-level representation learning, we found that this is a common challenge faced by BERT-like models in natural language processing (NLP) — dealing with long input sequences. Similar to Android malware detection, where capturing comprehensive app behavior is essential, text classification in NLP also requires assigning appropriate labels to specific input sequences [199]. This process is crucial across various domains, including sentiment analysis [200], fake news detection [201], and offensive language identification [202], among others. Recent years have seen the emergence of attention-based models like Transformer [203], GPT [204, 205], and the BERT family [175, 36, 174], which have established state-of-the-art benchmarks in text classification tasks. However, the challenge of processing very long sequences remains a significant obstacle, mainly due to their high computational requirements when facing extremely huge number of tokens. Thus, rather than designing an Android-specific solution to address DexBERT's input limitations, we aim to propose a versatile framework that can be applied to general BERT-like models.

There are mainly two types of solutions in the literature to address long token sequences: ① extending the input length limit by employing a sliding window to attention, such as Longformer [206], and ② dividing long documents into segments and recurrently processing the Transformer-based segment representations, such as RMT [207, 208]. Nevertheless, Longformer inherently struggles with global context capture. The sliding window mechanism can lead to a fragmented understanding of the overall sequence, as it primarily focuses on local context within each window. This becomes particularly challenging when dependencies span beyond the scope of these localized windows, a common occurrence in complex or extremely long text sequences. Similarly, the recurrent processing of RMT can also lead to information loss, especially when dependencies or context need to be carried over long sequences of text. Each recurrent step has the potential to dilute or overlook critical information from previous segments, leading to a gradual decay in context retention as the sequence progresses.

Recently, significant progress in large language models (LLMs) like GPT-4 [48] and Llama 2 [49] has showcased their impressive capabilities in various NLP tasks. Despite these advances, directly applying large language models (LLMs) to text classification through prompt engineering has not achieved optimal performance, and lightweight Transformer models, such as RoBERTa [45, 209], continue to excel in this essential task. This is demonstrated by RGPT [209], an adaptive boosting framework designed to fine-tune LLMs for text classification. Although RGPT sets new benchmarks, it has only been validated on short texts and demands substantial computational resources, requiring **eight** A100-SXM4-40GB GPUs, and a total of 320 GB of memory. Consequently, developing a more resource-efficient and effective strategy for classifying long documents remains a critical challenge.

In this chapter, we innovatively tackle this challenge by adopting Multiple Instance Learning [133, 134] (MIL), wherein we conceptualize a large file as a 'bag' and its constituent chunks as 'instances' within the MIL framework. We introduce **LaFiCMIL**, a simple yet effective **La**rge **Fi**le **C**lassification approach based on correlated **M**ultiple **I**nstance **L**earning. On the one hand, as proven in Theorem 1 (cf., Section 5.2), a MIL score function for a bag classification task can be approximated

by a series of sub-functions of the instances. This inspires us to split a large file into smaller chunks and extract their features separately using BERT. On the other hand, we aim to guide the model to learn high-level overall features from all instances, rather than deriving the final bag prediction from instance predictions based on a simplistic learned projection matrix. In addition, in contrast to the basic version of MIL [210], where instances within the same bag exhibit neither dependency nor ordering among one another, we claim that the small chunks from the same large file are correlated in some way (e.g., semantic dependencies in paragraphs). This implies that the presence or absence of a positive instance in a bag can be influenced by the other instances contained within the same bag. As a result, relying on our computationally efficient LaFiAttention layer, our approach is capable of **efficiently** extracting **correlations** among **all chunks** as additional information to boost classification performance.

In our evaluation, LaFiCMIL consistently achieved new state-of-the-art performance across all seven benchmark datasets, especially when tested with long documents in the evaluation sets. A notable highlight is LaFiCMIL's performance on the full test set of the Paired Book Summary dataset, where it demonstrated a significant 4.41 percentage point improvement. This dataset is especially challenging as it contains the highest proportion of long documents, exceeding 75%. Furthermore, LaFiCMIL also distinguished itself by having the fastest training process compared to other baseline models.

The contributions of our study are as follows:

- We introduce, LaFiCMIL, a novel approach for large file classification from the perspective of correlated multiple instance learning.
- The training of LaFiCMIL is super efficient, which requires only 1.86× training time than the original BERT, but is able to handle 39× longer sequence on a single GPU with only 32 GB of memory.
- We perform a comprehensive evaluation, illustrating that LaFiCMIL achieves new state-of-the-art performance across all seven benchmark datasets.
- We share the datasets and source code to the community at: `https://github.com/Trustworthy-Software/LaFiCMIL`

5.2 Technical Preliminaries

In this section, we describe several essential technical preliminaries that inspire and underpin the design of LaFiCMIL. We first present a pair of theorems that are the fundamental principles of c-MIL.

Theorem 1. *Suppose $S : \chi \to \mathbb{R}$ is a continuous set function w.r.t Hausdorff distance [211] $d_H(.,.)$. $\forall \varepsilon > 0$, for any invertible map $P : \chi \to \mathbb{R}^n$, $\exists$ function σ and g, such that for any set $X \in \chi$:*

$$|S(X) - g(P_{X \in \chi}\{\sigma(x) : x \in X\})| < \varepsilon \tag{5.1}$$

The proof of Theorem 1 can be found in [133]. This theorem shows that a Hausdorff continuous **set function** $S(X)$ can be arbitrarily approximated by a function in the form $g(P_{X \in \chi}\{\sigma(x) : x \in X\})$. This insight can be applied to MIL, as the mathematical definition of **sets** in the theorem is equivalent to that of **bags** in MIL framework. Consequently, the theorem provides a foundation for approximating bag-level predictions in MIL using instance-level features.

Theorem 2. *The instances in the bag are represented by random variables $\theta_1, \theta_2, ..., \theta_n$, the information entropy of the bag under the correlation assumption can be expressed as $H(\theta_1, \theta_2, ..., \theta_n)$, and the information entropy of the bag under the i.i.d. (independent and identical distribution) assumption can be expressed as $\sum_{t=1}^{n} H(\theta_t)$, then we have:*

$$\begin{aligned} H(\theta_1, \theta_2, ..., \theta_n) &= \sum_{t=2}^{n} H(\theta_t | \theta_1, \theta_2, ..., \theta_{t-1}) + H(\theta_1) \\ &\leq \sum_{t=1}^{n} H(\theta_t) \end{aligned} \tag{5.2}$$

The proof of Theorem 2 can be found in [133]. This theorem demonstrates that the information entropy of a bag under the correlation assumption is smaller than the information entropy of a bag under the i.i.d. assumption. The lower information entropy in c-MIL suggests reduced uncertainty and the potential to provide more information for bag classification tasks. In Section 5.3.1, we introduce c-MIL, and in Section 5.3.2, we derive the efficient LaFiCMIL based on c-MIL.

Next, we present the necessary preliminaries for our efficient attention layer inspired by the Nyströmformer [212], referred as LaFiAttention, which performs as a sub-function within our LaFiCMIL. In the original Transformer [203], an input sequence of n tokens of dimensions d, $X \in \mathbf{R}^{n \times d}$, is projected using three matrices $W_Q \in \mathbf{R}^{n \times d_q}$, $W_K \in \mathbf{R}^{n \times d_k}$, and $W_V \in \mathbf{R}^{n \times d_v}$, referred as query, key, and value respectively with $d_k = d_q$. The outputs Q, K, V are calculated as

$$Q = XW_Q,\ K = XW_K,\ V = XW_V \tag{5.3}$$

Therefore, the self-attention can be written as:

$$D(Q, K, V) = SV = softmax(\frac{QK^T}{\sqrt{d_q}})V \tag{5.4}$$

Then, the softmax matrix S used in self-attention can be written as

$$S = softmax(\frac{QK^T}{\sqrt{d_q}}) = \begin{bmatrix} A_S & B_S \\ F_S & C_S \end{bmatrix} \tag{5.5}$$

where $A_S \in \mathbf{R}^{m \times m}$, $B_S \in \mathbf{R}^{m \times (n-m)}$, $F_S \in \mathbf{R}^{(n-m) \times m}$, $C_S \in \mathbf{R}^{(n-m) \times (n-m)}$, and $m < n$.

In order to **reduce** the memory and time **complexity** from $O(n^2)$ to $O(n)$, LaFiAttention approximates S by

$$\hat{S} = softmax(\frac{Q\widetilde{K}^T}{\sqrt{d_q}})A_S^+ softmax(\frac{\widetilde{Q}K^T}{\sqrt{d_q}}), \tag{5.6}$$

where $\widetilde{Q} = [\widetilde{q_1}; ...; \widetilde{q_m}] \in \mathbf{R}^{m \times d_q}$ and $\widetilde{K} = [\widetilde{k_1}; ...; \widetilde{k_m}] \in \mathbf{R}^{m \times d_q}$ are the selected landmarks for inputs $Q = [q_1; ...; q_n]$ and $K = [k_1; ...; k_n]$, A_S^+ is the Moore-Penrose inverse [213] of A_S.

Lemma 1. *For $A_S \in \mathbf{R}^{m \times m}$, the sequence $\{Z_j\}_{j=0}^{j=\infty}$ generated by [214],*

$$Z_{j+1} = \frac{1}{4} Z_j (13I - A_S Z_j (15I - A_S Z_j (7I - A_S Z_j))) \tag{5.7}$$

converges to Moore-Penrose inverse A_S^+ in the third-order with initial approximation Z_0 satisfying $\left\| A_S A_S^+ - A_S Z_0 \right\| < 1$.

LaFiAttention approximates A_S^+ by Z^* with Lemma 1. Following the empirical choice from [212], we run 6 iterations to achieve a good approximation of the pseudoinverse. Then, the softmax matrix S is approximated by

$$\hat{S} = softmax(\frac{Q\widetilde{K}^T}{\sqrt{d_q}})Z^* softmax(\frac{\widetilde{Q}K^T}{\sqrt{d_q}}). \tag{5.8}$$

5.3 Approach

In this section, we first introduce customized c-MIL for large file classification and then provide technical details about our LaFiCMIL approach.

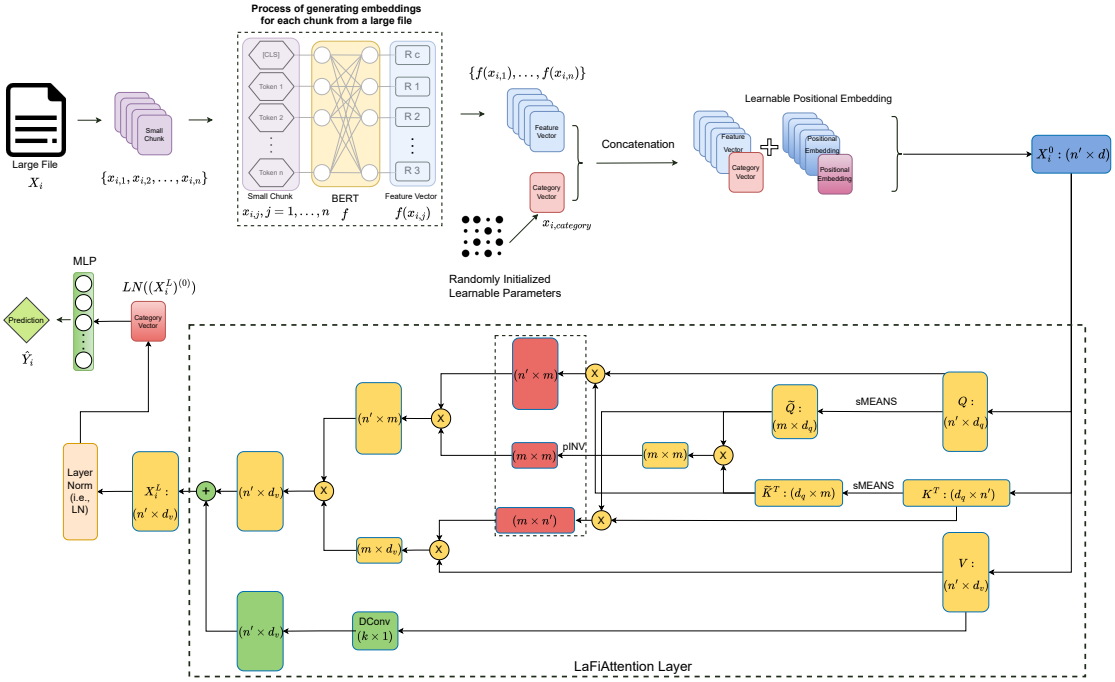

Figure 5.1: LaFiCMIL. Initially, document chunks are transformed into embedding vectors using BERT. A learnable category vector is then concatenated to these embeddings to form an augmented bag X_i^0 with $n' = n+1$ instances. The LaFiAttention layer captures the inter-instance correlations within X_i^0. Operations within this layer, such as matrix multiplication ($\times$) and addition ($+$), are specified alongside the variable names and matrix dimensions. Key processes include sMEANS for landmark selections similar to [3], pINV for pseudoinverse approximation, and DConv for depth-wise convolution. Classification is completed by passing the learned category vector through a fully connected layer.

5.3.1 Correlated Multiple Instance Learning

Unlike traditional supervised classification, which predicts labels for individual instances, Multiple Instance Learning (MIL) predicts bag-level labels for bags of instances. Typically, individual instance labels within each bag exist but inaccessible, and the number of instances in different bags may vary.

In the basic MIL concept [210], instances in a bag are independent and unordered. However, correlations among chunks of a large file exist due to the presence of semantic dependencies between paragraphs. According to Theorem 2, these correlations can

be exploited to reduce uncertainty in prediction. In other words, this relationship can be leveraged as additional information to boost the performance of long document classification tasks. The Correlated Multiple Instance Learning (c-MIL) is defined as below.

Here, we consider a binary classification task of c-MIL as an example. Given a bag (i.e., a large file) X_i composed of instances (i.e., chunks) $\{x_{i,1}, x_{i,2}, ..., x_{i,n}\}$, for $i = 1, ..., N$, that exhibit dependency or ordering among each other. The bag-level label is Y_i, yet the instance-level labels $\{y_{i,1}, y_{i,2}, ..., y_{i,n}\}$ are not accessible. Then, a binary classification of c-MIL can be defined as:

$$Y_i = \begin{cases} 0, & if \sum y_{i,j} = 0 \quad y_{i,j} \in \{0,1\}, j = 1, ..., n \\ 1, & otherwise \end{cases} \tag{5.9}$$

$$\hat{Y}_i = S(X_i), \tag{5.10}$$

where S is a scoring function, and $\hat{Y}$ is the predicted score. N is the total number of bags, and n is the number of instances in the ith bag. The number n generally varies for different bags.

5.3.2 LaFiCMIL

According to Theorem 1, we leverage Multi-layer Perceptron [187], BERT [175], LaFiAttention Layer and Layer Normalization [215] as **sub-functions to approximate** the c-MIL score function S defined in Equation 6.2.

Given a set of bags $\{X_1, ..., X_N\}$, where each bag X_i contains multiple instances $\{x_{i,1}, ..., x_{i,n}\}$, a bag label Y_i, and a randomly initialized category vector $x_{i,category}$, the goal is to learn the maps: $\mathbb{X} \to \mathbb{T} \to \gamma$, where $\mathbb{X}$ is the bag space, $\mathbb{T}$ is the transformer space and γ is the label space. The map of $\mathbb{X} \to \mathbb{T}$ can be defined as:

$$\begin{aligned} X_i^0 =& [x_{i,category}; f(x_{i,1}); ...; f(x_{i,n})] \\ &+ E_{pos}, \quad X_i^0, E_{pos} \in \mathbb{R}^{(n+1)\times d} \end{aligned} \tag{5.11}$$

$$\begin{aligned} Q^l = X_i^{l-1} W_Q, \quad K^l = X_i^{l-1} W_K, \quad V^l = X_i^{l-1} W_V, \\ l = 1, ..., L \end{aligned} \tag{5.12}$$

where function f is approximated by a BERT model, E_{pos} is the Positional Embedding, and L is the number of Multi-head Self-Attention (MSA) block.

$$\begin{aligned} head &= LaFiSA(Q^l, K^l, V^l) \\ &= softmax(\frac{Q^l(\widetilde{K}^l)^T}{\sqrt{d_q}}) Z^{*l} softmax(\frac{\widetilde{Q}^l(K^l)^T}{\sqrt{d_q}}) V^l, \end{aligned} \tag{5.13}$$

$$MSA(Q^l, K^l, V^l) = Concat(head_1, ..., head_h) W_O, \tag{5.14}$$

$$X_i^l = MSA(LN(X_i^{l-1})) + X_i^{l-1}, \quad l = 1, ..., L \tag{5.15}$$

where $W_O \in \mathbb{R}^{hd_v \times d}$, $head \in \mathbb{R}^{(n+1)\times d_v}$, $LaFiSA$ denotes the approximated Self-attention layer by Nyström method [216] according to Equation 5.8, h is the number of head in each MSA block, and Layer Normalization(LN) is applied before each MSA block.

The map of $\mathbb{T} \to \gamma$ can be simply defined as:

$$\hat{Y}_i = MLP(LN((X_i^L)^{(0)})), \tag{5.16}$$

where $(X_i^L)^{(0)}$ represents the learned category vector, and MLP means Multi-layer Perceptron (i.e., fully connected layer).

From the above formulation, we can find that the most important part is to efficiently learn the map from bag space $\mathbb{X}$ to Transformer space $\mathbb{T}$. As illustrated in Figure 5.1, this map is approximated by a series of sub-functions which are approximated by various neural layers. The overall process is summarized as follows: given a large file, we use a BERT model to generate the representations of the divided chunks (i.e., instances in the concept of c-MIL). Then, we initialize a **learnable** category vector that follows a normal distribution and has the same shape as each instance. By considering the category vector as an additional instance, we learn the correlation between each instance using LaFiAttention layer. With the help of the attention mechanism, the category vector exchanges information with each chunk and extracts necessary features for large file classification. Finally, the category vector is fed into a fully connected layer to finalize the classification task.

5.4 Experimental Setup

Table 5.1: Statistics on the datasets. # BERT Tokens indicates the average token number obtained via the BERT tokenizer. % Long Docs means the proportion of documents exceeding 512 BERT tokens.

Dataset	Type	# Total	# Train	# Val	# Test	# Labels	# BERT Tokens	% Long Docs
Hyperpartisan	binary	645	516	64	65	2	744.18±677.87	53.49
20NewsGroups	multi-class	18846	10182	1132	7532	20	368.83±783.84	14.71
Book Summary	multi-label	12788	10230	1279	1279	227	574.31±659.56	38.46
-Paired	multi-label	6393	5115	639	639	227	1148.62±933.97	75.54
EURLEX-57K	multi-label	57000	45000	6000	6000	4271	707.99±538.69	51.3
-Inverted	multi-label	57000	45000	6000	6000	4271	707.99±538.69	51.3
Devign	binary	27318	21854	2732	2732	2	615.46±41917.54	39.76

Datasets. To ensure a fair comparison with baselines, we adopt the same benchmark datasets utilized in the state-of-the-arts for long document classification [217]. We first evaluate LaFiCMIL on these six benchmark datasets: ① Hyperpartisan [218], a compact dataset encompassing 645 documents, designed for *binary classification*. ② 20NewsGroups [219], comprising 20 balanced categories and 11 846 documents. ③ CMU Book Summary [188], tailored for *multi-label classification*, contains 12 788 documents and 227 genre labels. ④ Paired Book Summary [217], formulated by combining pairs of documents from the CMU Book Summary dataset, features longer documents. ⑤ EURLEX-57K [220], a substantial *multi-label classification* dataset consisting of 57 000 EU legal documents and 4271 available labels. ⑥ Inverted EURLEX-57K [217], a modified version of EURLEX-57K dataset in which the order of sections is inverted, ensuring that core information appears towards the end of the document. To evaluate our method's effectiveness with longer documents, we incorporated the Devign dataset [189] for code defect detection. It features documents that are substantially longer than those in the other six datasets, with lengths approaching or exceeding 20,000 tokens. Table 5.1 provides details of the datasets, covering metrics such as the average token count and its standard deviation, the maximum and minimum token counts, and the percentage of large documents, etc.

Implementation Details. We split a long text document into chunks (i.e., c-MIL instances), and follow the standard BERT input length (i.e., 512 tokens) for

each chunk. To ensure a fair comparison, in line with the baselines, we employ BERT [175] for the first six datasets, and CodeBERT [36] and VulBERTa [221] for Devign, as the feature extractor. Since we treat all chunks in each long document as a mini-batch, the actual batch size varies depending on the number of chunks in the long document. We construct LaFiAttention layer with eight attention heads. Under these configurations, a single Tesla V-100 GPU with 32GB of memory on an NVIDIA DGX Station can fully process 100% of the large documents in the first six benchmark datasets and 99.92% of those from Devign. As a result, the average inference time (0.026s) of each mini-batch is almost the same as BERT (0.022s). During training, the Adam optimizer [197] is leveraged. The loss function varies depending on specific task. Following the baseline [217], we use sigmoid and binary cross entropy for binary and multi-label classification, and softmax and cross entropy loss for multi-class classification. For the Hyperpartisan, Book Summary, EURLEX-57K, and Devign datasets, a learning rate of `5e-6` is used, while `5e-7` is applied for the 20NewsGroups dataset. We fine-tune the model for 10, 20, 40, 60, and 100 epochs on Devign, Hyperpartisan, 20NewsGroups, EURLEX-57K, and Book Summary, respectively.

Evaluation Setup. We evaluate the performance of LaFiCMIL using the same metrics as those employed in the baselines [217, 221]. We report the accuracy (%) for binary and multi-class classification. We use micro-F1 (%) for multi-label classification, and detection accuracy (%) for code defect detection. The results are the average of five independent runs, each with a different random seed. As we explained in the introduction, our work aims to address this challenge by focusing on using a more practical resource: a single GPU with 32 GB of memory, which is only a tenth of what RGPT demands (i.e., 320 GB). Therefore, given the distinct resource requirements and application contexts, comparing our approach with LLMs falls outside the scope of this study.

5.5 Experimental Results

In this section, we analyze the performance of LaFiCMIL in long document classification. We first discuss the overall performance, followed by an computational efficiency analysis and an ablation study on the core concepts of LaFiCMIL.

5.5.1 Overall Performance

Table 5.2: Performance metrics on complete test set. The highest score in each column is bolded and underlined, while the second highest score is only bolded.

Model	Hyperpartisan	20News	EURLEX	-Inverted	Book	-Paired
BERT	92.00	84.79	73.09	70.53	58.18	52.24
-TextRank	91.15	84.99	72.87	71.30	58.94	55.99
-Random	89.23	84.65	**73.22**	**71.47**	**59.36**	56.58
Longformer	**95.69**	83.39	54.53	56.47	56.53	**57.76**
ToBERT	89.54	**85.52**	67.57	67.31	58.16	57.08
CogLTX	94.77	84.62	70.13	70.80	58.27	55.91
RMT	94.34	82.87	71.46	70.99	57.30	56.95
LaFiCMIL	**96.92**	**85.07**	**73.72**	**72.03**	**61.34**	**62.17**

Table 5.3: Performance metrics on only long documents in test set. The highest score is bolded and underlined, while the second highest score is only bolded. The subsequent tables of this task are organized in a consistent manner.

Model	Hyperpartisan	20News	EURLEX	-Inverted	Book	-Paired
BERT	88.00	86.09	66.76	62.88	60.56	52.23
-TextRank	85.63	85.55	66.56	64.22	61.76	56.24
-Random	83.50	**86.18**	**67.03**	**64.31**	**62.34**	56.77
Longformer	**93.17**	85.50	44.66	47.00	59.66	**58.85**
ToBERT	86.50	-	61.85	59.50	61.38	58.17
CogLTX	91.91	86.07	61.95	63.00	60.71	55.74
RMT	90.04	83.62	64.16	63.21	60.62	58.27
LaFiCMIL	**95.00**	**87.49**	**67.28**	**65.04**	**65.41**	**63.03**

Our experimental results reveal a phenomenon similar to [217] in that no existing approach consistently outperforms the others across all benchmark datasets. However, as shown in Table 5.3, our LaFiCMIL establishes new state-of-the-art performance on all six NLP benchmark datasets when considering only long documents in the test set. Here, we define a long document as one containing over 512 BERT tokens. As shown in Table 5.2, we also achieve new state-of-the-art performance on five out of six NLP datasets when considering the full data (i.e., a mix of long and short documents) in the test set. Particularly, we significantly improve the state-of-the-art score from 57.76% to 62.17% on the Paired Book Summary dataset, which contains the highest proportion of long documents (i.e., more than 75%). In contrast, we fail to achieve the best performance on 20NewsGroups, as the proportion of long documents in this dataset is very small (only 14.71%); thus, our improvement on **long** documents (as shown in Table 5.3) cannot dominate the overall performance on the entire dataset. This phenomenon is consistent with our motivation that the more long documents present in the dataset, the more correlations LaFiCMIL can extract to boost classification performance.

Table 5.4: Accuracy (%) comparison of different models on Devign dataset for code defect detection. The highest accuracy score is bolded and underlined, and the base model results are only bolded.

RoBERTa	CodeBERT	Code2vec [222]	PLBART [223]	VulBERTa	CodeBERT+ LaFiCMIL	VulBERTa+ LaFiCMIL
61.05	**62.08**	62.48	63.18	**64.27**	63.43	**64.53**

Given that 100% of long documents from the six NLP datasets can be fully processed, we conduct an additional evaluation of LaFiCMIL's ability to process **extremely long** sequences, based on the code defect detection dataset Devign. Our findings reveal that LaFiCMIL is capable of handling inputs of up to nearly 20 000 tokens when utilizing CodeBERT [36] and VulBERTa [221] as feature extractors on a **single** GPU setup. This capability allows for 99.92% of the code files in the Devign dataset to be processed in their entirety. Concurrently, as shown in Table 5.4, LaFiCMIL enhances the performance of both CodeBERT and VulBERTa, establishing a new state-of-the-art in accuracy over the evaluated baselines. We find

that code defect detection is a challenging task on which most existing state-of-the-art models struggle to achieve even a single percentage point improvement over previous models. Nonetheless, our LaFiCMIL helps CodeBERT gain 1.35 percentage points, representing a significant improvement.

5.5.2 Computational Efficiency Analysis

Table 5.5: Runtime and memory requirements of each model, **relative to BERT**, based on the Hyperpartisan dataset. Training and inference time were measured and compared in seconds per epoch. GPU memory requirement is in GB.

Model	Train Time	Inference Time	GPU Memory
BERT	1.00	1.00	<16
-TextRank	1.96	1.96	16
-Random	1.98	2.00	16
Longformer	12.05	11.92	32
ToBERT	1.19	1.70	32
CogLTX	104.52	12.53	<16
RMT	2.95	2.87	32
LaFiCMIL	1.86	1.18	<32

In this section, we provide a comprehensive analysis of computational efficiency outlined in Table 5.5. All models were evaluated on a single GPU with 32GB of memory using the Hyperpartisan dataset. LaFiCMIL performs distinctly in this context, demonstrating a runtime nearly on par with BERT. The balance between high computational efficiency and advanced classification capability illustrates LaFiCMIL's exceptional capability to efficiently process long documents without significant computational overhead.

5.5.3 Ablation Study

Table 5.6: Concept ablation study on long documents in test set. "wo" means "LaFiCMIL **w**ith**o**ut".

Model	Hyperpartisan	20News	EURLEX	-Inverted	Book	-Paired
wo BERT	85.00	53.92	60.54	54.14	50.11	46.61
wo LaFiAttn	87.50	84.97	**66.82**	**64.89**	**62.50**	**60.13**
wo c-MIL	**88.00**	**86.09**	66.76	62.88	60.56	52.23
LaFiCMIL	**95.00**	**87.22**	**67.28**	**65.04**	**65.41**	**63.03**

To gain a comprehensive understanding of the efficacy of each core concept in our approach (namely, BERT, LaFiAttention, and c-MIL), we conduct an ablation study. This study aims to evaluate the classification performance of LaFiCMIL when each concept is systematically removed, allowing us to evaluate their individual contributions.

Without a feature extractor, any approach would be ineffective. Thus, when BERT is removed, the LaFiAttention layer must assume the role of feature extractor

instead of c-MIL. This would result in the disappearance of the c-MIL mechanism, and the approach can now only take the first chunk as input, transforming it into a basic attention-based classifier. As might be expected, the absence of the BERT concept leads to the worst performance across all datasets among the three variants, as shown in the first row of Table 5.6. If excluding the LaFiAttention concept, c-MIL devolves into a standard MIL, for which we employ the widely accepted Attention-MIL [210]. The results of this setting are presented in the second row of Table 5.6. Given that this variant can still process all chunks of a lengthy document, it performs best among all three variants on the four datasets with the largest number and longest length of documents. When the c-MIL concept is removed, the LaFiAttention layer will also be absent as it executes c-MIL which is no longer needed, leaving only BERT. Due to its restriction to process only the first chunk as input, this variant fails to achieve the best results on the four datasets that are dominated by long documents. Finally, upon comparing the three variants with the full LaFiCMIL, shown in the fourth row of Table 5.6, it becomes evident that the exclusion of any concept significantly weakens performances across all datasets.

5.6 Summary

In this chapter, we introduced LaFiCMIL, a novel framework for large file classification based on correlated multiple instance learning (c-MIL). By treating large document chunks as instances, LaFiCMIL enables efficient feature extraction from correlated sections without significant information loss. Our experimental results demonstrate that LaFiCMIL significantly outperforms state-of-the-art baselines across multiple benchmark datasets in terms of both efficiency and accuracy. This work offers a new perspective on addressing the challenges of large file classification by leveraging c-MIL.

Rather than developing an Android-specific approach, LaFiCMIL provides a versatile framework that can be applied to general BERT-like models for large-scale data processing. Additionally, LaFiCMIL can be adapted for Android malware detection by enabling DexBERT to generate full-app-level representations of Android applications, which will be explored in the following chapter.

6

DetectBERT: Towards Full App-Level Representation Learning to Detect Android Malware

Recent advancements in machine learning (ML) and deep learning (DL) have improved Android malware detection, but many existing approaches still fail to fully capture the complexity of malicious behaviors. DexBERT, a BERT-like model designed for Android representation learning and introduced in Chapter 4, enhances class-level analysis by processing Smali code extracted from APKs. However, its ability to handle multiple Smali classes simultaneously remains limited. In this chapter, we introduce DetectBERT, an adaptation of the LaFiCMIL framework, which combines DexBERT with Correlated Multiple Instance Learning (c-MIL) to address the high dimensionality and variability inherent in Android malware detection. By treating class-level features as instances within c-MIL bags, DetectBERT aggregates them into robust app-level representations, allowing for more effective malware detection at the app level. Our evaluations demonstrate that DetectBERT not only outperforms state-of-the-art methods but also shows adaptability to evolving malware threats.

This chapter is based on the work published in the following research paper [224]:

- Sun, T., Daoudi, N., Kim, K., Allix, K., Bissyandé, T.F. and Klein, J., 2024. DetectBERT: Towards Full App-Level Representation Learning to Detect Android Malware. In 2024 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM).

Contents

6.1 Overview

The ubiquity of Android devices in today's digital ecosystem has made them a prime target for malicious actors. As the sophistication of Android malware continues to evolve, so does the need for effective detection mechanisms. Traditional malware detection approaches, while beneficial, often struggle to keep pace with the rapid development of new and complex malware apps [70, 93]. This challenge is exacerbated by the dynamic and open nature of the Android platform, which allows for the frequent introduction of new applications and updates.

Recent advancements in machine learning [22, 98] (ML) and deep learning [99, 100, 19] (DL) have offered promising directions for enhancing Android malware detection. These techniques, capable of learning from large datasets to identify subtle patterns and anomalies, have shown their potential across a broad range of software engineering applications. However, they largely depend on static analysis [104, 109, 110], extracting basic features or low-level bytecode from APK files [19, 111], and analyzing function call graphs [112, 113]. While these approaches showcase some effectiveness, they often fall short in capturing the nuanced and complex behaviors that characterize new emerging malware apps.

In Chapter 4, we introduced DexBERT [174], a pre-trained BERT [175]-like model specifically designed for class-level Android representation learning. By learning the Smali code disassembled from APKs, DexBERT demonstrated its capability of significantly enhancing performance in class-level Android analysis tasks, such as malicious code localization. Nevertheless, DexBERT is constrained by its input limitations, allowing it to analyze only a single Smali class at a time, preventing it from providing a comprehensive app-level understanding.

A similar challenge in whole-slide image classification has been addressed by TransMIL [133] using correlated Multiple Instance Learning (c-MIL). Following this idea, in Chapter 5, we extended c-MIL to develop LaFiCMIL [198] to address the problem of large file classification. Both TransMIL and LaFiCMIL enhance the ability to extract valuable features from a wide array of cropped image patches and segmented document fragments, which are analogous to the multiple classes present within an Android APK. Recognizing this parallel, we adapt c-MIL principles and extend LaFiCMIL to enable full app-level representation learning for Android malware detection, resulting in the development of DetectBERT. This model sets itself apart from TransMIL and LaFiCMIL by:

- Employing Smali classes directly as natural instances within the c-MIL framework, avoiding the need for preliminary data processing such as image cropping or document splitting.
- Omitting positional embeddings due to the non-sequential nature of Smali class interconnections, which are based on invocation links.
- Freezing DexBERT's weights during training to leverage the pre-trained model's original capabilities for generating class embeddings, thus conserving computational resources.

To implement c-MIL for our scenario, DetectBERT employs the Nyström Attention layer [212], a technique that efficiently learns the relationships among embedding vectors. Specifically, upon generating class embeddings via DexBERT, we introduce a learnable category vector that is initialized to conform to a normal distribution and is dimensionally compatible with the class embeddings. Subsequently, both

the category vector and all class embeddings are processed through the Nyström Attention layer. This mechanism helps DetectBERT learn the intricate correlations among the vectors, enabling a dynamic exchange of information between the category vector and each class embedding. Such an exchange is pivotal for highlighting features essential to malware detection. The process enriches the category vector, which is then passed through a fully connected layer, serving as the foundation for the ultimate malware detection decision.

We conduct an extensive evaluation on a large dataset of 158 803 applications. The experimental results demonstrate that DetectBERT not only surpasses three basic feature aggregation methods but also outperforms two state-of-the-art Android malware detection techniques. Furthermore, we conduct a temporal consistency evaluation to assess how well DetectBERT performs over time, especially when confronted with newer malware samples absent in the training data. This assessment reaffirms the robustness of DetectBERT, demonstrating its sustained effectiveness against emerging threats and highlighting its relevance in dynamic real-world scenarios. This research bridges the gap between sophisticated representation learning models and the practical demands of app-level malware analysis. Looking forward, DetectBERT showcases significant potential to expand its utility in software engineering, particularly by using Multiple Instance Learning (MIL) to efficiently process and analyze large-scale inputs. This capability is crucial for advancing software quality and security assessments in complex environments like mobile and IoT ecosystems.

The contributions of our study are as follows:

- We introduce DetectBERT, an efficient and effective approach utilizing MIL to scale DexBERT for full app-level representation learning in Android malware detection.
- We perform a comprehensive evaluation, demonstrating DetectBERT's superiority over both basic feature aggregation methods and state-of-the-art malware detection techniques, including its robustness through a temporal consistency evaluation to verify performance against new, unseen malware samples.
- We highlight DetectBERT's potential to revolutionize software engineering, specifically through its use of MIL to manage large-scale data, offering significant improvements in software assessments.
- To facilitate replication, we have made the dataset and source code available at:
  `https://github.com/Trustworthy-Software/DetectBERT`

6.2 Approach

In this section, we start with the key theoretical foundations of our method, then detail the design of DetectBERT.

6.2.1 Theoretical Foundations

Recalling Theorem 1 from Chapter 5, it shows that a Hausdorff continuous **set function** $S(X)$ can be arbitrarily approximated by a function in the form $g(P_{X\in\chi}\{\sigma(x) : x \in X\})$. This concept applies to MIL, where the theorem's sets correspond to MIL's bags. Consequently, the theorem provides a foundation for approximating bag-level predictions in MIL using instance-level features. In the context of Android malware detection, this principle instructs us to achieve app-level predictions through the utilization of class-level embeddings produced by DexBERT.

Referring to Theorem 2 from Chapter 5, it demonstrates that a bag under the correlation assumption has lower information entropy than that under the i.i.d. assumption (i.e., in standard MIL). The lower information entropy in c-MIL suggests reduced uncertainty and the potential to provide more valuable information for bag classification tasks than the standard MIL. In the context of Android malware detection, we claim that the Smali classes from the same APK are correlated in some way (e.g., invocation relationships between classes). This implies that the presence or absence of a malicious class in a bag can be influenced by the other classes contained within the APK. Therefore, c-MIL seems a perfect choice to leverage the class embeddings of DexBERT for the app-level task of Android malware detection.

Given an app X_i composed of Smali classes $\{x_{i,1}, x_{i,2}, ..., x_{i,n}\}$, for $i = 1, ..., N$, that exhibit correlation among each other. The app-level label is Y_i, yet the class-level labels $\{y_{i,1}, y_{i,2}, ..., y_{i,n}\}$ are not accessible. Then, detecting Android malware in the context of c-MIL can be defined as:

$$Y_i = \begin{cases} 0, & if \sum y_{i,j} = 0 \quad y_{i,j} \in \{0,1\}, j = 1, ..., n \\ 1, & otherwise \end{cases} \tag{6.1}$$

$$\hat{Y}_i = S(X_i), \tag{6.2}$$

where S is a scoring function and $\hat{Y}$ is the predicted score indicating the likelihood of the analyzed application being malware. N is the total number of applications, and n is the number of Smali classes in the ith application. The number n generally varies for different applications.

6.2.2 DetectBERT

According to Theorem 1 from Chapter 5, we leverage a series of **sub-functions to approximate** the c-MIL score function S defined in Equation 6.2. Given a set of APKs $\{X_1, ..., X_N\}$, where each APK X_i contains multiple Smali classes $\{x_{i,1}, ..., x_{i,n}\}$ and an APK label Y_i, a corresponding category vector $x_{i,category}$ is randomly initialized. We represent DexBERT [174] as a feature extracting function f, Multi-layer Perceptron [187] as MLP, Nyström Attention Layer [212] as NAL and Layer Normalization [215] as LN. The c-MIL score function S can be approximated as follows:

$$X'_i = [x_{i,category}, f(x_{i,1}), ..., f(x_{i,n})] \tag{6.3}$$

$$x'_{i,category} = (NAL(LN(X'_i)) + X'_i)^{(0)} \tag{6.4}$$

$$\hat{Y}_i = MLP(LN(x'_{i,category})) \tag{6.5}$$

The selection of the Nyström Attention layer in DetectBERT's architecture is driven by its capability to efficiently handle large sequences of data. First, as an attention mechanism, Nyström Attention naturally facilitates the learning of correlations among class embeddings, treating them akin to tokens in natural language processing. This enables DetectBERT to dynamically assess and prioritize the relevance of each Smali class' features based on their contextual relationship, crucial for accurate malware detection. Secondly, when confronted with a vast number of tokens—typical in complex Android applications—the Nyström Attention layer proves significantly more efficient than traditional attention mechanisms. Its design

reduces computational complexity from quadratic to linear, making it especially suitable for large-scale datasets where efficiency is paramount. These characteristics make Nyström Attention an ideal choice for DetectBERT.

DetectBERT distinguishes itself from TransMIL and LaFiCMIL in three key aspects. Firstly, unlike the image cropping in TransMIL or document splitting in LaFiCMIL, DetectBERT directly utilizes Smali classes as natural instances for c-MIL without requiring any preprocessing steps. Secondly, our model does not employ positional embeddings, as outlined in Equation 6.3. This decision is based on the observation that Smali classes are interconnected through invocation relationships rather than a sequential order. Lastly, to enhance efficiency, the weight parameters of DexBERT are frozen during training. DetectBERT leverages the DexBERT's original capacity to generate meaningful class embeddings without the need for further fine-tuning, thus avoiding additional computational costs.

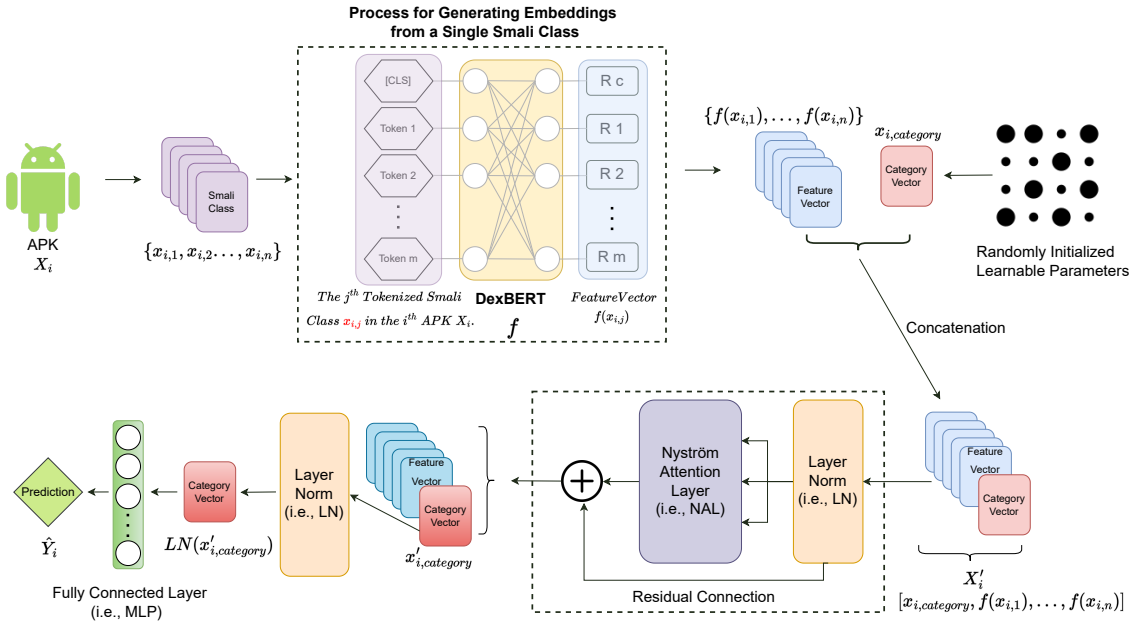

Figure 6.1: Overview of DetectBERT Workflow. First, DexBERT produces Smali class embeddings as c-MIL instances. A category vector of the same size is then introduced as an additional instance. The Nyström Attention layer helps DetectBERT find correlations among instances, allowing the category vector to capture key information from class embeddings for malware detection. Lastly, this vector is processed in a fully connected layer to make the detection decision.

The overall process of DetectBERT is outlined in Figure 6.1 and can be detailed as follows: given an APK X_i containing n Smali classes $\{x_{i,1}, ..., x_{i,n}\}$, we use the pretrained DexBERT model f to generate the corresponding embedding vectors $\{f(x_{i,1}), ..., f(x_{i,n})\}$. Then, we initialize a **learnable** category vector $x_{i,category}$ that conforms to a normal distribution and has the same shape as each class embedding. By considering the category vector as an additional embedding vector, we concatenate it with the class embeddings as $[x_{i,category}, f(x_{i,1}), ..., f(x_{i,n})]$, and learn the correlation between each embedding vector using the Nyström Attention layer. Layer normalization and residual connection [225] are applied to standardize the input and output data of the Nyström Attention layer and facilitate gradient backpropagation,

enhancing model stability and performance. With the help of the attention mechanism, the category vector exchanges information with each class embedding and extracts necessary features for malware detection. Finally, the category vector is fed into a fully connected layer (i.e., MLP) to finalize the detection task.

6.3 Study Design

This section outlines the design of our study, detailing the research questions, dataset, empirical setup, and evaluation metrics used to assess the performance of DetectBERT.

6.3.1 Research Questions

We mainly investigate three research questions, each aimed at exploring different aspects of DetectBERT's performance:

- **RQ1**: How does DetectBERT perform compared to basic feature aggregation methods in detecting Android malware?
- **RQ2**: How does DetectBERT perform compared to state-of-the-art Android malware detection models?
- **RQ3**: How does DetectBERT maintain its detection effectiveness over time in the face of evolving Android malware?

These questions are designed to comprehensively evaluate the effectiveness, competitiveness, and robustness of DetectBERT.

6.3.2 Dataset

For our evaluation, we utilize the large-scale benchmark dataset from DexRay [19], which contains a total of 158 803 apps. This dataset facilitates a direct comparison of DetectBERT against basic aggregation methods and the state-of-the-art Android malware detection approaches. The malware and benign apps in the DexRay dataset are selectively collected from the popular and continually growing Android repository named AndroZoo [29]. Specifically, the DexRay dataset contains 96 994 benign and 61 809 malware apps. Benign apps are defined as the apps that have not been detected by any antivirus from VirusTotal [1], while the malware collection contains the apps that have been detected by at least two antivirus engines.

6.3.3 Empirical Setup

In our empirical setup, we employ the DexBERT model to generate a feature vector for each Smali class. Considering the extensive size of the dataset, we optimize our process by storing these generated feature vectors on disk, thereby significantly reducing computational costs by preventing the need to regenerate feature vectors during each training phase. Consequently, the DetectBERT model requires only 2 GB of GPU memory for training and achieves an average inference time of just 0.005 seconds per app, underscoring its computational efficiency.

To enhance the model's capacity without altering the underlying DexBERT architecture, we freeze DexBERT's parameters and incorporate dual Nyström Attention layers. This approach allows us to expand the model's capabilities while maintaining a stable structure. For the optimization process, we employ the Lookahead optimizer [226] to fine-tune the model. The training is conducted for 20 epochs with a learning rate set to `1e-4`.

[1] `https://www.virustotal.com/`

6.3.4 Evaluation Metrics

For comparison with basic aggregation methods in Section 6.4.1 and comparison with state-of-the-art approaches in Section 6.4.2, we follow the baseline DexRay [19] by shuffling the dataset and split it into 80% for training, 10% for validation, and 10% for test. This process is repeated 10 times to ensure robustness, with the average results reported. In Section 6.4.3, to assess temporal consistency, we temporally partition our dataset, using apps from 2019 for training (90%) and those from 2020 for testing (10%), based on the date information in the app's DEX file. Results are reported using the four metrics: accuracy (%), precision (%), recall (%), and F1 Score (%), with precision maintained to two decimal places for consistency with baseline results.

6.4 Experimental Results

In this section, we evaluate DetectBERT's performance by addressing the three research questions outlined in Section 6.3.1, aiming to provide insights into distinct aspects of its effectiveness.

6.4.1 RQ1: How does DetectBERT perform compared to basic feature aggregation methods in detecting Android malware?

In this subsection, we evaluate DetectBERT's performance in detecting Android malware by comparing it against three fundamental techniques for aggregating DexBERT embeddings: Random Selection, Element-wise Addition, and Element-wise Average. Each technique compresses information from multiple class embeddings into a single representative vector, which is then used as input to a fully connected layer for final prediction.

- **Random Selection**: It randomly selects a single class embedding to represent the entire APK, testing the efficacy of individual class features in malware detection.
- **Element-wise Addition**: It aggregates class embeddings by adding corresponding elements together, aiming to capture the cumulative effect of features across all classes.
- **Element-wise Average**: Similar to addition, it calculates the average of corresponding elements, offering a normalized representation of class features.

Table 6.1: Performance comparison with basic feature aggregation approaches.

Model	Accuracy	Precision	Recall	F1 Score
Random Selection	0.82	0.82	0.82	0.82
Element-wise Addition	0.86	0.87	0.84	0.86
Element-wise Average	0.92	0.92	0.92	0.92
DetectBERT	**0.97**	**0.98**	**0.95**	**0.97**

Table 6.1 shows that while the Element-wise Average method achieves an F1 score of 0.92, indicating the effectiveness of DexBERT embeddings, DetectBERT significantly outperforms this and other basic methods. Notably, DetectBERT attains

superior precision and F1 scores of 0.98 and 0.97, respectively, improving performance by 6 and 5 percentage points over the Element-wise Average method.

This substantial improvement is attributed to DetectBERT's sophisticated architecture, which more effectively captures and utilizes the intricate relationships among class embeddings. Unlike basic aggregation methods that compress embeddings into a singular vector, DetectBERT employs Nyström Attention layers. These layers dynamically adjust and integrate information from different classes based on their contextual relevance, facilitating a more nuanced aggregation. This advanced capability proves crucial in detecting Android malware, where subtle interactions among app features often signify malicious activity.

RQ1 Answer: DetectBERT significantly outperforms basic aggregation methods in detecting Android malware, emphasizing the importance of advanced embedding processing techniques for handling complex Android APKs, thus enhancing the reliability and accuracy of malware detection systems.

6.4.2 RQ2: How does DetectBERT perform compared to state-of-the-art Android malware detection models?

In this subsection, we evaluate the competitive performance of DetectBERT against two established state-of-the-art models in the field of Android malware detection: Drebin [22] and DexRay [19]. Drebin is known for its application of machine learning techniques to hand-crafted features, while DexRay utilizes a deep learning framework to analyze low-level bytecode images.

Table 6.2: Performance comparison with existing state-of-the-art approaches.

Model	Accuracy	Precision	Recall	F1 Score
Drebin	0.97	0.97	0.94	0.96
DexRay	0.97	0.97	0.95	0.96
DetectBERT	**0.97**	**0.98**	**0.95**	**0.97**

According to the findings in Table 6.2, DetectBERT not only matches but also exceeds the precision and F1 scores of the well-established Drebin and DexRay models on the challenging DexRay dataset. It elevates the precision score to 0.98, improving upon the already high score of 0.97 achieved by both Drebin and DexRay while maintaining high recall. These slight yet significant improvements in precision and F1 scores are critical as they highlight DetectBERT's exceptional ability to pinpoint malware accurately without missing genuine threats. This performance enhancement is primarily due to DetectBERT's effective use of sophisticated c-MIL mechanisms. These mechanisms dynamically adjust based on the relationships among class embeddings, enabling DetectBERT to capture subtle nuances and connections within the Smali classes. This capability distinguishes it from other models that rely on conventional feature analysis techniques and may overlook such intricate relationships. This nuanced detection capability is key to DetectBERT's success, offering deeper insights into the complex behaviors typical of modern malware.

RQ2 Answer: DetectBERT slightly surpasses the performance of state-of-the-art models Drebin and DexRay. This superior performance still demonstrates the effectiveness of its architecture in leveraging MIL to enhance detection accuracy in complex malware datasets.

6.4.3 RQ3: How does DetectBERT maintain its detection effectiveness over time in the face of evolving Android malware?

Model aging [227, 228] is a significant challenge in machine learning, where a model's performance tends to degrade when applied to new, previously unseen samples. In this section, we assess the robustness of DetectBERT against model aging by evaluating its temporal consistency, a measure of how well a model trained on historical data performs against recent threats. The methodology for this temporal consistency evaluation is detailed in Section 6.3.4, where the dataset is divided into training sets from 2019 and testing sets from 2020 to accurately emulate real-world applications against evolving malware. The performance of DetectBERT and other state-of-the-art models is compared using these temporally distinct datasets.

Table 6.3: Temporal consistency performance comparison with state-of-the-art approaches.

Model	Accuracy	Precision	Recall	F1 Score
Drebin	0.96	0.95	0.98	0.97
DexRay	0.97	0.97	0.98	0.98
DetectBERT	**0.99**	**0.99**	**0.99**	**0.99**

Table 6.3 showcases the evaluation results, indicating that all models tested exhibit commendable adaptability to new malware samples. Notably, DetectBERT excels, achieving superior accuracy, precision, recall, and F1 scores of 0.99 across all metrics. This performance not only surpasses the other state-of-the-art models but also demonstrates significant improvements over the scores reported in Table 6.2, where the earlier data sample composition varied. This discrepancy highlights the exceptional capability of DetectBERT to generalize from past data to future scenarios without significant performance degradation. The increased proportion of training data from 2019 in this setup (90% compared to the previous 80%) likely played a significant role in enhancing the robustness of DetectBERT. This broader learning base enabled it to more effectively identify malware characteristics that persist or evolve over time, contributing to its heightened effectiveness.

DetectBERT's standout performance in this temporal consistency evaluation confirms its effectiveness in handling new malware threats, a key advantage for maintaining relevance in the rapidly evolving landscape of Android security. This high level of adaptability stems from DetectBERT's sophisticated representation learning capabilities, which extract and leverage the high-level semantics from Smali code in Dex files. Unlike simpler models that might rely on surface-level features, DetectBERT deeply understands the underlying behaviors and patterns encoded in the application's code, allowing it to detect even subtly disguised malware.

This adaptability is critical for practical deployment, where the ability to perform consistently over time is paramount.

RQ3 Answer: DetectBERT exhibits exceptional temporal consistency in malware detection, effectively mitigating model aging issues and demonstrating robust generalization capabilities to new and evolving threats. These qualities position our model as an invaluable asset in the ongoing battle against Android malware.

6.5 Summary

In this chapter, we introduced DetectBERT, a framework adapted from LaFiCMIL that extends DexBERT's capabilities for app-level Android malware detection. By utilizing a correlated Multiple Instance Learning (c-MIL) strategy, DetectBERT surpasses traditional feature aggregation methods and outperforms existing state-of-the-art malware detection models. These improvements suggest that DetectBERT has the potential to set a new standard for Android security analysis, representation learning, and other software engineering tasks.

Part III

Temporal-Incremental Malware Learning

Within the scope of Android malware learning, beyond detection, a key challenge is malware classification, which involves categorizing identified malware into families or types. However, this task is significantly affected by concept drift—the performance of malware classifiers degrades over time as malware evolves. As new malware families emerge and the behaviors of existing families shift, traditional classifiers struggle to maintain accuracy. In this part, we explore the critical problem of Android malware family classification in a temporal-incremental learning scenario. We address the challenges posed by concept drift, focusing on how to adapt classification systems to handle the continuous evolution of malware families and changes in the data distribution over time.

7

Temporal-Incremental Learning for Android Malware Classification

Malware classification is critical in cybersecurity, particularly due to the constantly evolving nature of malicious software. To keep up with this evolution, traditional approaches often require frequent retraining on historical data, which is both time-consuming and resource-intensive. Class-Incremental Learning (CIL) provides an alternative by progressively learning new malware classes while preserving previous knowledge. However, CIL assumes no overlap between classes across time intervals, which is unrealistic as malware families persist across multiple intervals. To address this limitation, we introduce Temporal-Incremental Malware Learning (TIML), a formulation that adapts to shifts in malware family distributions. We utilize the MalNet dataset, comprising over a million entries of Android malware across a decade, organized in chronological order. First, we adapt state-of-the-art CIL methods to TIML and evaluate their performance. Then, we propose a novel multimodal TIML approach that integrates information from multiple malware modalities. Extensive evaluations show that our TIML approaches are significantly superior than the original CIL approaches, and reveal it is feasible to periodically update malware classifiers at a very low cost. This update process is efficient, incurring minimal storage and computational overhead, albeit at the expense of a slight dip in performance compared with retraining classifiers with full historical data.

This chapter is based on the work published in the following research paper [229]:

- Sun, T., Daoudi, N., Pian, W., Kim, K., Allix, K., Bissyandé, T.F. and Klein, J., 2024. Temporal-Incremental Learning for Android Malware Detection. ACM Transactions on Software Engineering and Methodology.

Contents

7.1 Overview

The ever-evolving landscape of malware poses significant challenges for professionals and researchers from both the cybersecurity domain [230, 231, 232] and the software engineering field [22, 233, 19]. As malicious actors continuously craft newer and more sophisticated malware variants, traditional static classification models often fall short, unable to adapt dynamically to these emerging threats [116, 118, 119]. In the era of rapidly advancing digital threats, there is a pressing need for dynamic, efficient, and adaptive solutions to recognize and categorize newly emerging malware families and adapt to shifts in data distribution within known malware families. One commonly adopted strategy to address this evolving challenge involves periodically retraining malware classifiers using the complete repository of historical data during each model update [162, 234]. However, while this approach may initially seem effective, it imposes significant training time overhead and intensifies data storage demands. Indeed, the growing volume and complexity of accumulated data increase training time and storage requirements substantially, leading to escalating costs and data management complexities in terms of pre-processing, analysis, and learning iterations. Moreover, in many real-world settings, retaining historical data for full retraining is not feasible, whether due to privacy protection policies, data security concerns, or storage limitations. In such contexts, the full retraining approach, which requires access to all past data, becomes impractical if not impossible.

Class-incremental learning (CIL) has emerged as a potential solution to address these challenges associated with continuously evolving data streams [157, 159]. At its core, CIL aims to iteratively enhance the model's knowledge as new classes are introduced, relying mainly on data from current classes and minimizing the dependence on data from older classes. This paradigm ensures efficient use of computational resources and offers the potential for timely model adaptability. However, CIL operates on the assumption that each class, or in the context of our study, each malware family, appears uniquely in a distinct time interval, with no overlap across sequential time intervals. The concept is that once the model is trained on a malware family, it will not be retrained on its newly emerged malware samples that will appear in future updates. Unfortunately, this fundamental assumption does not align with the reality of malware evolution, which involves two types of concept drift: ① the emergence of new malware families, and ② shifts in the data distribution of old families. Each type has its unique challenges, and their negative impacts on the effectiveness of malware classification models are detailed in Section 7.6.1. A malware family, rather than appearing in a fixed time period, often has an extended life cycle. This longevity means that even after a model is trained on a particular malware family (included in the training data of the current time interval), new samples from that family may appear in subsequent time intervals. Consequently, samples from the same malware family can exhibit considerable shifts in data distribution if they are from diverse time periods. This nuanced complexity of malware dynamics renders conventional CIL approaches inadequate.

To address the aforementioned challenge, we introduce Temporal-Incremental Malware Learning (TIML). This learning problem is formulated to account for the aforementioned challenge and is thus tailored to the practical needs of evolving malware classification. TIML acknowledges the dynamic nature of malware families and their extended lifecycles, considering the presence of these families across multiple

time intervals and adapting to variations in data distribution across instances. TIML builds upon foundational incremental learning principles, customized to address the complexities posed by malware classification. Figure 7.1 provides an illustration of time steps in the context of TIML, which depicts *when the model should be retrained*. In the example depicted in this diagram, each time step indicates the endpoint of a time interval when the model requires (re)training—whether for initial training or due to a deterioration in performance that necessitates updates. Each time interval corresponds to the life spans of specific malware families, highlighting the need for model updates to integrate new malware samples and maintain robustness against evolving malware signatures.

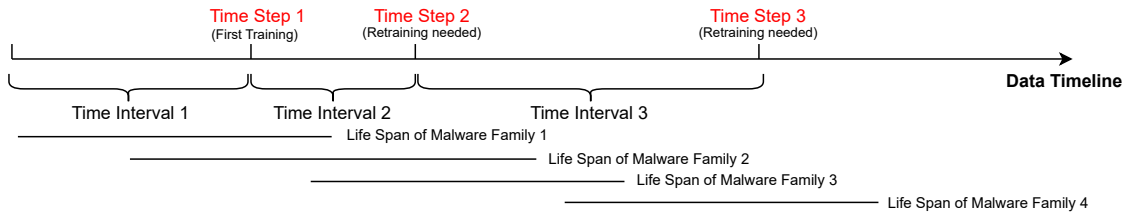

Figure 7.1: The TIML Framework's Retraining Schedule, depicting the specific time steps when the model should undergo retraining based on the chronological emergence of malware families.

In practical terms, the TIML problem formulation allows to reason about efficient model updates. Indeed it sets constraints on the need to retain acquired knowledge and adapt it in the case of concept drift without the need to train on the entire historical dataset. TIML therefore targets solutions that minimize resource and time requirements, leading to models that are ready for deployment in a timely and cost-effective manner. To better illustrate the reality of the problem, we consider a simple real-world scenario. Specifically, we assume that we have a malware classifier trained on samples from numerous families. The classifier can indeed efficiently assign a malware label to unseen malware samples that belong to families seen in the training process. However, after a few months, new malware emerged (i.e., samples exhibiting the two types of concept drift), which impacted the performance of the malware classifier. Therefore, the model needs to be retrained on these new samples so it can preserve its performance. Retraining the model on the full dataset (i.e., newly collected samples and samples previously used to train the model) is impractical as it requires huge resources and time (e.g., the retraining can take more than one month based on our dataset). Consequently, updating the model based primarily on newly emerged samples seems a suitable solution. This strategy ensures that the model update is cheap as storing the full dataset from previous classes is not required. Moreover, the training time is fast, which would make the model ready for deployment in a timely manner.

To address the TIML problem, we first adapt several state-of-the-art CIL approaches to ensure their compatibility with this new framework. Then, we propose a novel multimodal TIML approach to capitalize on the detailed information available in MalNet [28] images and Malscan [235] call graphs. This dual-modality approach is designed to improve malware classification by combining the visual cues from images with the behavioral patterns from call graphs, providing a richer context for classifying both known and emerging malware threats with greater precision. To

start our evaluation, the prerequisite is a robust and comprehensive dataset. For that, we consider MalNet [28], which is a benchmark dataset for malware classification containing over **1.2 million** Android malware from nearly seven hundred families. However, its current structure does not inherently meet the requirements of the TIML setting. Therefore, we carefully organized it according to the chronological order of the emergence of malware samples. Finally, we conduct comprehensive experiments from various perspectives on the task of Android malware detection to demonstrate that TIML-based malware classifiers can be dynamically updated with minimal storage and computational burdens. Our findings reveal the effectiveness of both the adapted TIML methods and our novel multimodal approach in classifying malware families within the TIML paradigm. The results indicate a slight dip in performance during subsequent time steps when compared to models retrained on the entire dataset, a consequence of the limited access to historical data. However, this decline is counterbalanced by the substantial reduction in training time and data storage requirements that TIML approaches offer compared to the comprehensive retraining method. To illustrate, the adapted TIML approach, "LwF with Exemplar", showcases a notable efficiency, reducing training time and data storage demands by 81% and 78% respectively, while only experiencing a modest 6.9 percentage points decline in accuracy. This underscores the operational efficiency of TIML approaches, highlighting their potential for real-world applications where resource optimization is crucial. Through these explorations, our ultimate goal is to pave the way for more robust, adaptive, and efficient malware classification systems in an increasingly volatile digital world.

The contributions of this chapter are as follows:

- We formulate TIML: Temporal-Incremental Malware Learning, an overlooked problem which is introduced to dynamically accommodate shifts in data distributions of malware families and to efficiently classify emerging threats while reducing resource utilization.
- We provide a meticulously restructured **million-scale** dataset of Android malware tailored for the TIML paradigm, serving as a valuable resource for future research endeavors.
- We adapt four CIL approaches to address the TIML problem. The results show that TIML approaches effectively classify malware families with limited resources.
- We introduce a novel multimodal TIML approach that enhances classification performance by effectively utilizing multiple malware data modalities.
- To ease replication, we share the dataset and source code with the community at the following address: `https://github.com/Trustworthy-Software/TIML`

7.2 Background

This section provides an overview of the context and challenges addressed by our study. We begin with a general background discussing the role of deep learning in malware classification and the concept of concept drift. Following this, we delve into the technical background necessary for our proposed approaches.

7.2.1 General Background

This subsection explores the integration of deep learning techniques in malware classification and discusses the implications of concept drift in this domain.

7.2.1.1 Deep Learning in Malware Classification

Our study tackles a critical issue in software engineering: the evolution and classification of malware, a problem underscored by the dynamic nature of software threats and the necessity for robust defensive mechanisms. In the era of advanced artificial intelligence, the software engineering community has increasingly adopted deep learning techniques, resulting in significant advancements across various sub-domains, including malware classification. Specifically, recent seminal works such as MetaMAMC [236], XMal [237], MalCertain [238], AMCDroid [239], and API2Vec++ [240] have introduced significant deep learning innovations to tackle the challenges of malware classification. These contributions highlight the vibrant and ongoing efforts within the software engineering community to harness deep learning for developing more effective solutions in malware defense. Our research contributes to this ongoing trend by addressing specific gaps and introducing novel methodologies that further enhance the capability of deep learning in the context of malware classification.

7.2.1.2 Concept Drift

Concept drift represents a significant challenge in malware classification [31, 32, 241], arising from the dynamic and evolving nature of malware threats. Conceptually, it refers to the changes in the underlying patterns of data that models are trained to predict. In the context of malware, there are primarily two types of concept drift:

- **Emergence of New Malware Families**: This type of drift occurs when entirely new categories of malware are developed. These new families often exhibit unique behaviors and characteristics that were not present in the training data, making them difficult for existing classifiers to detect without updates.
- **Shifts in Data Distribution of Old Families**: Even when the malware family remains the same, its manifestations can evolve. This type of drift involves subtle changes in the distributions of feature vectors of known malware families. Variations in these distributions can be due to factors like new attack vectors, enhanced obfuscation techniques, or other evolutionary changes in malware behavior. These changes can gradually or suddenly render previously effective detection models obsolete by altering the underlying data patterns that the models were trained to recognize.

Our study is dedicated to rigorously investigating the presence and impact of these two types of concept drift using the large-scale MalNet dataset, which contains 1.2 million real-world malware samples. This comprehensive dataset provides a robust foundation for observing and analyzing how malware evolves over time.Following our investigation, we explore methodologies to efficiently and effectively update malware classifiers. This involves developing strategies that not only quickly adapt to new threats as they emerge but also recalibrate the model parameters to account for the subtle shifts in known malware distributions, thereby maintaining high levels of accuracy and responsiveness in malware detection systems.

7.2.2 Technical Background

In this section, we introduce two important concepts that are integral to our proposed approaches.

7.2.2.1 Exemplar Set

An Exemplar Set is a key concept in incremental learning, which involves selecting a representative subset of data from previously encountered classes to aid in the training of new models without revisiting the entire historical dataset [144, 141]. In the context of TIML, exemplar sets play a pivotal role by enabling the model to retain crucial information about older malware families while incorporating knowledge from newly emerging threats, thus preventing catastrophic forgetting. To align with the terminology used in the incremental learning literature, we refer to the process of training the model on a dataset from a specific time step as a *Task*.

Exemplar Set Management: With the evolving nature of the data stream, managing the exemplar set becomes crucial. Two predominant strategies emerge in the standard incremental learning [242]. Transferring to malware family classification, we have:

1. *Fixed Exemplars Per Family:* This approach maintains a constant count of exemplars for each family. Consequently, as the number of malware families increases, the size of the exemplar set expands. However, this leads to a linear increase in memory usage, making it unsustainable for real-world learning systems.

2. *Fixed Total Exemplars:* This strategy focuses on maintaining a constant total number of exemplars across all families. Specifically, the model allocates $\left\lfloor \frac{M}{|\mathcal{Y}|} \right\rfloor$ exemplars per family, where M is the fixed total number of exemplars and $|\mathcal{Y}|$ represents the total number of known malware families up to the most recent task. The $\lfloor \cdot \rfloor$ denotes the floor function. By standardizing the total memory used for exemplars, this approach ensures a consistent and manageable memory footprint, addressing storage constraints effectively.

In our research, we adopt the second strategy for exemplar set management.

Exemplar Selection: Exemplars are representative instances from each known family, chosen from the entire training set. A straightforward approach to determine the exemplars is to sample them randomly for each family, ensuring instance diversity. In contrast, another prevalent method in standard class-incremental learning is the herding technique [145, 243], which seeks to pinpoint the most representative instances for each class. Despite its advantage, this approach increases computational overhead and operates on the assumption that each class appears only once in a single time step—a premise not applicable to our TIML scenario. Consequently, we adopt the random sampling method.

7.2.2.2 Loss Functions

Before delving into the technical details of our adapted approaches and the proposed new approach, we first formalize the two standard loss functions that are generally used as sub-loss functions for incremental learning.

Cross-entropy loss [190], often used in classification tasks, measures the dissimilarity between the predicted probability distribution P and the ground truth distribution Q. It is especially common in training neural networks for multi-class classification problems. Given:

- $P(y)$ as the predicted probability distribution,
- $Q(y)$ as the ground truth distribution

The cross-entropy loss is defined as:

$$\mathcal{L}_{CE}(P, Q) = -\sum_{y} Q(y) \log(P(y)) \tag{7.1}$$

Kullback-Leibler (KL) Divergence [244] is a measure of how one probability distribution diverges from an expected probability distribution. It is often used in scenarios involving generative models or when comparing two probability distributions. Given:

- $P(y)$ as the true probability distribution,
- $Q(y)$ as the approximate distribution,

The KL Divergence is defined as:

$$\mathcal{L}_{KL}(P||Q) = \sum_{y} P(y) \log\left(\frac{P(y)}{Q(y)}\right) \tag{7.2}$$

7.3 Temporal-Incremental Malware Learning

Temporal-Incremental Malware Learning (TIML) is an incremental learning problem that fits with the practical use of evolving malware classification. We address TIML by dynamically responding to changes in data distributions among known malware families, while also proficiently identifying and categorizing newly emerging ones. To that end, we consider that the historical timeline of the data can be split into different *time steps*. A fundamental stipulation inherited from incremental learning is the limited data accessibility for model updates; specifically, the model training is limited to data from the current time step, supplemented by a restricted subset of samples from previous time steps—known as the exemplar set—when applicable. Figure 7.2 presents an overview of the Temporal-Incremental Malware Learning (TIML) problem formulation, with data variables detailed in the subsequent formal definition.

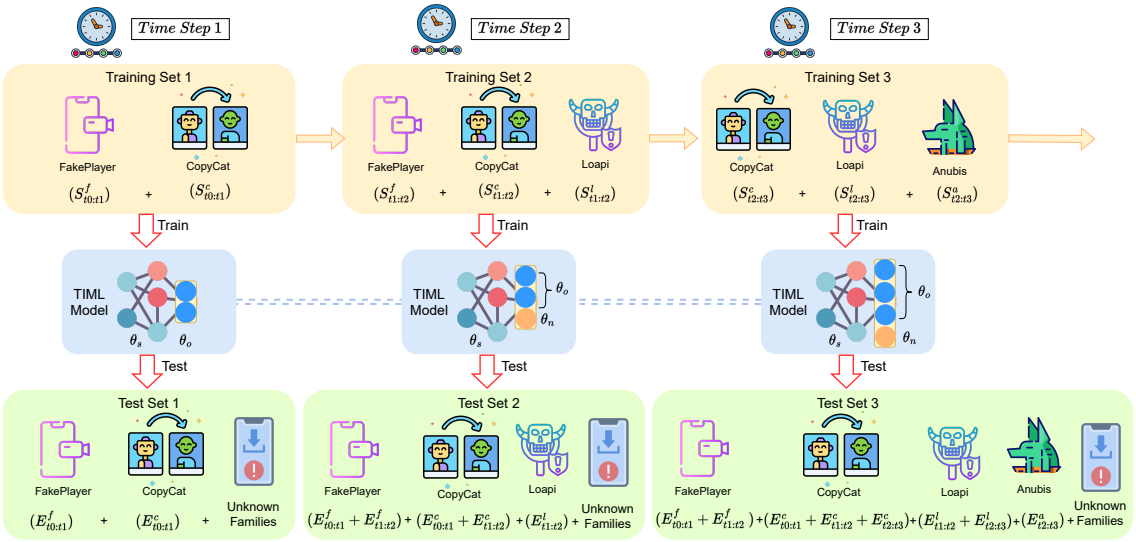

Figure 7.2: Overview of Temporal-Incremental Malware Learning.

To precisely define the TIML problem, let us start by delineating a series of time steps:

$$T = \{T_1 : [t_0, t_1], T_2 : [t_1, t_2], \dots, T_b : [t_{b-1}, t_b], \dots, T_B : [t_{B-1}, t_B]\}. \quad (7.3)$$

Here, each time period T_b represented as a distinct time interval $[t_{b-1}, t_b]$, corresponds to a unique duration for data accumulation. Training executed on the data from each time step t_b is perceived as a distinct task. Given a dataset S, this results in a sequence of tasks $\left\{D^1, D^2, \dots, D^b, \dots, D^B\right\}$, where the subset $S_{t_{b-1}:t_b}$ corresponding to task D^b encompasses n_b training instances, expressed as $\{(x_i^b, y_i^b)\}_{i=1}^{n_b}$. Each instance x_i^b belongs to malware family y_i which is situated within the label space Y_b for that task. Given that malware families typically have a life cycle, they may appear across multiple time steps. Consequently, consecutive label spaces from S, namely $\{Y_1, ..., Y_b\}$, could have overlapping malware families. We give the definition of TIML at a specific time step as follows:

Definition 1. *Given a specific time step T_b (relating to the period $[t_{b-1}, t_b]$ within T), the following components and specifications apply:*

1. *The training dataset during T_b, denoted as $S_{t_{b-1}:t_b}$ corresponding to task D^b, is comprised of:*
   - *Samples originating from the newly introduced malware families: $N_{t_{b-1}:t_b}$.*
   - *New samples from previously seen families: $O_{t_{b-1}:t_b}$.*
2. *The dataset $S_{t_{b-1}:t_b}$ encompasses instances $\{(x_i^b, y_i^b)\}_{i=1}^{n_b}$, where x_i^b indicates the malware feature vector and the corresponding label y_i^b belongs to the label space Y_b, inclusive of both old and new malware families introduced only in current time step.*
3. *For the instances contained in $N_{t_{b-1}:t_b}$, there are no overlapping malware families with the previous sequences $\{N_{t_0:t_1}, \dots, N_{t_{b-2}:t_{b-1}}\}$.*
4. *Samples located in $O_{t_{b-1}:t_b}$, and originating from the same unique malware family, are expected to conform to a distribution $F_{t_{b-1}:t_b}(x, y)$. Any shift in data distribution in current time step's samples is observed if:*

$$F_{t_{b-l}:t_{b-l+1}}(x, y) \neq F_{t_{b-1}:t_b}(x, y), \text{where } 2 \leqslant l \leqslant b. \quad (7.4)$$

5. *For evaluation purposes during the T_b time step, the testing data includes samples from all tasks up to the present: $D^{t_0:t_b} : \{D^1, \dots, D^b\}$. This incorporates instances from the malware families in current time step as well as all previously observed ones, i.e., $\{E_{t_0:t_1}, \dots, E_{t_{b-1}:t_b}\}$. The overall label space is $\mathcal{Y}_b = Y_1 \cup \dots Y_b$.*

Consequently, the TIML problem requires a fitted model $f(x) : X \to \mathcal{Y}_b$, which minimizes the expected risk:

$$f^* = \arg\min_{f \in \mathcal{H}} \mathbb{E}_{(x,y) \sim D_t^1 \cup \dots \ D_t^b} \mathbb{I}(y \neq f(x)) \quad (7.5)$$

where $\mathcal{H}$ is the hypothesis space, $\mathbb{I}(\cdot)$ is the indicator function which outputs 1 if the expression holds and 0 otherwise. D_t^b denotes the data distribution of task b. A good TIML model satisfying Equation 7.5 is able to discriminate all malware families, which does not only work well on new samples with two types of concept drift but also preserves the knowledge of former samples.

Essentially, Temporal-Incremental Malware Learning (TIML) leverages training samples exclusively from the current time step, facilitating several key functionalities: It efficiently recognizes and classifies newly emerging malware families; it continuously adapts to distribution shifts within new samples of previously seen malware families; and it effectively preserves the knowledge from old samples of former families that are not impacted by distribution shifts. This paradigm requires that the model remains both current and robust, capable of handling the dynamic nature of malware threats without the need for comprehensive retraining on historical data. Nonetheless, in certain scenarios where more robust knowledge retention is necessary, this constraint is eased, allowing the model to maintain an exemplar set for enhanced adaptability and learning continuity.

7.4 TIML Methodology

In this section, we first provide our methodology of the data organization in Section 7.4.1. Then, we introduce the adapted TIML approaches in Section 7.4.2. In Section 7.4.3, we present our proposed multimodal TIML approach.

7.4.1 Data Organization

In this section, we describe how we organize the data and find appropriate time intervals, aiming to respect the reality of malware families' emergence. We start by identifying the dates of the sample apps from Androzoo [245] and arranging them chronologically. We then proceed to systematically organize the collected samples for distinct tasks towards incrementally training the model. The general rule we use involves splitting the samples over a period of `timeWindow` months, starting from the earliest date associated with our data samples.

Before moving to identify the samples for the next task, it is crucial to ensure that a minimum number of malware families are available to train the model effectively for a given task. To this end, we use the threshold `minNbrFamiliesInTask`, which dictates the minimum number of families required in a task before proceeding to the next. If the number of families identified within a given time window is less than `minNbrFamiliesInTask`, we incrementally expand the time window by one month until this requirement is met. It's important to note that the end date of a task becomes the start date for the next. In addition, each malware family must meet a minimum sample threshold, denoted as `minNbrSamples`, to be included in the training set. A family with fewer than `minNbrSamples` is not considered during training and its samples are instead included in the testing set, labeled as "unknown". A family is counted towards `minNbrFamiliesInTask` only if it satisfies the `minNbrSamples` criterion.

These thresholds are essential for determining the appropriate time intervals for training tasks, given the irregular timing patterns of malware emergence. They ensure that we collect a sufficient number of samples and families within each interval, which is vital for optimizing training resources and maintaining the robustness of the TIML model. Retraining the model too frequently with only a few new malware samples would be inefficient and ineffective. This strategic approach to data collection and training ensures that our model is both efficient and effective, capable of learning robustly from sufficiently populated data sets without the inefficiencies associated with sparse data. We summarize our data organization method in Algorithm1.

Algorithm 1: Algorithm describing the dataset organization

```
Input:  dictAppDate, dictAppFamily, timeWindow, minNbrFamiliesInTask, minNbrSamples
Output:  Subsets of malware apps organized according to TIML
minDate ← dictAppDate.values().min()
maxDate ← dictAppDate.values().max()
tasks ← ∅
currentMinDate ← minDate
Function constructTask (currentMinDate, currentMaxDate):
    task ← getAppsBetweenDates(dictAppDate, currentMinDate, currentMaxDate)
    task ← getFamiliesOfAppsInTask(task, dictAppFamily)
    familiesInTask ← getUniqueFamiliesInTask(task, minNbrSamples)
    return task, familiesInTask
while (currentMinDate ≤ maxDate) do
    currentMaxDate ← currentMinDate+timeWindow
    task, familiesInTask = constructTask(currentMinDate, currentMaxDate)
    while FamiliesInTask.length() ≤ minNbrFamiliesInTask do
        currentMaxDate ← currentMaxDate + 1
        task, familiesInTask = constructTask(currentMinDate, currentMaxDate)
    end
    currentMinDate ← currentMaxDate
    tasks.append(task)
end
```

7.4.2 TIML Approaches

In this section, we provide a formal definition of the baseline approaches considered in this study. Specifically, our four baselines are adaptations of three class-incremental learning techniques: LwF [151], iCaRL [145], and SS-IL [147]. Due to space constraints, we focus primarily on describing the adapted approaches, with a particular emphasis on the modifications to the loss functions.

While our manuscript centers on these loss function adaptations, it's crucial to note that our actual adaptations extend beyond this single aspect. These adaptations are pivotal for aligning the class-incremental learning techniques with the unique requirements of Temporal-Incremental Malware Learning (TIML), effectively addressing differences in data composition and optimizing the knowledge distillation process inherent to TIML. The choice to focus on loss functions was strategic; these modifications encapsulate the core changes made to the baseline approaches and are most indicative of the methodological shifts required for successful application within our TIML framework. For comprehensive details on the original methods and a broader context of the adaptations, we direct readers to the respective foundational papers.

7.4.2.1 Adaptations for TIML

We now elaborate on the adaptations brought to CIL approaches in order to fit with the TIML problem. Given a model with shared parameters θ_s and task-specific parameters θ_o (as shown in Figure 7.2), our goal is to ① add task-specific parameters θ_n for newly introduced malware families and ② learn parameters adapted to old and new tasks, using samples and labels only from the new task (and exemplars, if applicable). The general algorithm of TIML and the notations we will use in the following are outlined in Algorithm 2. Particularly, the training data X_n, Y_n from the new task consists of two sources: ① seen malware families (i.e., samples belonging to malware families used to train the model in previous time steps) $X_n^{(seen)}, Y_n^{(seen)}$, and ② unseen malware families (i.e., samples from newly emerging malware families) $X_n^{(unseen)}, Y_n^{(unseen)}$. Following the experimental setting in MalNet [28], we employ ResNet-18 [225] as the backbone of each classifier. Note that the malware images in MalNet dataset are obtained by converting the bytecode of Android apps into

images, with each byte in the bytecode mapped to a pixel value in the image.

Algorithm 2: The general TIML algorithm

```
Start with:
    θ_s:  shared parameters
    θ_o:  task specific parameters for malware families introduced in old tasks
    X_n, Y_n:  training data and ground truth from the new task
    X_e, Y_e:  exemplar data and ground truth from the old tasks (optional)
Initialize:
    Y_o ← Model(X_n, θ_s, θ_o)      // compute old task output of old model for new data
    θ_n ← RandInit(|θ_n|)      // randomly initialize new parameters
Train:
    Define Ŷ_o ≡ Model(X_n, θ̂_s, θ̂_o)   // old task output of new model for new data
    Define Ŷ_n ≡ Model(X_n, θ̂_s, θ̂_n)   // new task output of new model for new data
    Define Ŷ_e ≡ Model(X_e, θ̂_s, θ̂_o)   // old task output of new model for exemplars (optional)
    θ*_s, θ*_o, θ*_n, ← arg min_{θ̂_s,θ̂_o,θ̂_n} L_TIML(Y_o, Y_n, Y_e; Ŷ_o, Ŷ_n, Ŷ_e)   // depends on the specific method
```

The loss function for each baseline approach is defined as follows:

Fine-tuning: Serving as the lower bound among the baseline approaches, Fine-tuning solely utilizes new data from the current time step and employs a straightforward loss function, namely the cross-entropy loss. Contrary to the standard class-incremental learning scenario, we adapt Fine-tuning in the context of TIML to take into account the newly introduced samples belonging to previously seen malware families:

$$\mathcal{L}_{CE}(Y_n, \hat{Y}_n) = \mathcal{L}_{CE}(Y_n^{(seen)}, \hat{Y}_n^{(seen)}) + \mathcal{L}_{CE}(Y_n^{(unseen)}, \hat{Y}_n^{(unseen)}) \tag{7.6}$$

Full Retraining: Serving as the upper bound, Full Retraining employs the same loss function as Fine-tuning. The key distinction is that the upper bound utilizes, in addition to the new data, all the historical data for training in each time step.

LwF: In contrast to Fine-tuning, LwF employs a task-specific knowledge distillation technique to retain knowledge from prior data. Specifically, it incorporates the KL divergence as an additional loss term to ensure the model's output distribution remains consistent with that of previous time steps:

$$\mathcal{L}_{LwF} = \mathcal{L}_{CE}(Y_n, \hat{Y}_n) + \lambda_o \sum_{t=1}^{b-1} \mathcal{L}_{KL}(\hat{Y}_o^{(D_t)} || Y_o^{(D_t)}) \tag{7.7}$$

where b is the current time step and D_b is its corresponding task, $\hat{Y}_o^{(D_t)}$ corresponds to the output on families that were firstly introduced in time step t, and λ_o is a loss balance weight.

LwF with Exemplars: The original LwF methodology [151] does not incorporate exemplar data. However, in our experiments, we observed that integrating a small amount of exemplar samples can substantially enhance its performance. The corresponding loss function is defined as:

$$\mathcal{L}_{LwF-Exemp} = \mathcal{L}_{CE}(Y_n, \hat{Y}_n) + \mathcal{L}_{CE}(Y_e, \hat{Y}_e) + \lambda_o \sum_{t=1}^{b-1} \mathcal{L}_{KL}(\hat{Y}_o^{(D_t)} || Y_o^{(D_t)}) \tag{7.8}$$

iCaRL: The original iCaRL [145] introduces a herding-based method for selecting highly representative exemplars and employs a *nearest-mean-of-exemplar* rule for classification, both requiring stable data distributions across all the time steps for each family. Considering the potential shifts in malware family distributions, these

techniques are inapplicable to TIML. Consequently, we opt for a random exemplar selection strategy and a conventional softmax-based classification approach. For knowledge distillation, we retain its overall KL loss term:

$$\mathcal{L}_{iCaRL} = \mathcal{L}_{CE}(Y_n, \hat{Y}_n) + \mathcal{L}_{CE}(Y_e, \hat{Y}_e) + \lambda_o \mathcal{L}_{KL}(\hat{Y}_o || Y_o) \tag{7.9}$$

SS-IL: This approach is designed to tackle a primary cause of catastrophic forgetting: the classification score bias introduced by the data imbalance between unseen classes and seen classes (stored in the exemplar-memory). SS-IL [147] introduces a separated softmax output layer to compute the cross-entropy for both seen and unseen classes independently, coupled with a task-specific knowledge distillation similar to LwF. Under the TIML definition, the seen malware families encompass not only those from the exemplars but also the new samples from the current time step that belong to previously seen malware families. The resultant loss function is formulated as:

$$\begin{aligned}\mathcal{L}_{SS-IL} =& \mathcal{L}_{CE}((Y_n^{(seen)})^{(D_1:D_{b-1})}, (\hat{Y}_n^{(seen)})^{(D_1:D_{b-1})}) + \mathcal{L}_{CE}((Y_n^{(unseen)})^{(D_b)}, (\hat{Y}_n^{(unseen)})^{(D_b)}) \\ &+ \mathcal{L}_{CE}(Y_e^{(D_1:D_{b-1})}, \hat{Y}_e^{(D_1:D_{b-1})}) + \lambda_o \sum_{t=1}^{b-1} \mathcal{L}_{KL}(\hat{Y}_o^{(D_t)} || Y_o^{(D_t)})\end{aligned} \tag{7.10}$$

7.4.3 Multimodal TIML

In our study, we focus on two primary malware features: bytecode images and graph centrality vectors. While each offers unique insights, previously adapted TIML methods were limited to using only one type of feature. However, we observed that combining these features could significantly enhance our understanding and performance in malware classification within the TIML framework. Consequently, we introduce a multimodal TIML approach that integrates both features, aiming for superior performance by leveraging the complementary strengths of bytecode images and graph centrality vectors.

The core motivation for adopting a multi-modal approach stems from the need to enhance the performance of TIML by enabling a more comprehensive understanding of malware behaviors. This is especially crucial in TIML settings where the available training samples are significantly fewer than in full retraining scenarios. Maximizing the informational yield from each sample is essential to maintain robust model performance despite reduced training volumes. Our choice to combine bytecode images and function call graphs is driven by their complementary nature in representing malware characteristics:

- **Bytecode Images**: These provide a visual representation of the binary structure of the malware, offering insights into the compiled code without executing the malware. This modality helps in capturing structural patterns that are indicative of malicious intent or behavior.
- **Function Call Graphs**: These graphs map out the relationships and interactions between different functions within the malware code, offering a detailed view of the execution flow and potential points of malicious activity.

By combining these two modalities, our model benefits from a holistic view of each malware sample, leveraging structural insights from bytecode images and behavioral insights from function call graphs. This comprehensive input is designed to mitigate the impact of having fewer samples by enriching the information the model can learn from each individual sample.

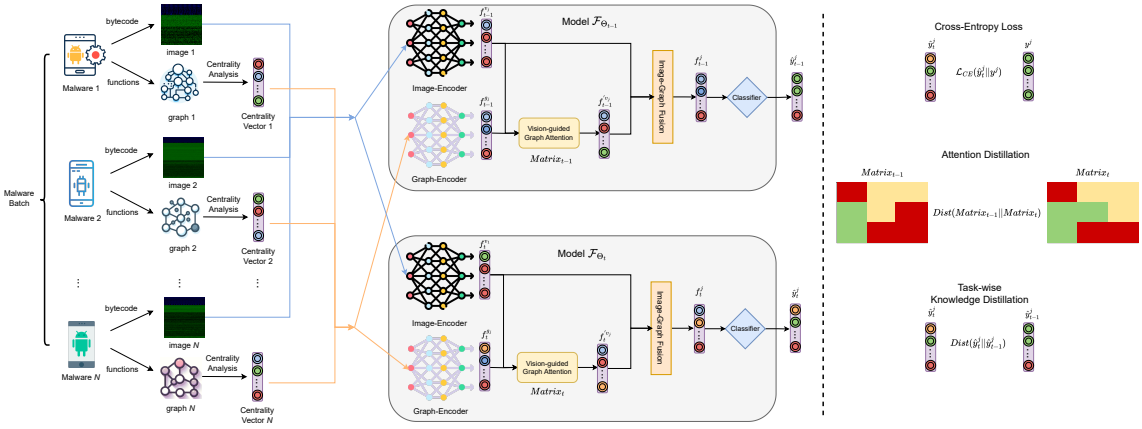

Figure 7.3: Illustration of our proposed multimodal TIML approach.

7.4.3.1 Overview

As depicted in Figure 7.3, our multimodal TIML approach begins by extracting two primary features from each APK: the bytecode image and the centrality vector from the function call graph. Following the methodology outlined by MalNet [28], bytecode is extracted from the DEX files within the APK and converted into an image. The conversion is performed by mapping each byte in the DEX files to a corresponding pixel value in the image. Simultaneously, we utilize Malscan [235] to derive the centrality vector, which captures the critical interaction points within the function call graph.

Each of these features undergoes a separate but specialized processing pathway using custom feature extractors designed to produce high-level embeddings. The centrality vectors, comprising various base features, are particularly refined through a novel attention mechanism. This mechanism leverages the image embeddings as a guide, enhancing the focus on essential features within the vector that are most indicative of malware characteristics. This process is crucial as it allows for the centrality vector to be dynamically adjusted based on the insights provided by the visual patterns observed in the bytecode images.

The final stage of our approach involves the integration of these processed embeddings. As illustrated in the central portion of Figure 7.3, embeddings from both the bytecode image and the graph centrality vector are fused together. This fusion occurs in a fully connected layer designed to consolidate and capitalize on the complementary strengths of both modalities, enabling a more robust classification of the malware family. The integrated embeddings are then used to classify the APK into one of the known malware families.

7.4.3.2 Model Architecture

In the adapted TIML model architecture, ResNet-18 [225] is employed as the primary feature extractor, originally designed for real-world image classification within CIL methodologies. Recognizing the distinct characteristics of malware data, specifically bytecode images and graph centrality vectors, which lack the spatial and semantic depth of natural images, we opt for a more tailored, lightweight backbone in our multimodal TIML approach. This customized approach includes three 2D convolutional layers (kernel size=5, padding=2) for bytecode images, and three 1D convolutional layers (kernel size=7, padding=1) for graph centrality vectors, each

followed by a linear layer to manage embedding dimensions.

The attention module plays a pivotal role, in processing the embeddings from both modalities. It adjusts the image embeddings for compatibility and computes attention weights by assessing the importance of different parts of vector embeddings relative to the overall guidance (i.e., image embedding). These weights are crucial for focusing the model's attention on the most pertinent data features. Subsequently, the module applies these weights to the vector embedding, yielding an "attended" vector that highlights information vital for malware classification, alongside the attention weights for further attention distillation in the training phase. Finally, we fuse the attended vector embedding with the image embedding through element-wise addition, directing the combined output to a fully connected layer for the ultimate task of malware family classification. This architecture leverages the distinct strengths of each data modality while ensuring a focused, efficient approach to malware identification.

7.4.3.3 Loss Function

The loss function for our multimodal TIML model is composed of three segments: cross-entropy for learning from new malware instances, task-specific KL divergence for distilling knowledge from prediction logits, and standard KL divergence to maintain attention-weight consistency. This comprehensive loss function ensures balanced learning and knowledge retention across updates and is defined as:

$$\begin{aligned}\mathcal{L}_{MM-TIML} = \mathcal{L}_{CE}(Y_n, \hat{Y}_n) + \mathcal{L}_{CE}(Y_e, \hat{Y}_e) + \lambda_o \sum_{t=1}^{b-1} \mathcal{L}_{KL}(\hat{Y}_o^{(D_t)} || Y_o^{(D_t)}) \\ + \lambda_1 \mathcal{L}_{KL}(\hat{W}^{(D_t)} || W^{(D_t)})\end{aligned} \tag{7.11}$$

where $\hat{W}^{(D_t)}$ is the attention weights of old model on new data, i.e., $Matrix_{i-1}$ in Figure 7.3, and $W^{(D_t)}$ is the attention weights of new model on new data, i.e., $Matrix_i$ in Figure 7.3.

7.5 Study Design

In this section, we first overview the research questions. Then, we present the details about the dataset and empirical setup. Last, we define the metrics we use for evaluation.

7.5.1 Research Questions

In this study, we aim to investigate the following research questions:

RQ1 *Is concept drift a significant factor affecting malware classification?*

RQ2 *How well do TIML approaches perform in malware classification?*

RQ3 *How resilient are TIML approaches to catastrophic forgetting?*

RQ4 *How effectively do TIML approaches optimize resource utilization?*

7.5.2 Dataset Description

We conducted our experiments on the MalNet dataset [28], which is uniquely suitable for our study's objectives. MalNet contains over 1.2 million malware images, originally sourced from AndroZoo [245], with each app's bytecode converted into

colored images of size (256, 256). This dataset encompasses 696 malware families, labeled based on analyses from VirusTotal [246] and Euphony [247]. Euphony takes ViruTotal report containing up to 70 labels across a variety of antivirus vendors and unifies the labeling process by learning the patterns, structure and lexicon of vendors over time. Importantly, MalNet covers a substantial time span of 10 years, providing essential temporal data that is critical for analyzing the evolution of malware threats.

The choice to exclusively utilize MalNet was driven by its comprehensive scale and depth, containing significantly more malware samples and families compared to other available datasets. For instance, when compared to Virus-MNIST, the next largest public dataset, MalNet offers approximately 24 times more malware samples and nearly 70 times more families. Furthermore, unlike Virus-MNIST, MalNet includes vital temporal information, such as the emergence dates of malware instances, making it the only dataset we identified that meets all critical requirements for Temporal-Incremental Malware Learning (TIML): large scale, extended duration, and detailed temporal information. These features make MalNet indispensable for studying the TIML problem effectively.

In our quest for a more comprehensive performance evaluation, we have also relied on MalScan [235] features that we extracted from the same apps of the MalNet dataset. MalScan considers the function call graph of Android apps as a social network to conduct a centrality analysis on the graph. We rely on *concatenate* feature type of MalScan as it includes the concatenation of its four base features: degree, katz, closeness, harmonic. In our novel multi-modal TIML approach, we utilize concatenated long vectors as input for MalScan. However, as the four adapted TIML approaches were originally designed to only take images as input, we follow on the previous work [248] that converted MalScan feature vectors to images, so that we can leverage the same model architecture for MalNet images. Since MalScan feature vectors have a size of 87 944, we pad them with zeros and convert them to images of size (176, 176).

Since some approaches make use of exemplars to not forget the knowledge learned from previous tasks, we rely on the *Fixed Total Exemplars* technique as we explained in Section 7.2. Specifically, we set M to 10 000, which means that the total number of exemplars we keep from previous tasks cannot exceed 10 000.

7.5.3 Empirical Settings

All of our experiments are conducted on MalNet and MalScan features, and our experimental configuration is derived from thorough and comprehensive experimentation. We set our dataset organization thresholds, i.e., `timeWindow`, `minNbrFamiliesInTask`, and `minNbrSamples`, to 4, 4, and 20 respectively, based on the constraints described in Section 7.4.1. Note that these parameters are the inputs of Algorithm 1, meaning that a time window can be 4 months or more, depending on the output of Algorithm 1 (which, as a reminder, can incrementally extend a time window if there are not enough malware samples and families - cf. Section 7.4.1). Moreover, these thresholds ensure that there are a minimum of 4 families per task and that each family has at least 20 samples.

After identifying the malware families in a given task, we systematically divide the samples of each family into training (80%) and test (20%). In the preliminary study phase, 70% of the training portion is used for actual training, while the remaining 10% is allocated as the validation set. This is done to fine-tune the hyperparameters and

prevent overfitting. In the second phase, where we perform the final evaluation on the full dataset, we use the entire 80% training portion for training without reserving any validation set, as the hyperparameters have already been optimized. This structured division ensures robust training during the preliminary phase and maximizes the use of data during the final evaluation phase. Moreover, a portion (20%) is always reserved as a test set. This strategy enables the evaluation of TIML's performance on unseen samples, allowing us to assess the model's effectiveness in recognizing new, previously unseen malware instances within the same time frame. The randomness in our evaluation arises from the use of the random seeds. To mitigate potential biases and ensure robustness, we conduct each experiment five times with different seeds and report the average results. We provide statistical analysis in the appendix. This method is in line with general practices in the literature and helps to stabilize the evaluation metrics.

It is imperative to note that families comprising fewer than 20 samples within a specific task will be uniformly integrated into the test set and assigned the categorical label "unknown". Since these families are not included in the training, the malware classifier does not learn their characteristics. Consequently, we consider two evaluation scenarios: (1)- We test the malware classifier only on the families that have been learned during the training, which implies removing the "unknown" families in this case. We refer to this scenario as *Unknown families excluded*. (2)- We consider the "unknown" families as new emerging malware families since only a few of their samples are available. In this case, they are not well-established families and are treated as the other samples in the test. Assuming that the labels of all the samples in the test set are known, the malware classifier is expected to incorrectly classify these unknown samples into a known malware family. We refer to this scenario as *Unknown families included*.

Each adapted TIML approach employs ResNet-18 [225] as the foundational network architecture for the classifier, whereas the proposed multimodal TIML approach incorporates the newly designed simple backbone outlined in Section 7.4.3.2. We train the models using a *learning rate* of `1e-4` and a *number of epochs* equals to `100`. For the case of *SS-IL*, we trained it for `150` epochs as it used a lower *learning rate* of `1e-5` in the original setting. We set the *batch size* to `128` and the *weight decay* to `1e-4`. All the experiments were conducted on a single Tesla V-100 GPU with 32GB of memory on an NVIDIA DGX Station. For more details about our implementation, please refer to our replication package.

7.5.4 Evaluation Metrics

An approach that addresses the TIML problem is effective when it ensures that the amount of resources needed to achieve standard performance is limited. The goal of our evaluation is therefore to assess the extent to which TIML approaches can reduce resource utilization (i.e., training time and data storage) while achieving high performance (i.e., accuracy and forgetting score).

The Average Training Time: It measures the average duration required to train the model for each time step. It is defined as follows:

$$Avg.T.Time = \frac{1}{T}\sum_{t=1}^{T} d_t, \tag{7.12}$$

where d_t indicates the duration of training at the current time step.

The Maximum Data Storage: To assess the data storage, we consider the Maximum Data Storage across all time steps, taking into account that storage space can be reused across different time steps. We formalize it as follows:

$$Max.D.Storage = \max_{t \in \{1,\ldots,T\}} (s_t + e_t), \tag{7.13}$$

where s_t represents the storage space of training data and e_t corresponds to the storage space of exemplar data at each time step.

We conduct the performance evaluation of the various approaches using Mean Accuracy and Average Forgetting metrics that are widely recognized in incremental learning research.

Mean Accuracy: It represents the cumulative average testing accuracy across all malware families encountered up to the current time step, and is mathematically defined as follows:

$$MeanAcc. = \frac{1}{T} \sum_{t=1}^{T} a_t, \tag{7.14}$$

where a_t denotes testing accuracy of all seen malware families until the current time step t.

Average Forgetting: It serves as a metric to quantify the degree of catastrophic forgetting of previously acquired knowledge over time. Catastrophic forgetting is a phenomenon where a model, much like humans, loses previously acquired knowledge. This occurs when the model is trained on new data and consequently forgets what it had learned from older data. This is a crucial issue in incremental learning, as it limits the model's ability to adapt to new information without losing existing knowledge. Addressing catastrophic forgetting is essential for developing models that can effectively learn and evolve over time in dynamic environments. The average forgetting score is formalized as:

$$Avg.Forget. = \frac{1}{T-1} \sum_{t=2}^{T} \left(\frac{1}{t-1} \sum_{i=1}^{t-1} \max_{\tau \in \{1,\ldots,t-1\}} (a_{\tau,i} - a_{t,i}) \right), \tag{7.15}$$

where $a_{\tau,i}$ is the testing accuracy on evaluation data of τ-th time step.

7.6 Experimental Results

In this section, we present the evaluation results of the different approaches based on two malware features, i.e., bytecode image and graph vector. Explicitly, we name the features the same as the name of approaches, i.e., MalNet [28] and MalScan [235]. This section also answers our research questions.

7.6.1 RQ1 Is concept drift a significant factor affecting malware classification?

Incremental learning starts gaining attention and significance in malware classification due to the high cost of model retraining which is required to tackle the concept drift of malware over time. Concept drift can affect the performance of a model that was trained on older data, making it less effective at identifying new or evolved malware samples. Therefore, with the first research question, we seek to rigorously investigate the presence and impact of concept drift using the large-scale MalNet

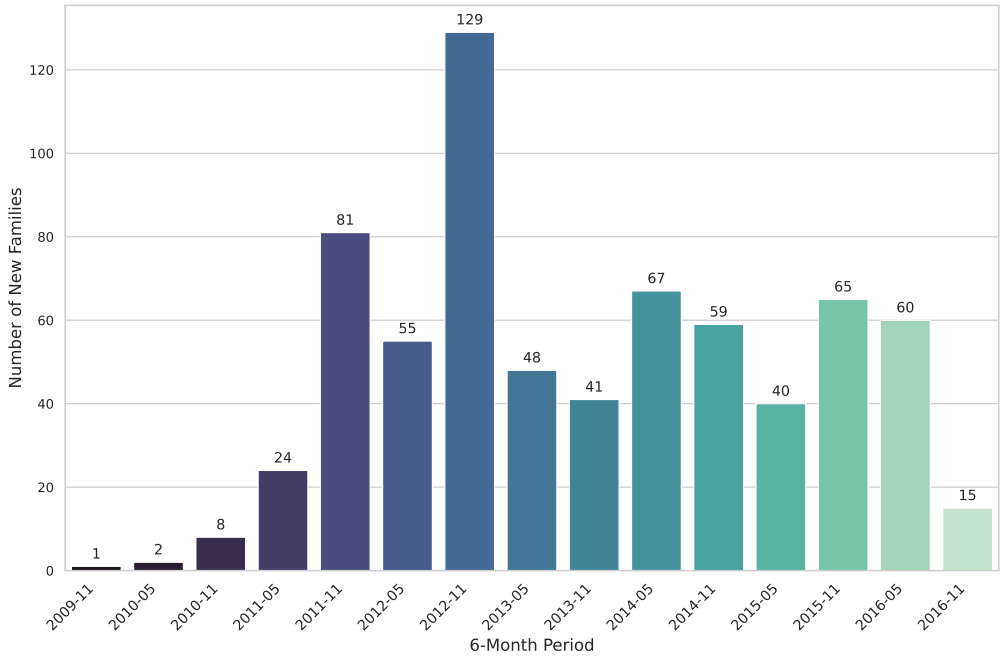

Figure 7.4: Distribution of new malware families – per 6-months time steps

dataset containing 1.2 million real-world malware. This analysis will support our primary motivation for the exploration of Temporal-Incremental Malware Learning.

We identified two primary forms of concept drift over time: ① the emergence of new malware families, and ② shifts in the distribution of old families. Specifically, we first obtained the emergence date from AndroZoo [245] for each malware instance. We then identified the date of the earliest malware sample within each family as the emergence date for that family. Using this method, we calculated the number of new malware families that emerged each half year. As depicted in Figure 7.4, new malware families have been consistently emerging each year since 2009, which follows the inaugural year of Android. A noticeable increase in new families occurred between 2011 and 2012, which is consistent with the peak popularity of the Android platform. The emergence of new families forces the updating of existing classification models to maintain their performance. Concurrently, our analysis indicates that individual malware families exhibit longevity, persisting for several years. Figure 7.5 illustrates the lifespan of the top 20 malware families, based on the number of samples, providing valuable insights into their respective life cycles. This phenomenon of enduring presence is observable across the vast majority of other malware families as well, raising an important question: Do the distributions of malware samples within existing families change over time? In other words, does a classifier's performance degrade when encountering new instances from families it was previously trained to recognize?

We conduct an experiment to examine the distributional shifts within malware families over time. To this end, we train a model on samples from eight malware families that appeared before 2012 and subsequently test it on samples from the same families but emerging between 2012 and 2017. This experimental setup allows us to directly observe the changes in malware characteristics over time. As depicted in Figure 7.6, we notice a consistent decline in mean accuracy when applying the

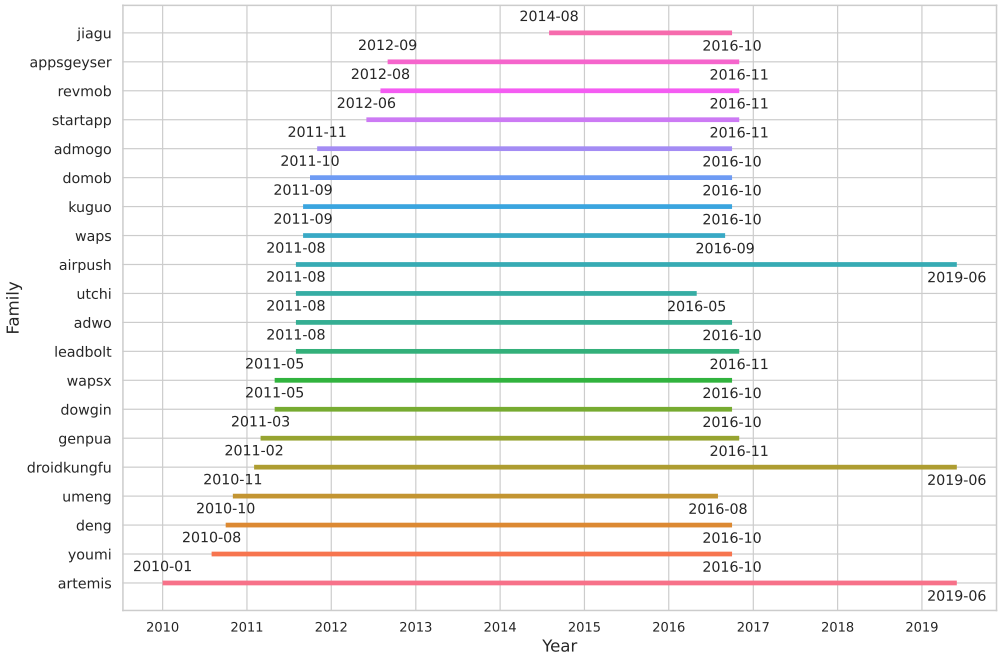

Figure 7.5: Life Span of Top 20 Malware Families

model to newer samples, using two distinct input features—MalNet and MalScan. This decline demonstrates what we refer to as "temporal evolution" within malware families. Specifically, the term describes how malware samples, even from the same families, exhibit shifts in their feature distributions over time, rendering models trained on earlier data less effective at recognizing newer variants. This phenomenon not only confirms the existence of temporal evolution but also underscores the necessity for our research into Temporal Incremental Malware Learning (TIML). Our TIML approaches are specifically designed to adapt to these temporal shifts, enabling the malware classification model to incrementally update itself and thereby reduce retraining costs while maintaining high levels of accuracy. Further elaboration on our methods and their implications in addressing these challenges is detailed in the answers to subsequent research questions.

RQ1 Answer: Concept drift negatively impacts the predictive capabilities of malware classification models, confirming that it is a significant factor in malware family classification.

7.6.2 RQ2 How well do TIML approaches perform in malware classification?

In this section, we first introduce a preliminary study we undertook to validate our hypothesis that conventional CIL methods fall short within the TIML framework. This investigation underpins the imperative for developing tailored TIML strategies. Subsequently, we evaluate the performance of the four adapted TIML methods and our newly developed multimodal TIML approach. These are benchmarked against three baseline strategies: Random Prediction, Fine-tuning, and Full Retraining, to ascertain their relative efficacy.

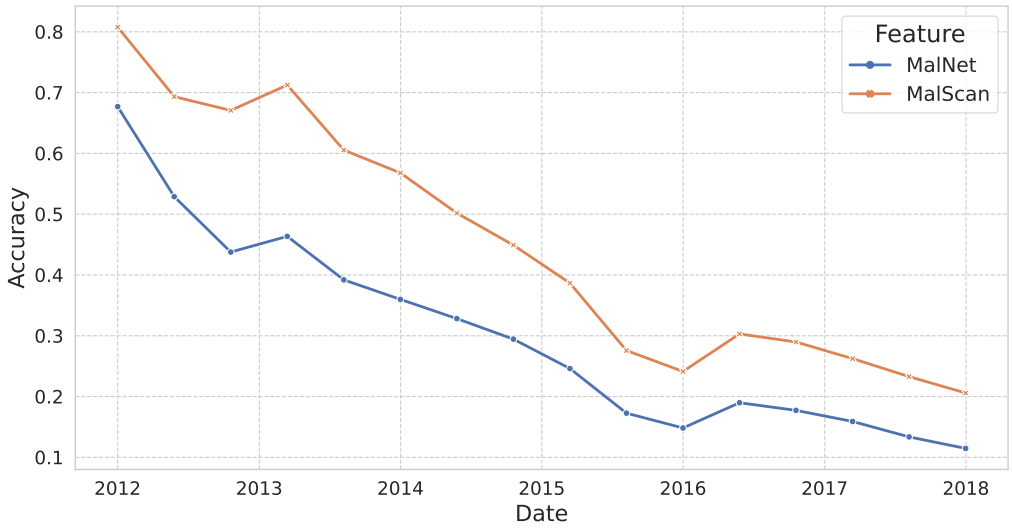

Figure 7.6: Performance drop curve of models trained on pre-2012 malware families and evaluated on post-2012 samples from the same families.

7.6.2.1 Preliminary Study

Table 7.1: Comparison of Accuracy between Adapted TIML Approaches and Their Original CIL Versions

Method	Adapted TIML Accuracy	CIL Accuracy
LwF	49.68%	27.99%
iCaRL	58.15%	23.13%
SS-IL	54.58%	21.65%

In our preliminary investigation, we utilized the official small version of the MalNet dataset, which comprises 87 430 malware samples. This exploratory phase was crucial for assessing the viability of CIL methods within the TIML framework. Our findings are significant, highlighting marked improvements in the performance of various TIML adaptations compared to their CIL counterparts. Specifically, as shown in Table 7.1, the adapted TIML version of LwF demonstrated an accuracy of 49.68%, markedly superior to its CIL version of 27.99%. Similarly, the adapted TIML version of iCaRL achieved an accuracy of 58.15%, a substantial improvement over its CIL version of 23.13%. Lastly, the adapted TIML version of the SS-IL method showed an accuracy of 54.58%, significantly outperforming its CIL version of 21.65%. These results underscore the potential benefits and necessity of developing specialized TIML approaches, as traditional CIL methods may not fully leverage the dynamics and challenges inherent in the TIML paradigm.

7.6.2.2 Performance Analysis of TIML Approaches

We first include three baseline TIML approaches. The Random Prediction model serves as our foundational baseline, where a family label is attributed to a malware sample based purely on randomness. Any approach surpassing the Random Prediction model in terms of accuracy indicates its capability to grasp the temporally incremental knowledge of evolving malware. Fine-tuning is a prevalent choice for

Table 7.2: Performance Comparison of Different Approaches based on Two Input Features: MalNet and MalScan.

	Unknown families excluded				*Unknown families included*	
	Mean Accuracy (%)		Average Forgetting		Mean Accuracy (%)	
Approach	MalNet	MalScan	MalNet	MalScan	MalNet	MalScan
Random Prediction	1.72	1.72	-	-	1.28	1.28
Fine-tuning *	51.63	64.66	13.25	18.59	35.46	41.43
LwF	52.82	65.69	12.48	18.09	37.86	45.67
SS-IL	51.73	67.14	8.24	8.73	35.10	42.30
iCaRL	53.57	68.57	8.79	8.57	36.17	44.34
LwF with Exemplars	56.28	69.74	8.07	8.13	39.53	44.95
Full Retraining *	63.23	75.08	-	-	45.35	54.91
MM-TIML	70.53		7.66		45.08	

* Fine-tunning and Full Re-training are referred as Lower and Upper Bound, respectively.

model updates. In this method, only the newly emerged data from the current time step is employed to fine-tune a model originally trained on older data. Given its narrow data scope within the TIML setting and the absence of tailored techniques for the TIML challenges, we consider it as a lower-bound approach. Conversely, the Full Retraining strategy makes use of the entire data seen up to the current time step, establishing itself as an upper-bound approach. The objective of the adapted TIML approaches and our new multimodal approach is to get the performance closer to Full Retraining while primarily relying only on the data from the current time step, similar to the Fine-tuning method, but potentially augmented with a minimal set of exemplar samples from previous data.

Table 7.2 provides an overview of the performance comparison among the various approaches based on two input features (i.e., MalNet and MalScan), in which the Mean Accuracy is an average value across all time steps. Specifically, we report the results for the two scenarios we introduced in Section 7.5.3, which aim to evaluate the performance of the studied approaches in the absence and presence of "unknown" families.

From Table 7.2, we observe that Random Prediction has a very low performance which offers insight into the inherent complexity and difficulty of the TIML problem. TIML approaches strive to bridge the performance gap between Fine-tuning and Full Retraining, aiming to achieve performance levels closer to Full Retraining while utilizing a constrained amount of historical data as the Fine-tuning does. The results in Table 7.2 show that the adapted TIML approaches indeed report a good performance in classifying malware families. As we can see, MalScan's graph vectors demonstrate greater effectiveness than MalNet images, thus our MM-TIML positions the lower bound and upper bound of MalScan as the performance boundaries. Especially, LwF with Exemplars stands out with the highest Mean Accuracy compared to other adapted TIML methods. Notably, our newly proposed multimodal TIML approach (i.e., MM-TIML) significantly outperforms all the single-feature-based methods. This success can be attributed to its innovative integration of diverse data modalities, which provides a more holistic understanding of malware characteristics. This trend is further confirmed by Figure 7.7, which illustrates the

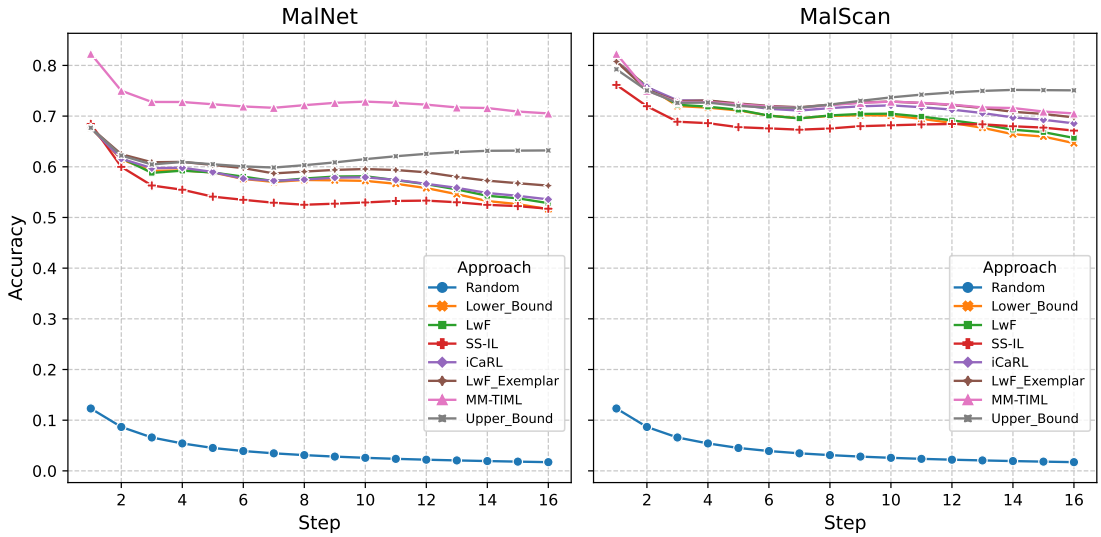

Figure 7.7: Performance curves of various approaches across consecutive time steps.

performance evolution of each approach across individual time steps. In the first time-steps, the performance of these TIML approaches was very close to the Upper Bound. In the last time-steps, we observe that the performance has slightly decreased for the TIML approaches compared to the Upper Bound, which is justifiable given that they do not have access to data from previous time-steps. Specifically, as depicted in Figure 7.7, the SS-IL exhibits subpar performance during the majority of the initial time steps. This is primarily due to the fundamental assumption underlying its original version, which assumes that the training samples of older malware families within each time step are solely derived from exemplars. However, this assumption does not align with the realities of the TIML scenario. Consequently, the adaptation process is unable to fully capitalize on the strengths of its original design.

Finally, we highlight that the performance of Temporal-Incremental Malware Learning (TIML) approaches should be understood within the specific context of incremental learning, which inherently utilizes significantly limited data for retraining the model. Comparing TIML performance to Full Retraining may initially suggest a drop in accuracy; however, such comparisons do not account for the operational constraints inherent to TIML. Full Retraining, representing an ideal upper bound, is often impractical due to data retention constraints such as privacy policies, data security, or storage limitations, especially in real-world settings where retaining comprehensive historical data is not feasible. In contrast, TIML approaches, requiring only current data increments, offer a more viable and often the only feasible option. Our evaluations demonstrate that TIML approaches not only adapt effectively within these constraints but also underscores the importance of this research direction, encouraging further investigation and advancements in this field.

RQ2 Answer: TIML approaches maintain strong performance in classifying malware families, despite a slight decrease compared to full retraining. This performance dip is primarily due to TIML's limited access to historical data, aligning with its design to operate effectively under practical constraints where extensive data retention is not feasible.

7.6.3 RQ3 How resilient are TIML approaches to catastrophic forgetting?

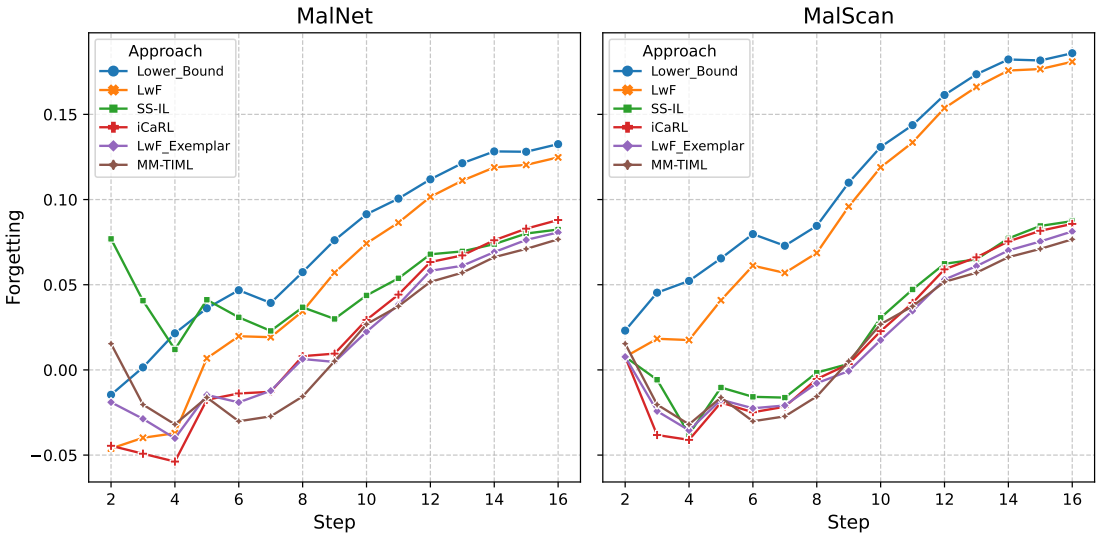

Figure 7.8: Forgetting curves of various approaches across consecutive time steps.

In this section, we explore how the TIML approaches deal with catastrophic forgetting, specifically assessing how they retain or lose knowledge learned from previous instances over time. As outlined in Section 7.5.4, the Average Forgetting score is employed to estimate the magnitude of catastrophic forgetting. The overall average forgetting score for each approach across all time steps can be found in Table 7.2. We also report the detailed forgetting scores across individual time steps in Figure 7.8. For the case when "unknown" families are included, it is meaningless to calculate the forgetting score since its value is zero for the "unknown" families. In instances where the forgetting score is low, it is crucial to note that the model effectively retains knowledge from historical data. Conversely, a high forgetting score indicates the model's difficulty in recalling information from preceding time-steps. The zero forgetting score on "unknown" families is misleading as it suggests that the model does not forget the knowledge from the "unknown" families even though these families were not included in the training. Consequently, the forgetting scores are omitted when the "unknown" families are included.

It is worth mentioning that the Random Prediction method derives its results purely from randomness; without any data training, there is no acquired knowledge to forget. In contrast, the Full Retraining approach, considered as the upper bound, utilizes all historical data in every time step. This comprehensive data usage allows it to continually revisit previously learned knowledge, ensuring no information loss. As a result, measuring its degree of catastrophic forgetting is irrelevant. Therefore, the forgetting scores for these two baselines are omitted. We note that a low forgetting score does not necessarily imply a high Mean Accuracy. "Forgetting" primarily focuses on the performance related to old samples, overlooking the samples newly introduced in the current step, while accuracy evaluates the classifier's performance on all samples accumulated up to the current time step. Furthermore, forgetting score, being a relative metric underscores the difference between old and new accuracy but is unable to offer insights into the absolute accuracy levels.

Overall, the forgetting scores for the studied approaches are continuously increasing. We observe that the four approaches: LwF with Exemplar, iCaRL, SS-IL, and MM-TIML exhibit a decreasing trend in forgetting up to step 4. This is largely attributed to the exemplar set that encompasses nearly all historical data during these steps. We also observe that our proposed MM-TIML depicts the lowest forgetting scores compared to the other baselines. The Lower Bound, as expected, has the highest forgetting score as it does not employ any mechanism to retain previous knowledge.

RQ3 Answer: TIML approaches exhibit signs of forgetting, with our proposed MM-TIML demonstrating the strongest retention of previous knowledge.

7.6.4 RQ4 How effectively do TIML approaches optimize resource utilization?

Having explored the evolution of malware over time and the performance of our TIML approaches in previous sections, we now delve into the resources saved by these methods. This analysis is pivotal for justifying our initial motivation and affirming the added value of these approaches. We focus on two primary metrics: computational efficiency, represented by training time, and maintenance cost, quantified by data storage space. We report their values in Table 7.3 and illustrate the evolution of these two metrics for each approach across time steps in Figure 7.9. Particularly, the storage curves of lower bound and LwF are overlapped as they only use the same newly added data at each step; LwF with Exemplar, iCaRL, and SS-IL are overlapped as they use both the same new data and the same exemplars.

Table 7.3: Resource Cost of Different Approaches based on Two Input Features: MalNet and MalScan. Note: A. Time = Average Training Time, T. Time = Total Training Time, D. Storage = Maximum Data Storage.

	A. Time (H)		T. Time (H)		D. Storage (G)	
Approach	MalNet	MalScan	MalNet	MalScan	MalNet	MalScan
Fine-tuning *	11.80	4.99	188.80	79.84	10.78	7.32
LwF	11.44	5.90	183.04	94.40	10.78	7.32
SS-IL	20.63	11.95	330.08	191.20	13.04	8.09
iCaRL	14.84	7.05	237.44	112.80	13.04	8.09
LwF with Exemplars	12.91	8.85	206.56	141.60	13.04	8.09
Full Re-training *	67.52	27.59	1080.32	441.44	58.33	36.22
MM-TIML	12.63		202.08		21.13	

* Fine-tunning and Full Re-training are referred as Lower and Upper Bound, respectively.

Overall, the training time and the data storage of TIML approaches are significantly lower than the Upper Bound. For instance, in the case of MalNet features, the Upper Bound requires a total training time of 1080.32 (H) (i.e., 45 days) while MM-TIML only needs 202.08 (H) (i.e., 8.4 days) to finish the training process. Additionally, to store the training data, the Upper Bound requires maximum data storage of 58.33 (GB) compared to 21.13 (GB) for MM-TIML. As the MM-TIML incorporates both modalities, it shows a modest result in data storage requirements

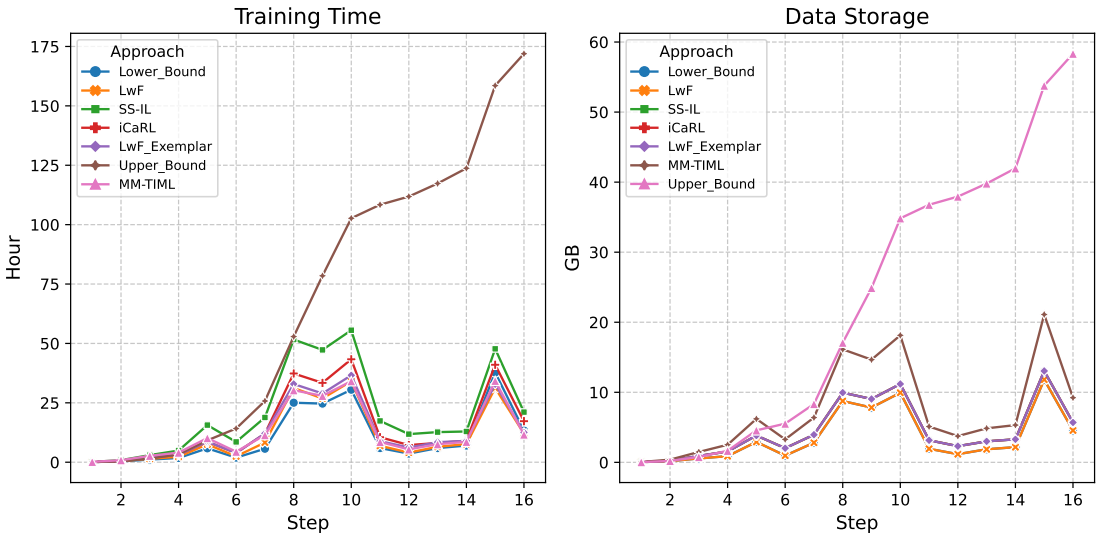

Figure 7.9: Training time and data storage comparison of different approaches, based on MalNet.

compared to the methods based on a single modality. Despite it is not the most efficient way for the storage requirement, it only consumes approximately 36.22% of what Upper Bound needs, which indicates that our strategy balances enhanced malware learning capabilities with efficient use of storage resources. The results in Figure 7.9 also show how fast the training time and storage of the Upper Bound is growing compared to TIML approaches. This pattern suggests that the gap between the training time and storage required for the Upper Bound and TIML approaches will be larger and larger when updating the model in future time-steps.

RQ4 Answer: TIML approaches require significantly lower training times and data storage compared to the Upper Bound, with the observed trend indicating a widening gap over future model updates.

7.7 Summary

With the continuous evolution of Android malware, the development of effective classification models remains paramount. After they are trained, malware classifiers need to be updated regularly to tackle concept drift. However, the updating process is resource-intensive, requiring a significant storage capacity and considerable amounts of time. In this chapter, we introduced Temporal-Incremental Malware Learning (TIML) and provided corresponding approaches to incrementally update malware classifiers which are able to effectively classify emerging malware families while adjusting to the evolving data distribution of old malware families. TIML approaches leverages primarily current data samples to update malware classifiers, which results in reduced resource usage. We adapted CIL approaches to function within the TIML framework, and further proposed a novel multimodal TIML approach to enhance the classification performance. Our results showed that training malware classifiers under TIML yields a promising performance while significantly reducing time and storage requirements.

Furthermore, the applicability of the Temporal-Incremental Malware Learning

paradigm potentially extends beyond the Android platform. The effectiveness of TIML in other contexts relies on the availability of large-scale datasets with chronological data and the development of representation learning techniques specifically tailored to the unique software characteristics of each platform.

Part IV

Conclusion and future work

8

Conclusion

In this chapter, we conclude this dissertation with an overview of our key contributions.

In this dissertation, we set out to advance the field of Android malware learning by addressing several key challenges. Our contributions span a range of topics, from the assessment of Android artifacts to advanced representation learning and malware family classification in temporal-incremental scenarios.

Our first contribution involved a comprehensive assessment of the effectiveness of various Android app artifacts in malware detection. We evaluated Dalvik bytecode, XML (Manifest files), and native code (shared objects), demonstrating that each artifact contributes unique and valuable information to malware detection. Models based on XML and native code artifacts were able to detect malware that escaped detection by models relying solely on Dalvik bytecode. We explored multiple approaches for combining these artifacts into unified image representations. While some combinations resulted in slight improvements, our findings indicated that Dalvik bytecode remains the most effective and efficient artifact for Android malware detection. These results suggest that while combining artifacts can provide marginal performance gains, Dalvik bytecode, due to its standalone capabilities, continues to be the preferred artifact for malware detection tasks.

Our second contribution introduced DexBERT, a pre-trained representation learning model designed to tackle various fine-grained Android analysis tasks. DexBERT processes disassembled Smali code from Dalvik bytecode and generates embeddings that capture nuanced malicious behaviors. To overcome the input length limitations present in original BERT models, we developed an aggregation method based on Auto-Encoder architectures. Freezing the parameters of the pre-trained DexBERT model allowed it to be directly applied to several class-level downstream tasks, including malicious code localization, Android application defect detection, and component type classification. Experimental results consistently demonstrated DexBERT's superior effectiveness compared to baseline models, showcasing its power in Android malware analysis.

Our third contribution introduced LaFiCMIL, a novel framework for large file classification based on correlated multiple instance learning (c-MIL). LaFiCMIL processes large document chunks as instances, enabling BERT-based feature extraction while minimizing information loss. Extensive experiments demonstrated that LaFiCMIL significantly outperforms state-of-the-art baselines in terms of both efficiency and accuracy across multiple benchmark datasets. Rather than developing an Android-specific approach, LaFiCMIL was designed as a versatile framework that can be applied to general BERT-like models. Additionally, LaFiCMIL's framework can be adapted for Android malware detection by enabling DexBERT to generate full-app-level representations, further expanding its applicability in the Android malware learning domain.

Our fourth contribution involved the development of DetectBERT, a framework adapted from LaFiCMIL that extends DexBERT's capabilities to app-level Android malware detection. By employing a correlated Multiple Instance Learning (c-MIL) strategy, DetectBERT effectively aggregates class-level features into app-level representations. This approach not only outperformed traditional feature aggregation methods but also surpassed existing state-of-the-art malware detection models, highlighting its potential to become a new standard for Android security analysis, representation learning, and related software engineering tasks.

Our fifth contribution addressed the critical challenge of Android malware family classification in a temporal-incremental learning (TIML) scenario. We developed ap-

proaches that incrementally update malware classifiers to effectively detect emerging malware families while adjusting to shifts in the data distribution of older families. By primarily utilizing current data samples for updates, TIML approaches reduce both time and resource consumption. We adapted Class-Incremental Learning (CIL) approaches to the TIML framework and proposed a novel multimodal TIML approach to further enhance classification performance. Our experimental results demonstrated that TIML offers promising performance in classifying malware families, significantly reducing the need for time-consuming retraining and storage, while maintaining high accuracy.

The contributions outlined in this dissertation advance the state of Android malware learning by tackling key challenges in representation learning, large file classification, malware detection, and temporal-incremental malware family classification. These findings open new research directions and provide a strong foundation for future advancements in Android security, representation learning, and related areas of software engineering.

9

Future work

This chapter enumerates relevant research directions for future work.

Contents

9.1 Multi-Artifact Representation Learning

Our research on multi-artifact assessment has demonstrated the potential of extracting valuable information from diverse Android artifacts, yet several open questions remain. One key research direction lies in optimizing the integration of these artifacts. For example, in our study, we applied the same image size to all artifacts, despite significant differences in their actual size and structure. For instance, `.dex` files can be tens of megabytes, while Manifest files are only a few kilobytes. Future work could focus on customizing image sizes or feature representations to match the unique characteristics of each artifact type, which could result in more effective multi-artifact integration and improved detection performance.

Further exploration is needed to develop more sophisticated methods for merging or concatenating representations of different artifacts. Existing approaches may not fully exploit the complementary nature of these artifacts, and advanced integration techniques—such as attention mechanisms or multimodal learning architectures—could unlock deeper insights into malware behaviors.

Additionally, expanding the range of artifacts analyzed presents a promising opportunity. Beyond Dalvik bytecode, XML, and native code, other data sources within Android apps—such as cryptographic certificates, images, audio files, UI layouts, and external metadata like user reviews—could provide new layers of information to enhance malware detection systems.

Finally, investigating the use of specialized deep learning architectures tailored to each artifact type is another valuable direction. Different artifacts may require distinct neural architectures to extract meaningful information, and designing architecture-specific models could improve both the efficiency and accuracy of malware detection.

9.2 Multiple Instance Learning for Software Engineering

Given the success of our correlated Multiple Instance Learning (c-MIL) framework in both natural language processing (LaFiCMIL) and Android analysis (DetectBERT), there is significant potential to extend this approach to other software engineering tasks. The c-MIL framework could be adapted for large-scale data inputs across a variety of software engineering applications, including code quality analysis, bug detection, and software architecture analysis.

Future research could explore how c-MIL-based models can be applied to codebases that span millions of lines, distributed systems with heterogeneous data, or even software performance profiling. Additionally, c-MIL could be useful for addressing challenges in software testing, such as identifying faulty code segments within large test cases. By extending c-MIL beyond the current scope, we expect to broaden its impact and applicability, contributing to more efficient software engineering practices.

9.3 Temporal-Incremental Malware Learning

Our work on Temporal-Incremental Malware Learning (TIML) marks the first formal formulation and in-depth investigation of this problem. We believe that this work paves the way for more adaptive, efficient, and robust malware classification systems, especially as the digital landscape becomes increasingly volatile.

Future research in this area could focus on improving TIML by developing more

advanced algorithms that handle concept drift and emerging malware families more effectively. Moreover, investigating the real-time application of TIML in production environments could lead to more agile malware detection systems capable of adapting to new threats as they emerge.

9.4 Malicious Code Localization

While DexBERT represents a significant advancement in the field of malicious code localization, this area remains largely underexplored. One of the biggest challenges is the lack of a comprehensive benchmark dataset for malicious code localization, which has traditionally required substantial manual effort and domain expertise to build.

The development of Large Language Models (LLMs) and related AI techniques now offers new possibilities for automating this process. Leveraging LLMs, it may now be feasible to construct a foundational benchmark dataset that captures a wide range of malicious behaviors across different applications. Once such a dataset is available, research can focus on developing more accurate and efficient localization techniques, potentially leading to better identification and mitigation of malware within large codebases.

Furthermore, combining deep learning techniques with static and dynamic analysis methods could enhance the precision of malicious code localization, leading to more actionable insights for security analysts.

List of Papers

Papers included in this dissertation:

- Sun, T., Daoudi, N., Allix, K. and Bissyandé, T.F., 2021, November. Android malware detection: looking beyond dalvik bytecode. In 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW) (pp. 34-39). IEEE.
- Sun, T., Allix, K., Kim, K., Zhou, X., Kim, D., Lo, D., Bissyandé, T.F. and Klein, J., 2023. Dexbert: Effective, task-agnostic and fine-grained representation learning of android bytecode. IEEE Transactions on Software Engineering.
- Sun, T., Pian, W., Daoudi, N., Allix, K., F. Bissyandé, T. and Klein, J., 2024, June. LaFiCMIL: Rethinking Large File Classification from the Perspective of Correlated Multiple Instance Learning. In International Conference on Applications of Natural Language to Information Systems (pp. 62-77). Cham: Springer Nature Switzerland.
- Sun, T., Daoudi, N., Kim, K., Allix, K., Bissyandé, T.F. and Klein, J., 2024. DetectBERT: Towards Full App-Level Representation Learning to Detect Android Malware. In 2024 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM).
- Sun, T., Daoudi, N., Pian, W., Kim, K., Allix, K., Bissyandé, T.F. and Klein, J., 2024. Temporal-Incremental Learning for Android Malware Detection. ACM Transactions on Software Engineering and Methodology.

Papers not included in this dissertation:

- Pian, W., Peng, H., Tang, X., Sun, T., Tian, H., Habib, A., Klein, J. and Bissyandé, T.F., 2023, June. MetaTPTrans: A meta learning approach for multilingual code representation learning. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 37, No. 4, pp. 5239-5247).
- Kim, K., Kim, J., Park, B., Kim, D., Chong, C.Y., Wang, Y., Sun, T., Tang, X., Klein, J. and Bissyandé, T.F., 2024, October. DataRecipe — How to Cook the Data for CodeLLM? In 2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE.
- Li, Y., Dang, X., Tian, H., Sun, T., Wang, Z., Ma, L., Klein, J. and Bissyandé, T.F., 2024. An empirical study of AI techniques in mobile applications. Journal of Systems and Software, p.112233.
- Pian, W., Li, Y., Tian, H., Sun, T., Song, Y., Tang, X., Habib, A., Klein, J. and Bissyandé, T.F., 2024. You Don't Have to Say Where to Edit! Joint Learning to Localize and Edit Source Code. ACM Transactions on Software Engineering and Methodology. (Under Revision)

Bibliography

[1] Statcounter, "Mobile operating system market share worldwide," https://gs.statcounter.com/os-market-share/mobile/worldwide, accessed: 2024-09-10.

[2] AV-ATLAS, "Total amount of malware and pua under android," https://portal.av-atlas.org/malware/statistics, accessed: 2024-09-10.

[3] D. Shen, G. Wang, W. Wang, M. R. Min, Q. Su, Y. Zhang, C. Li, R. Henao, and L. Carin, "Baseline needs more love: On simple word-embedding-based models and associated pooling mechanisms," *arXiv preprint arXiv:1805.09843*, 2018.

[4] Bankmycell, "How many android users are there? global and us statistics (2024)," https://www.bankmycell.com/blog/how-many-android-users-are-there, accessed: 2024-09-10.

[5] N. Pachhala, S. Jothilakshmi, and B. P. Battula, "A comprehensive survey on identification of malware types and malware classification using machine learning techniques," in *2021 2nd international conference on smart electronics and communication (ICOSEC)*. IEEE, 2021, pp. 1207–1214.

[6] Y. Zhao, L. Li, K. Liu, and J. Grundy, "Towards automatically repairing compatibility issues in published android apps," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2142–2153.

[7] R. Mayrhofer, J. V. Stoep, C. Brubaker, and N. Kralevich, "The android platform security model," *ACM Transactions on Privacy and Security (TOPS)*, vol. 24, no. 3, pp. 1–35, 2021.

[8] D. J. Leith and S. Farrell, "Contact tracing app privacy: What data is shared by europe's gaen contact tracing apps," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 2021, pp. 1–10.

[9] A. K. Sikder, G. Petracca, H. Aksu, T. Jaeger, and A. S. Uluagac, "A survey on sensor-based threats and attacks to smart devices and applications," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 2, pp. 1125–1159, 2021.

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

[11] M. N. Alenezi, H. Alabdulrazzaq, A. A. Alshaher, and M. M. Alkharang, "Evolution of malware threats and techniques: A review," *International journal of communication networks and information security*, vol. 12, no. 3, pp. 326–337, 2020.

[12] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of android malware and android analysis techniques," *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, pp. 1–41, 2017.

[13] H. Meng, V. L. Thing, Y. Cheng, Z. Dai, and L. Zhang, "A survey of android exploits in the wild," *Computers & Security*, vol. 76, pp. 71–91, 2018.

[14] V. Sihag, M. Vardhan, and P. Singh, "A survey of android application and malware hardening," *Computer Science Review*, vol. 39, p. 100365, 2021.

[15] T. Sharma and D. Rattan, "Malicious application detection in android—a systematic literature review," *Computer Science Review*, vol. 40, p. 100373, 2021.

[16] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in android apps with permission use analysis," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 611–622.

[17] S. Rosen, Z. Qian, and Z. M. Mao, "Appprofiler: a flexible method of exposing privacy-related behavior in android applications to end users," in *Proceedings of the third ACM conference on Data and application security and privacy*, 2013, pp. 221–232.

[18] H. Gao, S. Cheng, and W. Zhang, "Gdroid: Android malware detection and classification with graph convolutional network," *Computers & Security*, vol. 106, p. 102264, 2021.

[19] N. Daoudi, J. Samhi, A. K. Kabore, K. Allix, T. F. Bissyandé, and J. Klein, "Dexray: a simple, yet effective deep learning approach to android malware detection based on image representation of bytecode," in *Deployable Machine Learning for Security Defense: Second International Workshop, MLHat 2021, Virtual Event, August 15, 2021, Proceedings 2*. Springer, 2021, pp. 81–106.

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

[21] P. Faruki, V. Ganmoor, V. Laxmi, M. S. Gaur, and A. Bharmal, "Androsimilar: robust statistical feature signature for android malware detection," in *Proceedings of the 6th International Conference on Security of Information and Networks*, 2013, pp. 152–159.

[22] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *Ndss*, vol. 14, 2014, pp. 23–26.

[23] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-An, and H. Ye, "Significant permission identification for machine-learning-based android malware detection," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3216–3225, 2018.

[24] L. Wen and H. Yu, "An android malware detection system based on machine learning," in *AIP conference proceedings*, vol. 1864, no. 1. AIP Publishing, 2017.

[25] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "Dl-droid: Deep learning based android malware detection using real devices," *Computers & Security*, vol. 89, p. 101663, 2020.

[26] A. Narayanan, M. Chandramohan, L. Chen, and Y. Liu, "A multi-view context-aware approach to android malware detection and malicious code localization," *Empirical Software Engineering*, vol. 23, pp. 1222–1274, 2018.

[27] F. Alswaina and K. Elleithy, "Android malware family classification and analysis: Current status and future directions," *Electronics*, vol. 9, no. 6, p. 942, 2020.

[28] S. Freitas, R. Duggal, and D. H. Chau, "Malnet: A large-scale image database of malicious software," in *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, 2022, pp. 3948–3952.

[29] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th international conference on mining software repositories*, 2016, pp. 468–471.

[30] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé *et al.*, "Deep android malware detection," in *Proceedings of the seventh ACM on conference on data and application security and privacy*, 2017, pp. 301–308.

[31] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro, "Transcend: Detecting concept drift in malware classification models," in *26th USENIX security symposium (USENIX security 17)*, 2017, pp. 625–642.

[32] F. Barbero, F. Pendlebury, F. Pierazzi, and L. Cavallaro, "Transcending transcend: Revisiting malware classification in the presence of concept drift," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 805–823.

[33] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[34] M. D. Ernst, "Natural language is a programming language: Applying natural language processing to software development," in *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[35] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[36] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[37] W. Hua, Y. Sui, Y. Wan, G. Liu, and G. Xu, "Fcca: Hybrid code representation for functional clone detection using attention networks," *IEEE Transactions on Reliability*, vol. 70, no. 1, pp. 304–318, 2020.

[38] A. Xu, T. Dai, H. Chen, Z. Ming, and W. Li, "Vulnerability detection for source code using contextual lstm," in *2018 5th International Conference on Systems and Informatics (ICSAI)*. IEEE, 2018, pp. 1225–1230.

[39] J. K. Siow, C. Gao, L. Fan, S. Chen, and Y. Liu, "Core: Automating review recommendation for code changes," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 284–295.

[40] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Thirtieth AAAI conference on artificial intelligence*, 2016.

[41] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," *arXiv preprint arXiv:1503.00075*, 2015.

[42] Y. Wan, J. Shu, Y. Sui, G. Xu, Z. Zhao, J. Wu, and P. Yu, "Multi-modal attention network learning for semantic source code retrieval," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 13–25.

[43] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 261–271.

[44] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "Albert: A lite bert for self-supervised learning of language representations," *arXiv preprint arXiv:1909.11942*, 2019.

[45] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[46] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," *ArXiv*, vol. abs/2009.08366, 2021.

[47] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Advances in neural information processing systems*, vol. 35, pp. 27 730–27 744, 2022.

[48] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[49] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[50] ——, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[51] Anthropic, "The claude 3 model family: Opus, sonnet, haiku," 2024, accessed: 2024-11-05. [Online]. Available: https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf

[52] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[53] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei *et al.*, "Starcoder 2 and the stack v2: The next generation," *arXiv preprint arXiv:2402.19173*, 2024.

[54] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[55] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[56] C. Team, "Codegemma: Open code models based on gemma," *arXiv preprint arXiv:2406.11409*, 2024.

[57] T. H. Le, H. Chen, and M. A. Babar, "Deep learning for source code modeling and generation: Models, applications, and challenges," *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–38, 2020.

[58] H. P. Samoaa, F. Bayram, P. Salza, and P. Leitner, "A systematic mapping study of source code representation for deep learning in software engineering," *IET Software*, vol. 16, no. 4, pp. 351–385, 2022.

[59] S. Wang, L. Huang, A. Gao, J. Ge, T. Zhang, H. Feng, I. Satyarth, M. Li, H. Zhang, and V. Ng, "Machine/deep learning for software engineering: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 49, no. 3, pp. 1188–1231, 2022.

[60] R. A. Husein, H. Aburajouh, and C. Catal, "Large language models for code completion: A systematic literature review," *Computer Standards & Interfaces*, p. 103917, 2024.

[61] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *arXiv preprint arXiv:2406.00515*, 2024.

[62] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 175–186.

[63] Z. Xu, K. Ren, S. Qin, and F. Craciun, "Cdgdroid: Android malware detection based on deep learning using cfg and dfg," in *International Conference on Formal Engineering Methods*. Springer, 2018, pp. 177–193.

[64] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "Maldozer: Automatic framework for android malware detection using deep learning," *Digital Investigation*, vol. 24, pp. S48–S59, 2018.

[65] X. Xiao, X. Xiao, Y. Jiang, X. Liu, and R. Ye, "Identifying android malware with system call co-occurrence matrices," *Transactions on Emerging Telecommunications Technologies*, vol. 27, no. 5, pp. 675–684, 2016.

[66] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio, "Detecting android malware using sequences of system calls," in *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, 2015, pp. 13–20.

[67] X. Xiao, Z. Wang, Q. Li, S. Xia, and Y. Jiang, "Back-propagation neural network on markov chains from system call sequences: a new approach for detecting android malware with system call sequences," *IET Information Security*, vol. 11, no. 1, pp. 8–15, 2017.

[68] L. Singh and M. Hofmann, "Dynamic behavior analysis of android applications for malware detection," in *2017 International Conference on Intelligent Communication and Computational Techniques (ICCT)*. IEEE, 2017, pp. 1–7.

[69] T. Bhatia and R. Kaushal, "Malware detection in android based on dynamic analysis," in *2017 International Conference on Cyber Security And Protection Of Digital Services*. IEEE, 2017, pp. 1–6.

[70] J. Qiu, J. Zhang, W. Luo, L. Pan, S. Nepal, and Y. Xiang, "A survey of android malware detection with deep neural models," *ACM Computing Surveys (CSUR)*, vol. 53, no. 6, pp. 1–36, 2020.

[71] D. Li, Z. Wang, and Y. Xue, "Fine-grained android malware detection based on deep learning," in *2018 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2018, pp. 1–2.

[72] J. Booz, J. McGiff, W. G. Hatcher, W. Yu, J. Nguyen, and C. Lu, "Tuning deep learning performance for android malware detection," in *2018 19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. IEEE, 2018, pp. 140–145.

[73] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial examples for malware detection," in *European symposium on research in computer security*. Springer, 2017, pp. 62–79.

[74] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "{TESSERACT}: Eliminating experimental bias in malware classification across space and time," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 729–746.

[75] S. Hou, A. Saas, Y. Ye, and L. Chen, "Droiddelver: An android malware detection system using deep belief network based on api call blocks," in *International conference on web-age information management*. Springer, 2016, pp. 54–66.

[76] A. Naway and Y. Li, "Using deep neural network for android malware detection," *International Journal of Advanced Studies in Computers, Science and Engineering*, vol. 7, no. 12, pp. 9–18, 2018.

[77] ——, "Android malware detection using autoencoder," *arXiv preprint arXiv:1901.07315*, 2019.

[78] W. Y. Lee, J. Saxe, and R. Harang, "Seqdroid: Obfuscated android malware detection using stacked convolutional and recurrent neural networks," in *Deep learning applications for cyber security*. Springer, 2019, pp. 197–210.

[79] N. He, T. Wang, P. Chen, H. Yan, and Z. Jin, "An android malware detection method based on deep autoencoder," in *2018 artificial intelligence and cloud computing conference*, 2018, pp. 88–93.

[80] Q. Jerome, K. Allix, R. State, and T. Engel, "Using opcode-sequences to detect malicious android applications," in *2014 IEEE International Conference on Communications (ICC)*, 2014, pp. 914–919.

[81] K. Allix, T. F. Bissyandé, Q. Jérome, J. Klein, R. State, and Y. Le Traon, "Empirical assessment of machine learning-based malware detectors for android," *Empirical Software Engineering*, vol. 21, no. 1, pp. 183–211, Feb 2016. [Online]. Available: https://doi.org/10.1007/s10664-014-9352-6

[82] R. Vinayakumar, K. Soman, P. Poornachandran, and S. Sachin Kumar, "Detecting android malware using long short-term memory (lstm)," *Journal of Intelligent & Fuzzy Systems*, vol. 34, no. 3, pp. 1277–1288, 2018.

[83] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, "Droid-sec: deep learning in android malware detection," in *Proceedings of the 2014 ACM conference on SIGCOMM*, 2014, pp. 371–372.

[84] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. A. Porras, "Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications," in *ESORICS*, 2014.

[85] M. A. Atici, Şeref Sağiroğlu, and I. A. Dogru, "Android malware analysis approach based on control flow graphs and machine learning algorithms," *2016 4th International Symposium on Digital Forensic and Security (ISDFS)*, pp. 26–31, 2016.

[86] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[87] J. Bruna, W. Zaremba, A. D. Szlam, and Y. LeCun, "Spectral networks and locally connected networks on graphs," *CoRR*, vol. abs/1312.6203, 2014.

[88] T. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *ArXiv*, vol. abs/1609.02907, 2017.

[89] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, "graph2vec: Learning distributed representations of graphs," *arXiv preprint arXiv:1707.05005*, 2017.

[90] T. Bilot, N. El Madhoun, K. Al Agha, and A. Zouaoui, "A survey on malware detection with graph representation learning," *ACM Computing Surveys*, vol. 56, no. 11, pp. 1–36, 2024.

[91] T. Sutter, T. Kehrer, M. Rennhard, B. Tellenbach, and J. Klein, "Dynamic security analysis on android: A systematic literature review," *IEEE Access*, 2024.

[92] B. Casey, J. Santos, and G. Perry, "A survey of source code representations for machine learning-based cybersecurity tasks," *arXiv preprint arXiv:2403.10646*, 2024.

[93] N.-U.-R. Chowdhury, A. Haque, H. Soliman, M. S. Hossen, T. Fatima, and I. Ahmed, "Android malware detection using machine learning: A review," in *Intelligent Systems Conference*. Springer, 2023, pp. 507–522.

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

[95] H. Kang, J.-w. Jang, A. Mohaisen, and H. K. Kim, "Detecting and classifying android malware using static analysis along with creator information," *International Journal of Distributed Sensor Networks*, vol. 11, no. 6, p. 479174, 2015.

[96] P. Faruki, V. Ganmoor, V. Laxmi, M. S. Gaur, and A. Bharmal, "Androsimilar: Robust statistical feature signature for android malware detection," in *Proceedings of the 6th International Conference on Security of Information and Networks*, ser. SIN '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 152–159. [Online]. Available: https://doi-org.proxy.bnl.lu/10.1145/2523514.2523539

[97] C. Li, K. Mills, D. Niu, R. Zhu, H. Zhang, and H. Kinawi, "Android malware detection based on factorization machine," *IEEE Access*, vol. 7, pp. 184 008–184 019, 2019.

[98] Y. Wu, X. Li, D. Zou, W. Yang, X. Zhang, and H. Jin, "Malscan: Fast market-wide mobile malware scanning by social-network centrality analysis," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 139–150.

[99] T. Hsien-De Huang and H.-Y. Kao, "R2-d2: Color-inspired convolutional neural network (cnn)-based android malware detections," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 2633–2642.

[100] Y. Ding, X. Zhang, J. Hu, and W. Xu, "Android malware detection method based on bytecode image," *Journal of Ambient Intelligence and Humanized Computing*, 2020.

[101] J. Garcia, M. Hammad, and S. Malek, "Lightweight, obfuscation-resilient detection and family identification of android malware," *ACM Trans. Softw. Eng. Methodol.*, vol. 26, no. 3, Jan. 2018. [Online]. Available: https://doi.org/10.1145/3162625

[102] D. Wu, C. Mao, T. Wei, H. Lee, and K. Wu, "Droidmat: Android malware detection through manifest and api calls tracing," in *2012 Seventh Asia Joint Conference on Information Security*, 2012, pp. 62–69.

[103] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models," *arXiv preprint arXiv:1612.04433*, 2016.

[104] H. Fereidooni, M. Conti, D. Yao, and A. Sperduti, "Anastasia: Android malware detection using static analysis of applications," in *2016 8th IFIP international conference on new technologies, mobility and security (NTMS)*. IEEE, 2016, pp. 1–5.

[105] Z. Wang, J. Cai, S. Cheng, and W. Li, "Droiddeeplearner: Identifying android malware using deep learning," in *2016 IEEE 37th Sarnoff Symposium*, 2016, pp. 160–165.

[106] Z. Yuan, Y. Lu, and Y. Xue, "Droiddetector: android malware characterization and detection using deep learning," *Tsinghua Science and Technology*, vol. 21, no. 1, pp. 114–123, 2016.

[107] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "Dynalog: an automated dynamic analysis framework for characterizing android applications," in *2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security)*, 2016, pp. 1–8.

[108] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 23–26.

[109] H. Sandeep, "Static analysis of android malware detection using deep learning," in *2019 International Conference on Intelligent Computing and Control Systems (ICCS)*. IEEE, 2019, pp. 841–845.

[110] Y. Pan, X. Ge, C. Fang, and Y. Fan, "A systematic literature review of android malware detection using static analysis," *IEEE Access*, vol. 8, pp. 116 363–116 379, 2020.

[111] T. Sun, N. Daoudi, K. Allix, and T. F. Bissyandé, "Android malware detection: Looking beyond dalvik bytecode," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. IEEE, 2021, pp. 34–39.

[112] A. Pektaş and T. Acarman, "Deep learning for effective android malware detection using api call graph embeddings," *Soft Computing*, vol. 24, pp. 1027–1043, 2020.

[113] Y. Yang, X. Du, Z. Yang, and X. Liu, "Android malware detection based on structural features of the function call graph," *Electronics*, vol. 10, no. 2, p. 186, 2021.

[114] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro, "Droidsieve: Fast and accurate classification of obfuscated android malware," in *Proceedings of the seventh ACM on conference on data and application security and privacy*, 2017, pp. 309–320.

[115] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning*, vol. 63, pp. 3–42, 2006.

[116] H. M. Kim, H. M. Song, J. W. Seo, and H. K. Kim, "Andro-simnet: Android malware family classification using social network analysis," in *2018 16th Annual Conference on Privacy, Security and Trust (PST)*. IEEE, 2018, pp. 1–8.

[117] L. Deshotels, V. Notani, and A. Lakhotia, "Droidlegacy: Automated familial classification of android malware," in *Proceedings of ACM SIGPLAN on program protection and reverse engineering workshop 2014*, 2014, pp. 1–12.

[118] Y. Fang, Y. Gao, F. Jing, and L. Zhang, "Android malware familial classification based on dex file section features," *IEEE Access*, vol. 8, pp. 10 614–10 627, 2020.

[119] Y. Sun, Y. Chen, Y. Pan, and L. Wu, "Android malware family classification based on deep learning of code images," *IAENG International Journal of Computer Science*, vol. 46, no. 4, pp. 524–533, 2019.

[120] N. Daoudi, K. Allix, T. F. Bissyandé, and J. Klein, "Lessons learnt on reproducibility in machine learning based android malware detection," *Empirical Software Engineering*, vol. 26, no. 4, pp. 1–53, 2021.

[121] H. H. R. Manzil and S. M. Naik, "Detection approaches for android malware: Taxonomy and review analysis," *Expert Systems with Applications*, vol. 238, p. 122255, 2024.

[122] S. K. Smmarwar, G. P. Gupta, and S. Kumar, "Android malware detection and identification frameworks by leveraging the machine and deep learning

techniques: A comprehensive review," *Telematics and Informatics Reports*, p. 100130, 2024.

[123] Y. Ji, H. Liu, B. He, X. Xiao, H. Wu, and Y. Yu, "Diversified multiple instance learning for document-level multi-aspect sentiment classification," in *EMNLP*, 2020.

[124] K. Song, L. Bing, W. Gao, J. Lin, L. Zhao, J. Wang, C. Sun, X. Liu, and Q. Zhang, "Using customer service dialogues for satisfaction analysis with context-assisted multiple instance learning," in *EMNLP*, 2019.

[125] R. Hebbar, P. Papadopoulos, R. Reyes, A. F. Danvers, A. J. Polsinelli, S. Moseley, D. Sbarra, M. R. Mehl, and S. Narayanan, "Deep multiple instance learning for foreground speech localization in ambient audio from wearable devices," *Audio, Speech, and Music Processing*, 2021.

[126] F. Kanavati, G. Toyokawa, S. Momosaki, M. Rambeau, Y. Kozuma, F. Shoji, K. Yamazaki, S. Takeo, O. Iizuka, and M. Tsuneki, "Weakly-supervised learning for lung carcinoma classification using deep learning," *Scientific reports*, 2020.

[127] G. Xu, Z. Song, Z. Sun, C. Ku, Z. Yang, C. Liu, S. Wang, J. Ma, and W. Xu, "Camel: A weakly supervised learning framework for histopathology image segmentation," in *ICCV*, 2019.

[128] J. Feng and Z.-H. Zhou, "Deep miml network," in *AAAI*, 2017.

[129] M. Lerousseau, M. Vakalopoulou, M. Classe, J. Adam, E. Battistella, A. Carré, T. Estienne, T. Henry, E. Deutsch, and N. Paragios, "Weakly supervised multiple instance learning histopathological tumor segmentation," in *MICCAI*, 2020.

[130] M. Y. Lu, D. F. Williamson, T. Y. Chen, R. J. Chen, M. Barbieri, and F. Mahmood, "Data-efficient and weakly supervised computational pathology on whole-slide images," *Nature biomedical engineering*, 2021.

[131] Y. Sharma, A. Shrivastava, L. Ehsan, C. A. Moskaluk, S. Syed, and D. Brown, "Cluster-to-conquer: A framework for end-to-end multi-instance learning for whole slide image classification," in *Medical Imaging with Deep Learning*, 2021.

[132] B. Li, Y. Li, and K. W. Eliceiri, "Dual-stream multiple instance learning network for whole slide image classification with self-supervised contrastive learning," in *CVPR*, 2021.

[133] Z. Shao, H. Bian, Y. Chen, Y. Wang, J. Zhang *et al.*, "Transmil: Transformer based correlated multiple instance learning for whole slide image classification," in *NeurIPS*, 2021.

[134] H. Zhang, Y. Meng, Y. Zhao, Y. Qiao, X. Yang, S. Coupland, and Y. Zheng, "Dtfd-mil: Double-tier feature distillation multiple instance learning for histopathology whole slide image classification," in *CVPR*, 2022.

[135] X. Wang, Y. Yan, P. Tang, X. Bai, and W. Liu, "Revisiting multiple instance neural networks," *Pattern Recognition*, 2018.

[136] Z.-H. Zhou, Y.-Y. Sun, and Y.-F. Li, "Multi-instance learning by treating instances as non-iid samples," in *ICML*, 2009.

[137] W. Zhang, "Non-iid multi-instance learning for predicting instance and bag labels using variational auto-encoder," *arXiv preprint arXiv:2105.01276*, 2021.

[138] H. M. Gomes, J. P. Barddal, F. Enembreck, and A. Bifet, "A survey on ensemble learning for data stream classification," *ACM Computing Surveys (CSUR)*, vol. 50, no. 2, pp. 1–36, 2017.

[139] L. Golab and M. T. Özsu, "Issues in data stream management," *ACM Sigmod Record*, vol. 32, no. 2, pp. 5–14, 2003.

[140] M. A. P. Chamikara, P. Bertók, D. Liu, S. Camtepe, and I. Khalil, "Efficient data perturbation for privacy preserving and accurate data stream mining," *Pervasive and Mobile Computing*, vol. 48, pp. 1–19, 2018.

[141] M. De Lange, R. Aljundi, M. Masana, S. Parisot, X. Jia, A. Leonardis, G. Slabaugh, and T. Tuytelaars, "A continual learning survey: Defying forgetting in classification tasks," *IEEE transactions on pattern analysis and machine intelligence*, vol. 44, no. 7, pp. 3366–3385, 2021.

[142] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," *ACM computing surveys (CSUR)*, vol. 46, no. 4, pp. 1–37, 2014.

[143] G. Ditzler, M. Roveri, C. Alippi, and R. Polikar, "Learning in nonstationary environments: A survey," *IEEE Computational Intelligence Magazine*, vol. 10, no. 4, pp. 12–25, 2015.

[144] D.-W. Zhou, Q.-W. Wang, Z.-H. Qi, H.-J. Ye, D.-C. Zhan, and Z. Liu, "Deep class-incremental learning: A survey," *arXiv preprint arXiv:2302.03648*, 2023.

[145] S.-A. Rebuffi, A. Kolesnikov, G. Sperl, and C. H. Lampert, "icarl: Incremental classifier and representation learning," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2017, pp. 2001–2010.

[146] A. Douillard, M. Cord, C. Ollion, T. Robert, and E. Valle, "Podnet: Pooled outputs distillation for small-tasks incremental learning," in *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XX 16*. Springer, 2020, pp. 86–102.

[147] H. Ahn, J. Kwak, S. Lim, H. Bang, H. Kim, and T. Moon, "Ss-il: Separated softmax for incremental learning," in *Proceedings of the IEEE/CVF International conference on computer vision*, 2021, pp. 844–853.

[148] M. Kang, J. Park, and B. Han, "Class-incremental learning by knowledge distillation with adaptive feature consolidation," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 16 071–16 080.

[149] W. Pian, S. Mo, Y. Guo, and Y. Tian, "Audio-visual class-incremental learning," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023.

[150] S. Mo, W. Pian, and Y. Tian, "Class-incremental grouping network for continual audio-visual learning," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023.

[151] Z. Li and D. Hoiem, "Learning without forgetting," *IEEE transactions on pattern analysis and machine intelligence*, vol. 40, no. 12, pp. 2935–2947, 2017.

[152] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang, "Learning under concept drift: A review," *IEEE transactions on knowledge and data engineering*, vol. 31, no. 12, pp. 2346–2363, 2018.

[153] Y. Wang, Z. Huang, and X. Hong, "S-prompts learning with pre-trained transformers: An occam's razor for domain incremental learning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 5682–5695, 2022.

[154] E. Fini, V. G. T. Da Costa, X. Alameda-Pineda, E. Ricci, K. Alahari, and J. Mairal, "Self-supervised models are continual learners," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 9621–9630.

[155] Z. Wang, Z. Zhang, C.-Y. Lee, H. Zhang, R. Sun, X. Ren, G. Su, V. Perot, J. Dy, and T. Pfister, "Learning to prompt for continual learning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 139–149.

[156] J. Xie, S. Yan, and X. He, "General incremental learning with domain-aware categorical representations," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 14 351–14 360.

[157] J. Li, D. Xue, W. Wu, and J. Wang, "Incremental learning for malware classification in small datasets," *Security and Communication Networks*, vol. 2020, pp. 1–12, 2020.

[158] J. Weston, C. Watkins *et al.*, "Support vector machines for multi-class pattern recognition." in *Esann*, vol. 99, 1999, pp. 219–224.

[159] Q. Qiang, M. Cheng, Y. Hu, Y. Zhou, J. Sun, Y. Ding, Z. Qi, and F. Jiao, "An incremental malware classification approach based on few-shot learning," in *ICC 2022-IEEE International Conference on Communications*. IEEE, 2022, pp. 2682–2687.

[160] G. Renjith and S. Aji, "On-device resilient android malware detection using incremental learning," *Procedia Computer Science*, vol. 215, pp. 929–936, 2022.

[161] M. S. Rahman, S. Coull, and M. Wright, "On the limitations of continual learning for malware classification," in *Conference on Lifelong Learning Agents*. PMLR, 2022, pp. 564–582.

[162] Y. Chen, Z. Ding, and D. Wagner, "Continuous learning for android malware detection," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1127–1144.

[163] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: hindering dynamic analysis of android malware," in *Proceedings of the seventh european workshop on system security*, 2014, pp. 1–6.

[164] J. Garcia, M. Hammad, and S. Malek, "Lightweight, obfuscation-resilient detection and family identification of android malware," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 26, no. 3, pp. 1–29, 2018.

[165] H. Cai, N. Meng, B. Ryder, and D. Yao, "Droidcat: Effective android malware detection and categorization via app-level profiling," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 6, pp. 1455–1470, 2018.

[166] W.-C. Wu and S.-H. Hung, "Droiddolphin: a dynamic android malware detection framework using big data and machine learning," in *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems*, 2014, pp. 247–252.

[167] F. Martinelli, F. Mercaldo, and A. Saracino, "Bridemaid: An hybrid tool for accurate detection of android malware," in *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, 2017, pp. 899–901.

[168] W. Niu, R. Cao, X. Zhang, K. Ding, K. Zhang, and T. Li, "Opcode-level function call graph based android malware classification using deep learning," *Sensors*, vol. 20, no. 13, 2020. [Online]. Available: https://www.mdpi.com/1424-8220/20/13/3645

[169] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 783–794.

[170] Y.-X. Ding, W.-G. Zhao, Z.-P. Wang, and L.-F. Wang, "Automaticlly learning featurs of android apps using cnn," in *2018 International Conference on Machine Learning and Cybernetics (ICMLC)*, vol. 1. IEEE, 2018, pp. 331–336.

[171] S. Raschka, "Model evaluation, model selection, and algorithm selection in machine learning," *arXiv preprint arXiv:1811.12808*, 2018.

[172] A. F. Agarap, "Deep learning using rectified linear units (relu)," *arXiv preprint arXiv:1803.08375*, 2018.

[173] S. Sharma and S. Sharma, "Activation functions in neural networks," *Towards Data Science*, vol. 6, no. 12, pp. 310–316, 2017.

[174] T. Sun, K. Allix, K. Kim, X. Zhou, D. Kim, D. Lo, T. F. Bissyandé, and J. Klein, "Dexbert: Effective, task-agnostic and fine-grained representation learning of android bytecode," *IEEE Transactions on Software Engineering*, 2023.

[175] J. Devlin, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[176] S. Mani, A. Sankaran, and R. Aralikatte, "Deeptriage: Exploring the effectiveness of deep learning for bug triaging," in *Proceedings of the ACM India Joint International Conference on Data Science and Management of Data*, 2019, pp. 171–179.

[177] H. J. Kang, T. F. Bissyandé, and D. Lo, "Assessing the generalizability of code2vec token embeddings," in *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1–12.

[178] Z. Ma, H. Ge, Z. Wang, Y. Liu, and X. Liu, "Droidetec: Android malware detection and malicious code localization through deep learning," *arXiv preprint arXiv:2002.03594*, 2020.

[179] R. S. Arslan, "Androanalyzer: android malicious software detection based on deep learning," *PeerJ Computer Science*, vol. 7, p. e533, 2021.

[180] F. Dong, J. Wang, Q. Li, G. Xu, and S. Zhang, "Defect prediction in android binary executables using deep neural network," *Wireless Personal Communications*, vol. 102, no. 3, pp. 2261–2285, 2018.

[181] A. Narayanan, C. Soh, L. Chen, Y. Liu, and L. Wang, "apk2vec: Semi-supervised multi-view representation learning for profiling android applications," in *2018 IEEE International Conference on Data Mining (ICDM)*. IEEE, 2018, pp. 357–366.

[182] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall, "Method-level bug prediction," in *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2012, pp. 171–180.

[183] M. Singh and V. Sharma, "Detection of file level clone for high level cloning," *Procedia Computer Science*, vol. 57, pp. 915–922, 2015.

[184] C. Tantithamthavorn, S. L. Abebe, A. E. Hassan, A. Ihara, and K. Matsumoto, "The impact of ir-based classifier configuration on the performance and the effort of method-level bug localization," *Information and Software Technology*, vol. 102, pp. 160–174, 2018.

[185] W. Zhang, Z. Li, Q. Wang, and J. Li, "Finelocator: A novel approach to method-level fine-grained bug localization by query expansion," *Information and Software Technology*, vol. 110, pp. 121–135, 2019.

[186] V. Frick, "Understanding software changes: extracting, classifying, and presenting fine-grained source code changes," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, 2020, pp. 226–229.

[187] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[188] D. Bamman and N. A. Smith, "New alignment methods for discriminative book summarization," *arXiv preprint arXiv:1305.1319*, 2013.

[189] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[190] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[191] S. Fadnavis, "Image interpolation techniques in digital image processing: an overview," *International Journal of Engineering Research and Applications*, vol. 4, no. 10, pp. 70–73, 2014.

[192] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler, "Aligning books and movies: Towards story-like visual explanations by watching movies and reading books," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 19–27.

[193] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[194] G. Meng, Y. Xue, C. Mahinthan, A. Narayanan, Y. Liu, J. Zhang, and T. Chen, "Mystique: Evolving android malware for auditing anti-malware tools," in *Proceedings of the 11th ACM on Asia conference on computer and communications security*, 2016, pp. 365–376.

[195] J. Samhi, L. Li, T. F. Bissyandé, and J. Klein, "Difuzer: Uncovering suspicious hidden sensitive operations in android apps," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 723–735.

[196] "https://checkmarx.com/," August 2022.

[197] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[198] T. Sun, W. Pian, N. Daoudi, K. Allix, T. F. Bissyandé, and J. Klein, "Laficmil: Rethinking large file classification from the perspective of correlated multiple instance learning," in *International Conference on Applications of Natural Language to Information Systems*. Springer, 2024, pp. 62–77.

[199] K. Kowsari, K. Jafari Meimandi, M. Heidarysafa, S. Mendu, L. Barnes, and D. Brown, "Text classification algorithms: A survey," *Information*, vol. 10, no. 4, p. 150, 2019.

[200] N. C. Dang, M. N. Moreno-García, and F. De la Prieta, "Sentiment analysis based on deep learning: A comparative study," *Electronics*, vol. 9, no. 3, p. 483, 2020.

[201] S. Kumar, R. Asthana, S. Upadhyay, N. Upreti, and M. Akbar, "Fake news detection using deep learning models: A novel approach," *Transactions on Emerging Telecommunications Technologies*, vol. 31, no. 2, p. e3767, 2020.

[202] T. Ranasinghe and M. Zampieri, "Multilingual offensive language identification with cross-lingual embeddings," *arXiv preprint arXiv:2010.05324*, 2020.

[203] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *NeurIPS*, 2017.

[204] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-training," 2018.

[205] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[206] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," *arXiv preprint arXiv:2004.05150*, 2020.

[207] A. Bulatov, Y. Kuratov, and M. Burtsev, "Recurrent memory transformer," *Advances in Neural Information Processing Systems*, vol. 35, pp. 11 079–11 091, 2022.

[208] A. Bulatov, Y. Kuratov, and M. S. Burtsev, "Scaling transformer to 1m tokens and beyond with rmt," *arXiv preprint arXiv:2304.11062*, 2023.

[209] Y. Zhang, M. Wang, C. Ren, Q. Li, P. Tiwari, B. Wang, and J. Qin, "Pushing the limit of llm capacity for text classification," *arXiv preprint arXiv:2402.07470*, 2024.

[210] M. Ilse, J. Tomczak, and M. Welling, "Attention-based deep multiple instance learning," in *ICML*, 2018.

[211] G. Rote, "Computing the minimum hausdorff distance between two point sets on a line under translation," *Information Processing Letters*, vol. 38, no. 3, pp. 123–127, 1991.

[212] Y. Xiong, Z. Zeng, R. Chakraborty, M. Tan, G. Fung, Y. Li, and V. Singh, "Nyströmformer: A nyström-based algorithm for approximating self-attention," in *AAAI*, 2021.

[213] A. Ben-Israel and T. N. Greville, *Generalized inverses: theory and applications*. Springer Science & Business Media, 2003, vol. 15.

[214] M. K. Razavi, A. Kerayechian, M. Gachpazan, and S. Shateyi, "A new iterative method for finding approximate inverses of complex matrices," in *Abstract and Applied Analysis*, 2014.

[215] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[216] C. T. Baker, *The numerical treatment of integral equations*. Oxford University Press, 1977.

[217] H. Park, Y. Vyas, and K. Shah, "Efficient classification of long documents using transformers," in *ACL*, 2022.

[218] J. Kiesel, M. Mestre, R. Shukla, E. Vincent, P. Adineh, D. Corney, B. Stein, and M. Potthast, "Semeval-2019 task 4: Hyperpartisan news detection," in *13th International Workshop on Semantic Evaluation*, 2019.

[219] K. Lang, "Newsweeder: Learning to filter netnews," in *Machine Learning Proceedings 1995*, 1995, pp. 331–339.

[220] I. Chalkidis, M. Fergadiotis, P. Malakasiotis, and I. Androutsopoulos, "Large-scale multi-label text classification on eu legislation," *arXiv:1906.02192*, 2019.

[221] H. Hanif and S. Maffeis, "Vulberta: Simplified source code pre-training for vulnerability detection," *arXiv preprint arXiv:2205.12424*, 2022.

[222] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, 2019.

[223] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.

[224] T. Sun, N. Daoudi, K. Kim, K. Allix, T. F. Bissyandé, and J. Klein, "Detectbert: Towards full app-level representation learning to detect android malware," in *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2024, pp. 420–426.

[225] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[226] M. Zhang, J. Lucas, J. Ba, and G. Hinton, "Lookahead optimizer: k steps forward, 1 step back," in *NeurIPS*, 2019.

[227] K. Xu, Y. Li, R. Deng, K. Chen, and J. Xu, "Droidevolver: Self-evolving android malware detection system," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 47–62.

[228] X. Zhang, Y. Zhang, M. Zhong, D. Ding, Y. Cao, Y. Zhang, M. Zhang, and M. Yang, "Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware," in *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, 2020, pp. 757–770.

[229] T. Sun, N. Daoudi, W. Pian, K. Kim, K. Allix, T. F. Bissyande, and J. Klein, "Temporal-incremental learning for android malware detection," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 4, pp. 1–30, 2025.

[230] J. B. Fraley and J. Cannady, "The promise of machine learning in cybersecurity," in *SoutheastCon 2017*. IEEE, 2017, pp. 1–6.

[231] S. Yan, J. Ren, W. Wang, L. Sun, W. Zhang, and Q. Yu, "A survey of adversarial attack and defense methods for malware classification in cyber security," *IEEE Communications Surveys & Tutorials*, 2022.

[232] I. B. Abdel Ouahab, M. Bouhorma, L. El Aachak, and A. A. Boudhir, "Towards a new cyberdefense generation: proposition of an intelligent cybersecurity

framework for malware attacks," *Recent Advances in Computer Science and Communications (Formerly: Recent Patents on Computer Science)*, vol. 15, no. 8, pp. 1026–1042, 2022.

[233] G. Xiao, J. Li, Y. Chen, and K. Li, "Malfcs: An effective malware classification framework with automated feature extraction based on deep convolutional neural networks," *Journal of Parallel and Distributed Computing*, vol. 141, pp. 49–58, 2020.

[234] S. Acharya, U. Rawat, and R. Bhatnagar, "A low computational cost method for mobile malware detection using transfer learning and familial classification using topic modelling," *Applied Computational Intelligence and Soft Computing*, vol. 2022, pp. 1–22, 2022.

[235] Y. Wu, X. Li, D. Zou, W. Yang, X. Zhang, and H. Jin, "Malscan: Fast market-wide mobile malware scanning by social-network centrality analysis," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 139–150.

[236] Y. Li, D. Yuan, T. Zhang, H. Cai, D. Lo, C. Gao, X. Luo, and H. Jiang, "Meta-learning for multi-family android malware classification," *ACM Transactions on Software Engineering and Methodology*, 2024.

[237] B. Wu, S. Chen, C. Gao, L. Fan, Y. Liu, W. Wen, and M. R. Lyu, "Why an android app is classified as malware: Toward malware classification interpretation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 2, pp. 1–29, 2021.

[238] H. Li, G. Xu, L. Wang, X. Xiao, X. Luo, G. Xu, and H. Wang, "Malcertain: Enhancing deep neural network based android malware detection by tackling prediction uncertainty," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[239] Z. Liu, L. F. Zhang, and Y. Tang, "Enhancing malware detection for android apps: Detecting fine-granularity malicious components," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1212–1224.

[240] L. Cui, J. Yin, J. Cui, Y. Ji, P. Liu, Z. Hao, and X. Yun, "Api2vec++: Boosting api sequence representation for malware detection and classification," *IEEE Transactions on Software Engineering*, 2024.

[241] D. E. García, N. DeCastro-García, and A. L. M. Castañeda, "An effectiveness analysis of transfer learning for the concept drift problem in malware detection," *Expert Systems with Applications*, vol. 212, p. 118724, 2023.

[242] S. Hou, X. Pan, C. C. Loy, Z. Wang, and D. Lin, "Learning a unified classifier incrementally via rebalancing," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 831–839.

[243] M. Welling, "Herding dynamical weights to learn," in *Proceedings of the 26th Annual International Conference on Machine Learning*, 2009, pp. 1121–1128.

[244] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[245] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 468–471. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903508

[246] VirusTotal, https://www.virustotal.com, , [Online; accessed 21-September-2023].

[247] M. Hurier, G. Suarez-Tangil, S. K. Dash, T. F. Bissyandé, Y. Le Traon, J. Klein, and L. Cavallaro, "Euphony: Harmonious unification of cacophonous anti-virus vendor labels for android malware," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 425–435.

[248] Y. Wu, S. Dou, D. Zou, W. Yang, W. Qiang, and H. Jin, "Contrastive learning for robust android malware familial classification," *IEEE Transactions on Dependable and Secure Computing*, 2022.

10

Appendix

10.1 Chapter 7

Hyper-parameter Sensitivity

Table 10.1: Summary of experimental results displaying the impact of varying mini batch sizes, learning rates, and weight decay settings on mean accuracy of model LwF.

Exp Index	Mini Batch Size	Learning Rate	Weight Decay	Mean Accuracy (%)
0	128	1×10^{-4}	1×10^{-4}	49.68
1	128	1×10^{-4}	1×10^{-3}	47.18
2	128	1×10^{-3}	1×10^{-4}	42.76
3	128	1×10^{-5}	1×10^{-4}	48.22
4	256	1×10^{-4}	1×10^{-4}	48.38

Table 10.2: Summary of experimental results displaying the impact of varying mini batch sizes, learning rates, and weight decay settings on mean accuracy of model SS-IL.

Exp Index	Mini Batch Size	Learning Rate	Weight Decay	Mean Accuracy (%)
5	128	1×10^{-5}	1×10^{-4}	54.58
6	128	1×10^{-4}	1×10^{-4}	38.81
7	128	1×10^{-3}	1×10^{-4}	26.67
8	128	1×10^{-5}	1×10^{-3}	54.70

During the preliminary study presented in Section 7.6.2.1, we conducted a comprehensive analysis of various hyper-parameters using the same dataset to ascertain their optimal combination for the models' performance. In this section, we provide detailed empirical experimental results and our findings from these investigations.

We initially used the Learning without Forgetting (LwF) to establish three fundamental hyper-parameters: mini-batch size, learning rate, and weight decay. As all the baseline approaches utilize the same backbone architecture (i.e., ResNet-18) and operate within the same incremental learning paradigm, these parameters were determined as the common optimal choices for other baseline approaches as well. However, due to the special strategy employed by SS-IL regarding the use of exemplars, we

conducted additional experiments specifically for this approach to identify the most suitable hyper-parameters tailored to its requirements. Tables 10.1 and 10.2 summarize the impact of varying mini-batch sizes, learning rates, and weight decay settings on the mean accuracy for the LwF and SS-IL models. These experiments highlight the sensitivity of these models to different parameter configurations. Specifically, the LwF model demonstrates relatively stable performance across various settings, whereas the SS-IL model exhibits a pronounced preference for lower learning rates, underlining its particular sensitivity to this parameter.

Table 10.3: Comparison of mean accuracy across different values of lambda for LwF with Exemplar and iCaRL methods.

LwF with Exemplar			iCaRL		
Exp Index	Lambda	Mean Accuracy (%)	Exp Index	Lambda	Mean Accuracy (%)
9	0.05	59.61	13	0.05	58.68
10	0.1	60.05	14	0.1	58.76
11	0.5	59.10	15	0.5	57.93
12	1	58.48	16	1	58.15

Table 10.4: Comparison of mean accuracy across different values of lambda for SS-IL and MM-TIML methods.

SS-IL			MM-TIML			
Exp Index	Lambda	Mean Accuracy (%)	Exp Index	Lambda 0	Lambda 1	Mean Accuracy (%)
17	0.05	55.19	21	0.1	0.1	66.32
18	0.1	55.18	22	0.5	0.1	65.86
19	0.5	55.72	23	0.1	0.5	66.01
20	1	54.58	24	0.5	0.5	65.94

Following the establishment of the three fundamental hyper-parameters, we engaged in further comparative experiments to fine-tune the parameter λ for each approach. λ refers to the trade-off weight in knowledge distillation techniques, impacting how previous knowledge is preserved while learning new information, which is critical in their respective loss functions. Note that the Fine-tuning and Full Retraining baselines do not include this parameter, as they do not utilize the knowledge distillation technique. Table 10.3 and 10.4 illustrate the impact of the parameter λ on the performance of different approaches. The results indicate that both LwF and iCaRL maintain relatively stable performance across a range of λ values, as shown in Table 10.3. Furthermore, as depicted in Table 10.4, the SS-IL and MM-TIML approaches demonstrate consistent performance across different settings of λ, with SS-IL particularly showing stable outcomes even with significant variations in λ. This consistency demonstrates the robustness of the λ parameterization of these models and particularly highlights the adaptability of SS-IL to different settings of the knowledge distillation parameter, in stark contrast to its sensitivity to the learning rate.

To conclude, while subtle changes in parameters resulted in corresponding shifts in performance, the overall performance of most of our TIML approaches remained relatively stable across different configurations. We selected the optimal parameter

combination for the final experiments conducted on the full dataset to ensure maximum efficacy. It is important to note that the scope of the empirical experiments conducted was far more extensive than what is detailed here. Only the most critical experiments that significantly influenced our understanding and development of the TIML framework are reported in this appendix.

Statistical Analysis

Table 10.5: Summary of experimental results displaying mean accuracy (%) across different methods and random seeds.

Methods	Seed 0	Seed 42	Seed 123	Seed 999	Seed 2023
LwF with Exemplar	59.58	60.05	59.13	58.85	59.42
iCaRL	57.96	58.76	58.24	57.39	58.33
SS-IL	55.74	55.72	55.61	55.58	55.94
MM-TIML	65.88	66.32	65.31	66.12	66.05

Table 10.6: Mean accuracy (%) and standard deviation across different methods over five random seeds.

Methods	Mean Accuracy (%)	Std Dev (%)
LwF with Exemplar	59.41	0.46
iCaRL	58.14	0.51
SS-IL	55.72	0.14
MM-TIML	65.94	0.38

Table 10.7: Paired t-Test results between MM-TIML and baseline methods. The results display the differences per seed, mean difference, standard deviation of differences, t-statistic, degrees of freedom (df), and p-value.

Comparison	Differences per Seed	Mean Difference (%)	Std Dev (%)	t-Statistic	df	p-Value
MM-TIML vs. LwF-w-E	[6.30, 6.27, 6.18, 7.27, 6.63]	6.53	0.45	32.68	4	< 0.0001
MM-TIML vs. iCaRL	[7.92, 7.56, 7.07, 8.73, 7.72]	7.80	0.61	28.69	4	< 0.0001
MM-TIML vs. SS-IL	[10.14, 10.60, 9.70, 10.54, 10.11]	10.22	0.37	62.43	4	< 0.0001

Following the hyper-parameter sensitivity investigation in the previous section, we perform a statistical analysis of our MM-TIML approach and three key baseline approaches using the same dataset and features under varying random seed conditions. This analysis uses the optimal hyper-parameter settings determined in the previous section and evaluates the methods across five different random seeds to ensure robustness and consistency in performance.

Table 10.5 shows the mean accuracy of the four methods for each random seed. The results demonstrate that MM-TIML consistently outperforms the other approaches across all five seeds. LwF with Exemplar, iCaRL, and SS-IL, on the other hand, exhibit lower mean accuracies, with SS-IL achieving the lowest values. The variability in performance across different seeds is relatively small, suggesting stable

model behavior. Table 10.6 summarizes the overall mean accuracy and standard deviation of the methods across the five random seeds. MM-TIML continues to exhibit the highest mean accuracy. The standard deviations are low across all models, indicating that the methods are relatively stable across different seeds. Table 10.7 presents the paired t-test results between MM-TIML and the baseline approaches (LwF with Exemplar, iCaRL, and SS-IL). The mean difference per seed is calculated along with the standard deviation, t-statistic, degrees of freedom, and p-values.

- The comparison between MM-TIML and LwF with Exemplar shows a mean difference of 6.53% in favor of MM-TIML, with a t-statistic of 32.68 and a highly significant p-value of less than 0.0001.
- For MM-TIML vs. iCaRL, the mean difference is 7.80%, with a t-statistic of 28.69, again demonstrating a highly significant result (p-value < 0.0001).
- Finally, MM-TIML vs. SS-IL yields the largest mean difference of 10.22%, with a t-statistic of 62.43 and a p-value less than 0.0001, strongly indicating that MM-TIML significantly outperforms SS-IL.

The statistical analysis confirms that MM-TIML significantly outperforms the other methods in terms of mean accuracy across different random seeds. The results are consistent and robust, as indicated by the low standard deviations and significant t-test results. The performance improvements of MM-TIML over the baseline methods are statistically significant, reinforcing the effectiveness of our proposed approach.