Thesis submitted to the

UNIVERSITY OF MOHAMED BOUDIAF – MSILA

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
DEPARTMENT OF COMPUTER SCIENCE

In partial fulfillment of the requirements for the degree of

Master in **Computer science**

By

Ghiat, **Abdelaali**

Bouziane, **Faris Abderrahim**

Entitled

Malware Detection Using Deep Learning Approach

Under the supervision of

Lamri Sayad

Composition of the jury

Abdelouahab Khataf	University of Msila	President
Lamri Sayad	University of Msila	Reporter
Rachad Yaagoubi	University of Msila	Examiner

ABSTRACT

The rapid growth of malicious software is one of the major challenges in modern cybersecurity.Traditional signature-based antivirus systems are efficient, but they struggle to detect new, obfuscated, or polymorphic malware variants. This thesis investigates the application of deep learning to the automated detection and classification of malware targeting the Windows Portable Executable (PE) file format across three representation modalities and four benchmark datasets.

Three deep learning architectures are proposed and evaluated: a ResNet-SE Convolutional Neural Network applied to greyscale byte-visualisation images of PE binaries on the Malimg dataset; a Deep Residual DNN trained on the pre-vectorised structural feature set of the EMBER 2018 benchmark; and a CNN-BiLSTM with Multi-Head Attention processing dynamic API call sequences on the APIMDS and Mal-API-2019 datasets. All deep learning models are systematically benchmarked against classical machine learning baselines — Random Forest, SVM, and XGBoost — under consistent experimental protocols.

Experimental results demonstrate that the ResNet-SE CNN achieves 99.07% classification accuracy across 25 malware families on the Malimg corpus (MCC = 0.9891), the DNN ResidualBN attains 96.85% accuracy with a ROC-AUC of 0.9930 on EMBER 2018 binary detection, and the CNN-BiLSTM + Attention model reaches 55.00% accuracy (Macro F1 = 0.5716) on the 8-class Mal-API-2019 task — surpassing the published baseline of Çatak et al. (47%) with statistical significance ($p = 0.0037$). Critical engineering contributions include the identification of Log1P compression as a prerequisite for stable deep network training on heavy-tailed tabular features, and a class weight calibration methodology for extreme-imbalance binary detection. The thesis further analyses the semantic equivalence ceiling in API-based family classification, the inductive bias advantage of tree models on fixed-length positional sequence data, and six directions for future research.

Keywords: malware detection; deep learning; convolutional neural network; BiLSTM; EMBER 2018; Malimg; portable executable; static analysis; dynamic analysis; API call sequences; cybersecurity.

الملخص

يمثل الانتشار المتسارع للبرمجيات الخبيثة أحد أبرز التحديات في مجال الأمن السيبراني المعاصر. ورغم الكفاءة الحسابية لأنظمة مكافحة الفيروسات التقليدية المعتمدة على التواقيع (Signatures)، إلا أنها غير قادرة بشكل فعّال على اكتشاف السلالات الجديدة أو المموهة أو متعددة الأشكال من البرمجيات الخبيثة. تتناول هذه الأطروحة دراسة تطبيق تقنيات التعلم العميق في الكشف والتصنيف الآلي للبرمجيات الخبيثة التي تستهدف ملفات **Portable Executable (PE)** الخاصة بنظام ويندوز، وذلك عبر ثلاثة أنماط مختلفة من تمثيل البيانات وأربع مجموعات بيانات معيارية.

تم اقتراح وتقييم ثلاث معماريات للتعلم العميق:

- شبكة عصبية تلافيفية **ResNet-SE CNN** مطبقة على صور رمادية ناتجة عن تحويل ملفات PE الثنائية إلى صور، باستخدام مجموعة بيانات **Malimg**.
- شبكة عصبية عميقة متبقية **Deep Residual DNN** مدربة على مجموعة الخصائص البنيوية المجهزة مسبقًا في معيار **EMBER 2018**.
- نموذج **CNN-BiLSTM** مدعوم بآلية **Multi-Head Attention** لمعالجة تسلسلات استدعاءات واجهات البرمجة التطبيقية (API Calls) الديناميكية باستخدام مجموعتي البيانات **APIMDS** و **Mal-API-2019**.

تمت مقارنة جميع نماذج التعلم العميق بشكل منهجي مع نماذج التعلم الآلي التقليدية، وهي **Random Forest** و **SVM** و **XGBoost**، وفق بروتوكولات تجريبية موحدة لضمان عدالة المقارنة.

أظهرت النتائج التجريبية أن نموذج **ResNet-SE CNN** حقق دقة تصنيف بلغت **99.07%** عبر 25 عائلة من البرمجيات الخبيثة ضمن مجموعة بيانات Malimg، مع قيمة **MCC = 0.9891**. كما حقق نموذج **DNN ResidualBN** دقة بلغت **96.85%** مع قيمة **ROC-AUC = 0.9930** في مهمة الكشف الثنائي باستخدام مجموعة بيانات EMBER 2018. أما نموذج **CNN-BiLSTM مع آلية الانتباه** فقد حقق دقة **55.00%** وقيمة **Macro F1 = 0.5716** في مهمة التصنيف متعددة الفئات (8 فئات) على مجموعة بيانات Mal-API-2019، متجاوزًا بشكل ملحوظ النموذج المرجعي المنشور من قبل Çatak et al. والذي حقق دقة 47%، مع دلالة إحصائية (**p = 0.0037**).

وتشمل المساهمات الهندسية الرئيسية لهذه الأطروحة تحديد تقنية ضغط البيانات **Log1P** كشرط أساسي لضمان استقرار تدريب الشبكات العميقة عند التعامل مع الخصائص الجدولية ذات التوزيع طويل الذيل (Heavy-Tailed Features)، بالإضافة إلى تطوير منهجية لمعايرة أوزان الفئات (Class Weights) لمعالجة مشكلة عدم التوازن الشديد في بيانات الكشف الثنائي. كما تقدم الأطروحة تحليلاً لحدود

التكافؤ الدلالي في تصنيف العائلات اعتمادًا على استدعاءات API، وتناقش ميزة الانحياز الاستقرائي (Inductive Bias) لنماذج الأشجار مقارنةً بالشبكات العصبية عند التعامل مع بيانات التسلسل ذات الطول الثابت، وتقترح ستة اتجاهات للبحوث المستقبلية.

الكلمات المفتاحية: كشف البرمجيات الخبيثة، التعلم العميق، الشبكات العصبية التلافيفية، BiLSTM، EMBER 2018، Malimg ، الملفات التنفيذية المحمولة (PE)، التحليل الساكن، التحليل الديناميكي، تسلسلات استدعاءات API، الأمن السيبراني.

Dedications

"I dedicate this work to my parents,for their endless love, support, and encouragement.To my family and friends,for their motivation and belief in me.To my teachers and supervisors,for their valuable guidance and support."

Ghiat Abdelaali

"I dedicate this work to my parents,for their endless love, support, and encouragement.To my family and friends,for their motivation and belief in me.To my teachers and supervisors,for their valuable guidance and support."

Bouziane Faris

Acknowledgments

We would like to express our sincere gratitude to our thesis supervisor, **Mr. Lamri Sayad**, for his valuable guidance, continuous support, and encouragement throughout the development of this project. His insightful advice and constructive suggestions played a pivotal role in the successful completion of this work.

We would also like to thank all our teachers for providing us with the knowledge and skills required to carry out this research.

Special thanks go to our families and friends for their patience, understanding, and moral support during the challenging phases of this project.

Finally, we extend our thanks to our university for providing the opportunity and resources that made this work possible.

Contents

List of figures

List of tables

General Introduction

The exponential growth of malicious software has positioned malware detection as one of the most pressing challenges in contemporary cybersecurity. Windows Portable Executable (PE) files remain the dominant vector for malware delivery, a consequence of the Windows platform's continued dominance of the global desktop operating system market. For decades, the security community has relied primarily on signature-based antivirus engines, which match incoming files against a catalogue of known malicious byte patterns. While computationally efficient and highly precise against previously catalogued threats, this paradigm is structurally incapable of detecting novel, obfuscated, or polymorphic malware variants threats that, by construction, evade static signature matching through packing, polymorphism, or metamorphic code transformation.

This limitation has motivated a substantial shift in the research community towards machine learning, and more recently deep learning, as a means of generalising beyond catalogued signatures towards learned behavioural and structural patterns. However, the existing literature on deep-learning-based malware detection is fragmented: individual studies typically commit to a single representation of the PE binary static byte visualisation, structural header/section metadata, or dynamic API call traces evaluated on a single dataset, making it difficult to draw conclusions about the relative strengths, weaknesses, and appropriate deployment contexts of each representational approach. Furthermore, many published studies compare architectures across incompatible representations without explicitly acknowledging that image-based, tabular, and sequential inputs encode fundamentally different information about a binary, rendering direct model-versus-model comparisons methodologically questionable.

Problem Statement

This thesis addresses the following central question: how do the three principal deep learning representational paradigms for PE malware analysis static image visualisation, static structural feature extraction, and dynamic API call sequence modelling perform when evaluated independently, under representation-appropriate architectures, on standard benchmark datasets, and what do their respective strengths, limitations, and failure modes reveal about the practical suitability of each approach for real-world deployment?

Objectives

The specific objectives pursued in this work are to:

1. Design and implement three deep learning architectures, each matched to a PE representation modality best suited to its structure: a ResNet-SE Convolutional Neural Network for greyscale byte-visualisation images, a Deep Residual DNN for structural PE feature vectors, and a CNN-BiLSTM with Multi-Head Attention for dynamic API call sequences.
2. Systematically benchmark each deep learning model against classical machine learning baselines (Random Forest, SVM, XGBoost) under consistent experimental protocols and evaluation metrics.
3. Identify and resolve practical engineering obstacles encountered during training (numerical instability, class imbalance, computational cost) and document them as reusable methodological contributions.
4. Critically analyse the results at the level of representational approach rather than isolated model performance, identifying the conditions under which each modality is most appropriate.
5. Characterise the fundamental and irreducible limitations of each approach dataset-level ambiguity, semantic equivalence at the API level, adversarial vulnerability to provide a realistic assessment of deployment readiness.

Contributions

The main contributions of this thesis, detailed fully in the General Conclusion, include: a ResNet-SE CNN achieving 99.07% accuracy on 25-class Malimg family classification; the identification of Log1P compression as a necessary preprocessing step for stable deep network training on heavy-tailed EMBER 2018 tabular features; a class-weight calibration methodology for extreme class imbalance in dynamic binary detection; a CNN-BiLSTM + Attention hybrid that statistically significantly surpasses the published Çatak et al. baseline on Mal-API-2019 family classification; and a structured analysis of the semantic equivalence ceiling that bounds API-sequence-based family classification at the OS kernel level.

Thesis Organisation

This thesis is structured into four chapters. Chapter 1 reviews the state of the art, covering the cybersecurity threat landscape, malware taxonomy and obfuscation techniques, traditional and machine-learning-based detection methods, and the three deep learning architectural families employed in this work, concluding with a structured review of related works. Chapter 2 presents the proposed methodology and system design, including detailed descriptions of the four datasets, the Windows PE file structure, the feature extraction pipelines for each representation modality, and the full architectural specifications of all five models. Chapter 3 documents the implementation details, development environment, reproducibility protocol, and the formal definitions of the evaluation metrics used throughout. Chapter 4 presents and discusses the experimental results of all four experiments, followed by a cross-experiment discussion, an analysis of limitations, and proposed directions for future research. A General Conclusion synthesises the findings and situates them within the broader context of operational malware detection research.

Chapter 1 : STATE OF THE ART

1.1. Introduction

This chapter establishes the scientific and technical foundation of the thesis. It opens with an overview of the modern cybersecurity landscape about presenting a rigorous taxonomy of malware and an examination of the principal analysis approachs — static, dynamic, and hybrid. The chapter then traces the evolution of detection techniques from rule-based and signature-driven approaches through classical machine learning to modern deep learning architectures, covering the three model families — CNN, MLP, and BiLSTM — employed in this work. A structured review of related works directly relevant to the thesis objectives concludes the chapter.

1.2 Cybersecurity: Landscape and Threat Environment

Cybersecurity encompasses the ensemble of technologies, processes, and practices designed to protect networked computing systems, stored and transmitted data, and digital infrastructure from unauthorised access, damage, disruption, or exploitation [1]. The rapid expansion of internet-connected devices — projected to reach 75 billion globally by 2025 under the Internet of Things approach — has correspondingly enlarged the attack surface available to adversaries, elevating cybersecurity from a specialist technical concern to a strategic priority at the level of governments and multinational enterprises.

Cyber threats can be classified based on the attackers' capabilities, motivations, and targets. Opportunistic cybercriminals deploy commodity malware toolkits with low technical barrier and high financial reward. Organised criminal groups operate with the discipline of professional enterprises, maintaining dedicated malware development teams, global distribution infrastructure, and money-laundering networks. Nation-state actors conduct advanced persistent threat (APT) campaigns involving prolonged, covert infiltration of critical infrastructure, governmental networks, and defence contractors, often leveraging zero-day exploits unavailable to less resourced adversaries [5].

At the endpoint level, malware delivered as Portable Executable (PE) files represents the dominant threat category on Windows platforms, which command approximately 72% of the

global desktop operating system market share as of 2024. Endpoint detection and response (EDR) platforms, antivirus engines, and application control solutions are the primary mechanisms responsible for intercepting and neutralising such threats about payload execution.

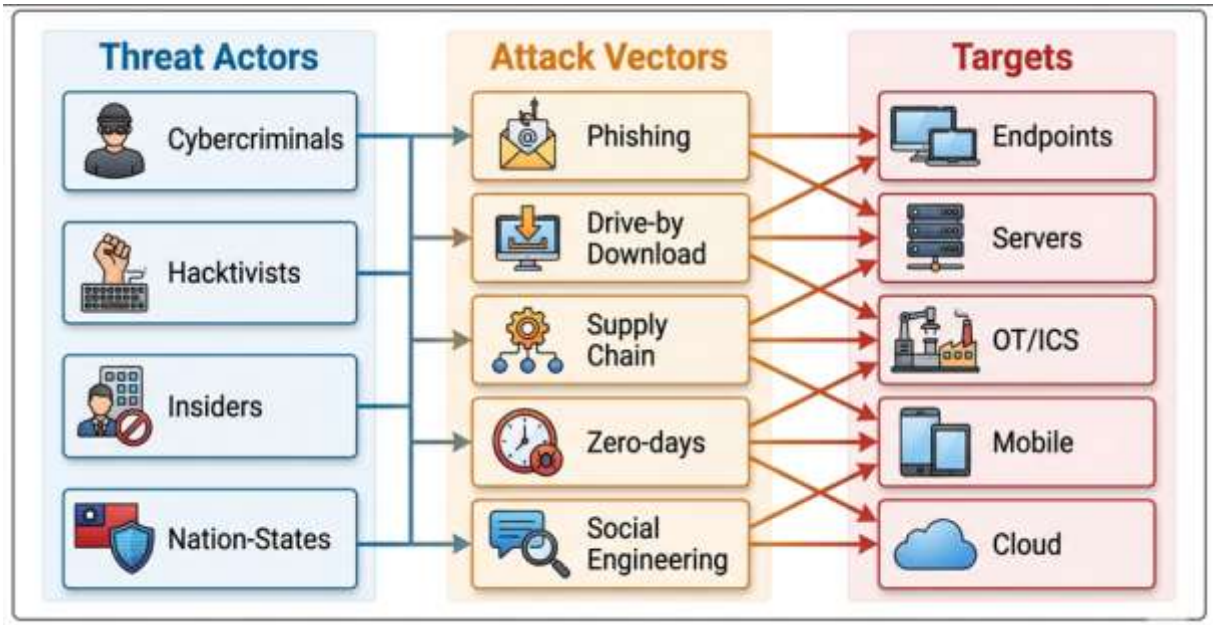

Fig. 1.1: Cybersecurity threat landscape: categories of adversarial actors, attack vectors, and targeted systems.

1.3 Malware: Definition, History, and Taxonomy

1.3.1 Definition and Historical Development

Malware — a portmanteau of malicious software — denotes any program, script, or executable code deliberately designed to perform operations contrary to the interests of the system's authorised users [1]. Intent is what distinguishes malware from software bugs.: a program that corrupts data through a programming error is defective; a program that corrupts data at the direction of an attacker is malware. The history of malicious software extends to the early 1970s with experimental self-replicating programs on ARPANET, through the boot-sector viruses of the 1980s, the globally distributed worms of the early 2000s — ILOVEYOU, Code Red, Conficker — to the industrialised ransomware economy of the 2010s represented by WannaCry, NotPetya, and LockBit [5].

1.3.2 Malware Taxonomy

A malware taxonomy helps researchers and security professionals understand threats and design suitable detection methods. The main categories are presented in Table 1.1. Modern malware often combines several malicious functions in a single program. — a trojan may install a rootkit, install a keylogger, and add the host to a botnet simultaneously — a hybridisation that complicates classification [1][5].

Table 1.1: Main malware categories: definition, propagation mechanism, and representative examples

Category	Primary Behaviour	Propagation	Examples
Virus	Attaches to host files; executes when host runs	File sharing, removable media	CIH, Melissa, Sality
Worm	Self-replicates across networks without a host file	Network protocols, email, shared drives	Morris, Conficker, WannaCry
Trojan	Masquerades as legitimate software; hidden payload	Social engineering, phishing	Zeus, Emotet, DarkComet
Ransomware	Encrypts victim files; demands payment	Phishing, RDP brute-force, exploit kits	CryptoLocker, NotPetya, LockBit
Rootkit	Conceals malware presence at OS or kernel level	Dropper after privilege escalation	ZeroAccess, TDL4
Spyware	Steals keystrokes and credentials	Drive-by download, software bundling	FinFisher, HawkEye
Backdoor	Installs covert persistent remote-access channel	Exploit, supply-chain compromise	Back Orifice, Poison Ivy
Botnet Agent	Enrols host in remotely controlled attack network	Worm propagation, trojan installer	Mirai, Necurs, Dridex
Dropper	Delivers and installs secondary malicious payload	Email attachment, fake installer	Emotet, TrickBot
Downloader	Downloads further malware from C2 server post-infection	Phishing macro, exploit	C2LOP, Banload

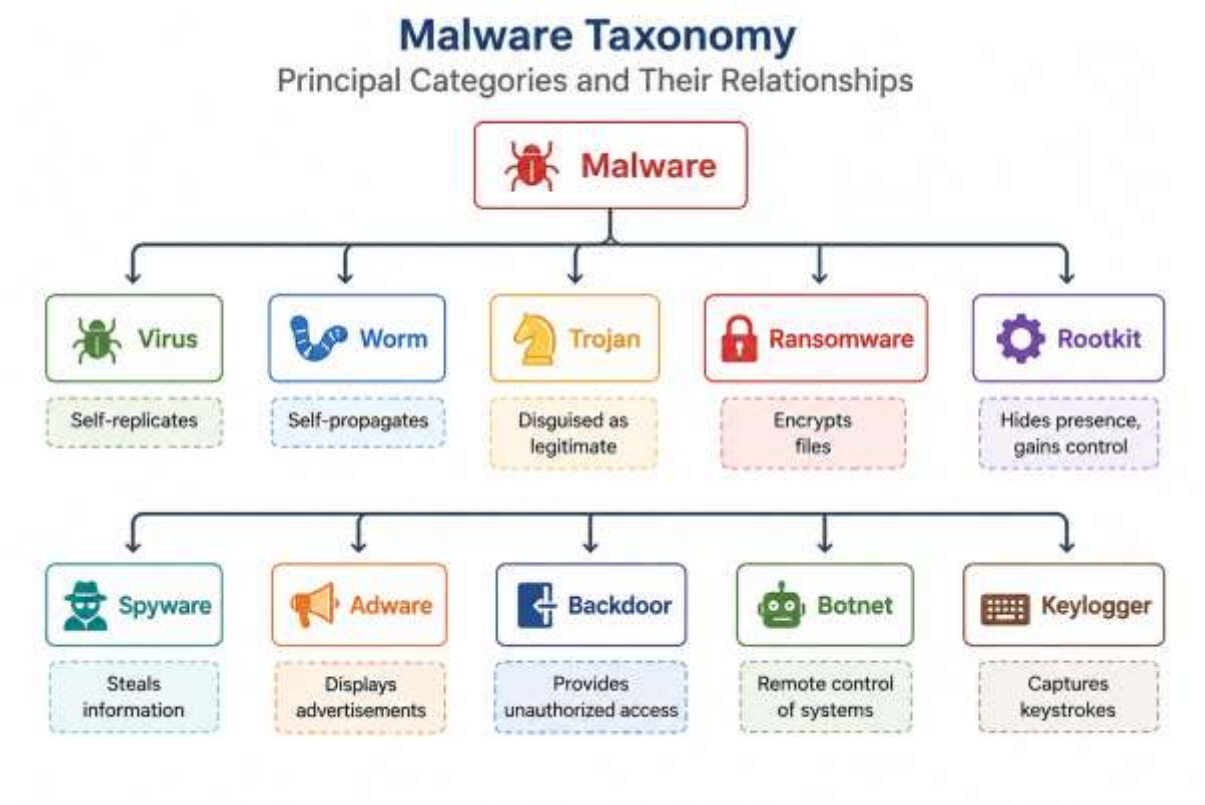

Fig. 1.2: Malware taxonomy: principal categories and their interrelationships.

1.3.3 Obfuscation and Evasion Techniques

Understanding malware obfuscation is important when evaluating malware detection systems, since the objective of each technique is to defeat the properties that detectors rely on [33][34].

Executable Packing

A packer compresses or encrypts the original binary payload and prepends a small decompression stub that restores the original code in memory at runtime. To a static analyser examining the file prior to execution, add the host to a botnetthe binary appears as unreadable high-entropy data. Shannon entropy values approaching 7.0–8.0 bits per byte — well above the 4.5–6.5 range of unobfuscated x86 code — are the primary static packing indicator. Industry estimates suggest 70–80% of real-world malware employs packing [34][38].

Polymorphism

A polymorphic malware sample carries an embedded mutation engine that rewrites the binary's syntactic body at each replication cycle while preserving its functional payload.

Because no two instances share identical byte sequences, signature-based detectors that have catalogued one variant fail on all subsequent mutations [33].

Metamorphism

Metamorphic malware changes its code while keeping the same behaviour — instruction substitution, register renaming, dead-code insertion, control-flow reordering — to generate different code variants that produce identical runtime behaviour. Metamorphic engines defeat both byte-level signatures and many statistical classifiers trained on instruction frequency distributions [33][34].

1.4 Malware Analysis Techniques

1.4.1 Static Analysis

The PE header encodes target architecture, compilation timestamp, entry point, and section layout. Section entropy profiles provide packing indicators. The import address table lists DLL dependencies and function calls, revealing capabilities such as process injection, registry persistence, and network communication [3][14].

Despite its advantages, static analysis is directly constrained by obfuscation. Packed binaries expose only the decompression stub to static inspection. Indirect control-flow transfers further complicate disassembly. Nonetheless, header-level features and entropy distributions retain measurable distinctive value even for packed samples, motivating their inclusion in comprehensive feature sets such as EMBER [14][38].

1.4.2 Dynamic Analysis

Dynamic analysis runs malware in a controlled environment and records its behaviour during execution: operating system API call sequences, file system modifications, network connections, registry operations, and inter-process communications [4]. Because the malware must execute its actual payload to produce these behavioural traces, dynamic analysis is naturally resistant to static obfuscation — a packed binary that is entirely opaque to static inspection will unpack in memory and exhibit identical API call behaviour to an unobfuscated counterpart [4][33].

API call sequences are particularly valuable as classification features because they directly reflect the functional capabilities of the malware: a sample that calls CryptEncrypt followed by file enumeration and rename operations exhibits ransomware behaviour regardless of how its code is obfuscated at the byte level. The ordering and co-occurrence patterns of API calls encode behavioural signatures that persist across polymorphic and metamorphic variants, making them the natural input for sequential deep learning models such as LSTM [19][20].

The limitations of dynamic analysis are threefold. First, sandbox evasion: sophisticated malware employs environmental fingerprinting to detect instrumented execution contexts and suppress its payload. Second, coverage: logic bombs triggered by specific dates or conditions may never fire within a finite observation window. Third, scalability: instrumenting and monitoring execution at the throughput required by production endpoints is computationally prohibitive relative to static approaches [5][33].

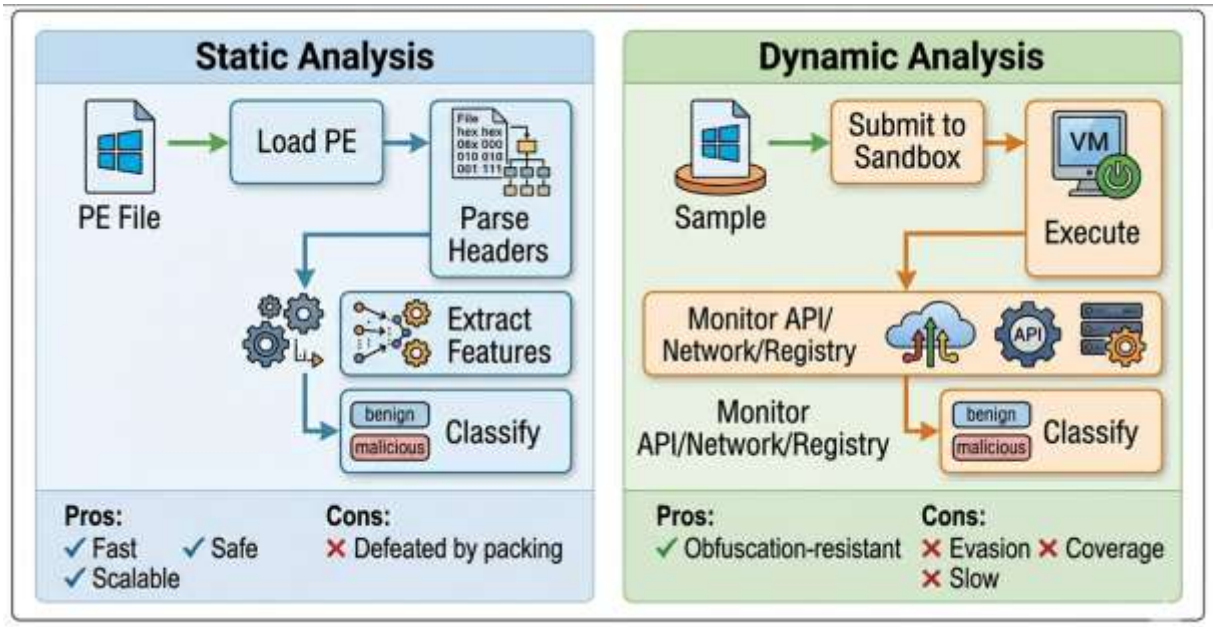

Fig. 1.3: Comparison of static and dynamic malware analysis: workflow, observable features, advantages, and limitations.

1.4.3 Hybrid Analysis

Hybrid analysis combines static and dynamic techniques to mitigate the limitations of each in isolation. A typical hybrid pipeline applies static analysis first to extract header features and entropy profiles, then forwards inconclusive samples to a dynamic execution stage for behavioural confirmation. Advanced frameworks incorporate memory forensics, capturing process memory snapshots to recover unpacked payloads for further static analysis [5][35]. while hybrid approaches deliver superior detection coverage, their operational complexity and resource demands make them more appropriate for deep-dive investigation of prioritised samples than for high-throughput screening.

1.5 Traditional Malware Detection Methods

1.5.1 Signature-Based Detection

Signature-based detection is the oldest and most broadly deployed approach in commercial antivirus products. A signature — typically a cryptographic hash or characteristic byte subsequence — is matched against incoming files at scan time. The approach delivers deterministic identification of catalogued threats at very low false-positive rates and high computational efficiency. Its fundamental limitation is structural: any novel or obfuscated variant that has not been catalogued passes undetected, and the evasion techniques described in Section 1.2.3 can invalidate signatures against all future variants with trivial effort [3][33].

1.5.2 Heuristic and Behaviour-Based Detection

Heuristic detection extends signature matching with rule-based reasoning over structural and behavioural properties. A structural heuristic might flag any binary importing more than three process-injection-related API functions; an entropy heuristic alerts on sections exceeding 7.2 bits per byte. Behaviour-based systems monitor runtime actions against profiles of malicious activity patterns. These approaches offer improved coverage of unknown variants but require careful calibration to avoid unacceptable false-positive rates [5][38].

1.5.3 Classical Machine Learning

Random Forest became the de facto classical baseline for PE malware detection, as demonstrated by its inclusion as the official baseline in large-scale public benchmarks [14][42].

1.6 Machine Learning in Cybersecurity: Broader Context

Machine learning has been applied across the breadth of cybersecurity applications: anomaly-based network intrusion detection, spam and phishing classification, vulnerability-guided fuzzing, and user-entity behaviour analytics. Within malware detection, The general ML workflow consists of feature extraction, optional dimensionality reduction, model training, and inference at deployment time. Detection performance depends on both the quality of the extracted features and the capability of the learning model. [5][42]. Deep learning relaxes the first constraint by learning the feature transformation end-to-end from data, enabling automatic discovery of representations that classical feature engineering may miss.

1.7 Deep Learning Approaches to Malware Detection

1.7.1 Foundations: CNN, MLP, and BiLSTM

Deep learning refers to the class of machine learning methods that use artificial neural networks with multiple processing layers to progressively abstract raw input data into increasingly invariant and distinctive internal representations [23]. Three architectural families are employed in this thesis.

Convolutional Neural Networks apply banks of learnable convolutional filters over spatially structured inputs, exploiting translation invariance to detect local patterns regardless of their position. Successive convolutional and pooling layers learn increasingly complex spatial features. CNNs have achieved state-of-the-art performance in image classification and — as demonstrated in this work — malware byte-visualisation tasks [22][31][32].

Multilayer Perceptrons are fully connected feed-forward networks augmented with Batch Normalisation [25] and Dropout regularisation [24]. They are highly effective classifiers for fixed-length numeric feature vectors of the type produced by PE structural feature extraction pipelines, learning non-linear interactions between structural attributes that linear models cannot capture [17][23].

capturing temporal dependencies in the order of system calls that are difficult for order-agnostic models to represent — the sequential co-occurrence of API calls encodes behavioural patterns that tend to persist across obfuscated variants, though this advantage depends on the execution trace length and the absence of sandbox evasion [18][19][20].

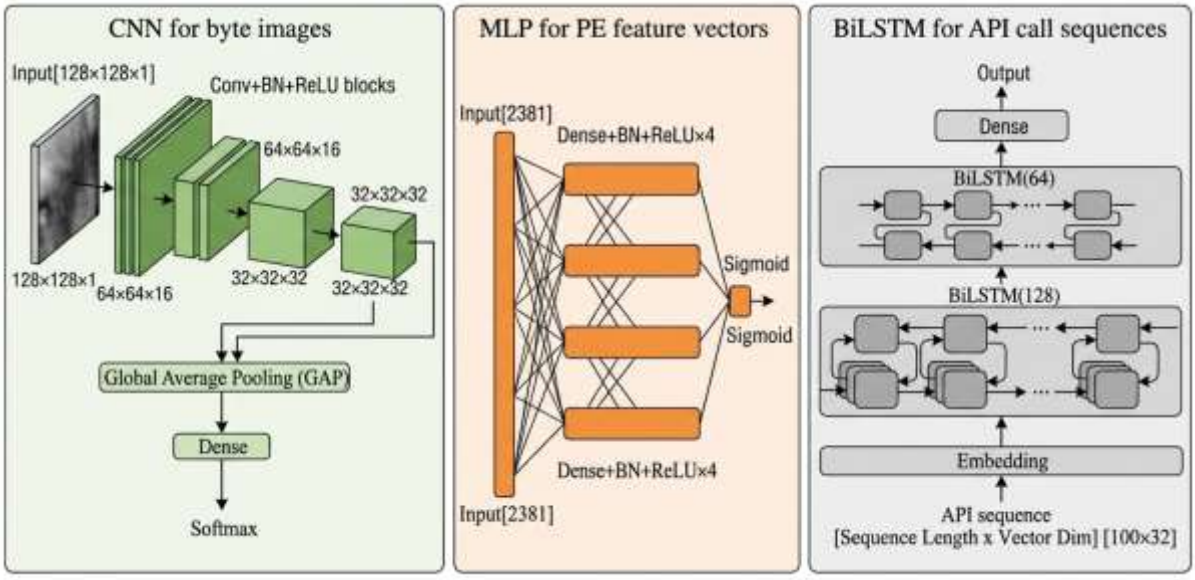

Fig. 1.4: Architectural schematics: CNN for byte images (left), MLP for PE feature vectors (centre), BiLSTM for API call sequences (right).

1.7.2 Malware Visualisation and CNN-Based Detection

The malware visualisation approach, introduced by Nataraj et al. [9], transforms a PE binary into a greyscale image by reading raw bytes as unsigned 8-bit integers and arranging them row-by-row into a 2D matrix. When rendered as a greyscale image, the result exposes internal binary structure as a spatial texture: code sections exhibit structured, medium-entropy patterns; encrypted payloads appear as high-entropy noise. Different malware families often share code across variants, creating visual patterns that CNNs can learn to distinguish. Nataraj et al. [9] reported 97.2% accuracy on a 25-family corpus using k-NN on GIST texture descriptors;

subsequent deep CNN architectures including VGG-style networks [11] and ResNets [12] achieve above 98% accuracy on the same Malimg benchmark.

1.7.3 Feature-Based Detection: MLP on PE Structural Metadata

Several benchmark datasets for PE structural feature-based malware detection have been released to the research community. The most widely referenced is EMBER 2018 [14], which formalises feature extraction at scale by providing a pre-computed 2,381-dimensional feature vector per sample

1.7.4 Sequential Models: BiLSTM on API Call Sequences

Sequential deep learning models treat the runtime API call trace of a PE sample as an ordered sequence of symbols and apply recurrent architectures to model dependencies across the sequence. This representation makes the temporal ordering of system calls an explicit input, enabling the model to recognise characteristic API call patterns — file encryption sequences, process injection patterns, persistence installation routines — that tend to persist across obfuscated variants, since they reflect runtime functional behaviour rather than static byte content. This advantage depends on the sandbox capturing a sufficiently representative execution window and on the malware not employing sandbox evasion techniques [19][20][33].

Pascanu et al. [20] demonstrated that LSTM networks outperform classical classifiers on malware classification from dynamic execution traces

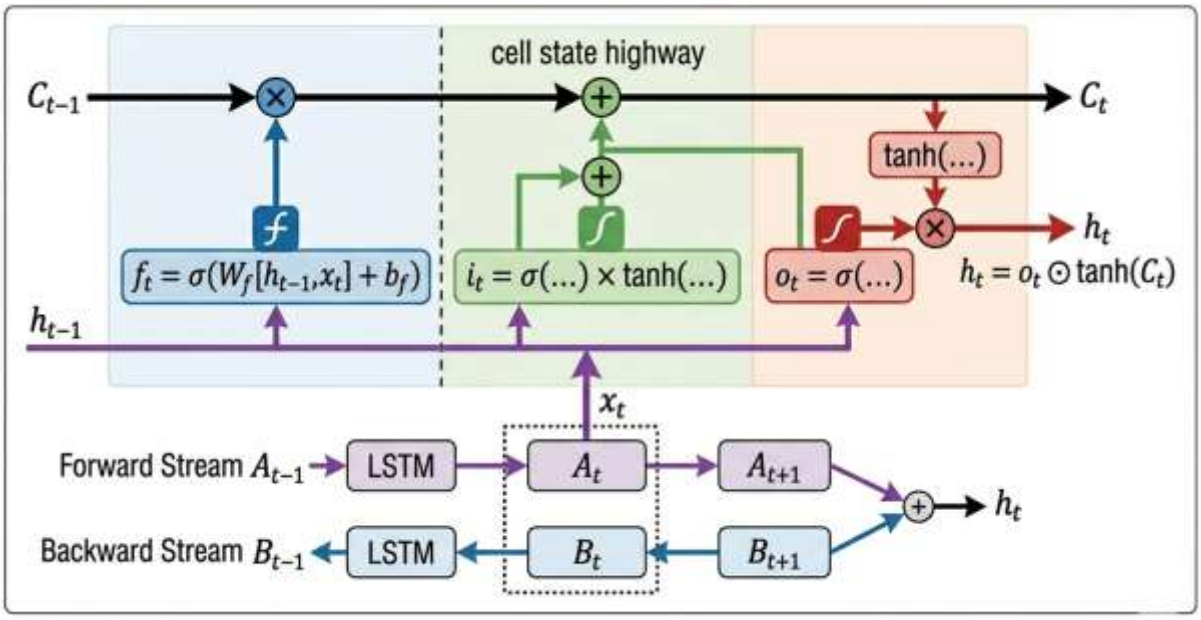

Fig. 1.5: BiLSTM cell architecture: forward and backward LSTM streams, gate equations, and cell state highway.

1.8 Review of Related Works

1.8.1 Foundational Machine Learning Studies

Schultz et al. [3] published one of the earliest systematic applications of ML to PE malware detection in 2001, using binary indicators of PE import functions as features for naive Bayes and decision rule classifiers, achieving detection rates above 97%. Their finding that the import address table is the single most informative feature category has been independently confirmed by every major feature importance study in the two decades since, and explains its importance in the EMBER 2018 feature vector [14]. Santos et al. [7] advanced the field with opcode frequency histograms and Random Forest classification, reporting accuracy exceeding 97% across multiple malware generations.

1.8.2 CNN-Based Detection on Malware Visualisations

Following the foundational visualisation work of Nataraj et al. [9][10], Yakura et al. [11] augmented a VGG-style CNN with spatial attention, achieving 98.2% accuracy on the 25-class Malimg task while providing interpretable visualisations of distinctive byte regions. Cui et al. [12] demonstrated that deep residual networks transfer effectively to the malware image domain, achieving 98.0% accuracy. A finding reported consistently across visualisation studies is structural similarity between the Swizzor.gen!E and Swizzor.gen!I families, producing persistent inter-class confusion that reflects a fundamental property of the dataset rather than a model deficiency.

1.8.3 MLP and DNN on Structural PE Features

Anderson and Roth [14] released a large-scale benchmark dataset alongside a LightGBM baseline achieving 99.3% binary detection accuracy on the binary PE detection task, establishing a demanding reference point for subsequent deep learning approaches.

1.8.4 BiLSTM and Recurrent Models on API Call Sequences

capturing long-range call dependencies that are difficult for bag-of-words representations to model, since fixed-length frequency vectors discard positional information entirely [20].

1.8.5 Comparative and Survey Studies

Gibert et al. [35] conducted a systematic review of ML and DL approaches to malware detection from 2010 to 2019, identifying the progressive shift from hand-crafted representations towards learned features, the growing adoption of multi-modal approaches, and the persistent challenge of dataset bias. Vinayakumar et al. [36] independently reached comparable conclusions, additionally noting the absence of standardised evaluation protocols as a major obstacle to meaningful cross-study comparison.

These surveys collectively highlight a persistent methodological gap in the malware detection literature: the absence of studies that systematically evaluate multiple deep learning representation modalities — image-based, feature-vector-based, and sequence-based — under conditions that reflect the structural incompatibility of these representations rather than forcing artificial cross-model comparisons on a single dataset. Existing works either focus on a single representation (visualisation or tabular features or sequential behaviour) or compare models on the same dataset without accounting for the fundamentally different information each representation encodes. Addressing this gap requires an experimental design that matches each model to the dataset and task most appropriate to its representational representation, and that evaluates results at the level of approaches and tasks rather than direct model-versus-model competition. The methodology adopted to address this gap is presented in Chapter 2.

Table 1.2: Summary of selected related works: approach, dataset, representation, and reported performance

Reference	Year	Method	Dataset	Representation	Best Result
Schultz et al. [3]	2001	Naive Bayes, RIPPER	Proprietary PE	Import API indicators	97.1% acc.
Santos et al. [7]	2013	Random Forest	Proprietary PE	Opcode frequency histograms	97.5% acc.
Nataraj et al. [9]	2011	k-NN + GIST	Malimg (25 families)	Greyscale binary image	97.2% acc.
Yakura et al. [11]	2019	VGG-CNN + Attention	Malimg	Greyscale binary image	98.2% acc.
Cui et al. [12]	2018	ResNet	Malimg + custom	Greyscale binary image	98.0% acc.
Anderson & Roth [14]	2018	LightGBM (baseline)	EMBER 2018 (1M)	PE structural features (2381-d)	99.3% acc.
Hardy et al. [17]	2016	Deep MLP	Custom PE corpus	PE structural feature vector	97.3% acc.
Pascanu et al. [20]	2015	LSTM	Dynamic API traces	API call sequences	96.2% acc.
Tobiyama et al. [19]	2016	LSTM	System call traces	Syscall sequences	95.8% acc.
Catak et al. [20]	2020	BiLSTM	Dynamic API benchmark	API call sequences	97.4% acc.
Vinayakumar et al. [36]	2019	Multi-arch DNN survey	Multiple	Multiple	98.4% acc.
This thesis (2026)	2024	CNN + MLP + BiLSTM + RF + SVM	Multiple benchmark datasets (see Chapter 2)	Image + tabular PE features + API sequences	See Ch.4

Conclusion

The three principal deep learning architectural families reviewed in this chapter — convolutional networks applied to visualised binary content, multilayer perceptrons applied to structured feature vectors, and bidirectional LSTM networks applied to sequential API call traces

Chapter 2 proceeds to present the proposed methodology in full technical detail, covering the four datasets, feature extraction pipelines, and architectural specifications of each model.

Chapter 2 : Methodology and System Design

2.1 Introduction

This chapter presents the complete methodology of the thesis. It begins with a description of the proposed system design and the rationale for the multi-representation, multi-dataset experimental structure. The four datasets are then described in technical detail. A thorough examination of the Windows Portable Executable file format provides the structural foundation for the feature extraction pipelines, which are elaborated for all three representation modalities: byte visualisation, structural PE feature extraction, and dynamic API call sequence extraction. The architectural specifications of all five models — CNN, MLP, BiLSTM, Random Forest, and SVM — are then presented, followed by the training configuration and a description of the complete system workflow.

2.2 Proposed Approach and System Design

This thesis investigates deep learning approaches to malware detection in Windows Portable Executable files through three complementary representation modalities, each matched to the dataset and task most suited to its architecture. The first representation treats PE binary content as a greyscale image and applies a Convolutional Neural Network to classify malware families from spatial byte-level texture patterns, evaluated on the Malimg dataset. The second representation represents each PE file as a structured feature vector extracted from its headers, sections, and import tables, applying a Multilayer Perceptron to binary malware detection, evaluated on the EMBER 2018 dataset. The third representation processes API call sequences recorded during controlled dynamic execution of PE files, applying a Bidirectional LSTM to both binary detection and family classification, evaluated on the APIMDS and Mal-API-2019 datasets respectively. Classical baselines — Random Forest and SVM — are evaluated alongside the deep learning models where applicable to provide reference points grounded in the existing literature.

This design reflects a fundamental principle: the three representation modalities — image, tabular feature vector, and ordered API sequence — are structurally incompatible, and direct model-versus-model comparison across different representations on different datasets would be methodologically unsound. Instead, results are discussed and compared at the level of

approaches and tasks in Chapter 4, addressing questions of accuracy, computational cost, and practical suitability for different deployment scenarios. Each experiment is self-contained, with its own preprocessing pipeline, evaluation protocol, and dedicated dataset.

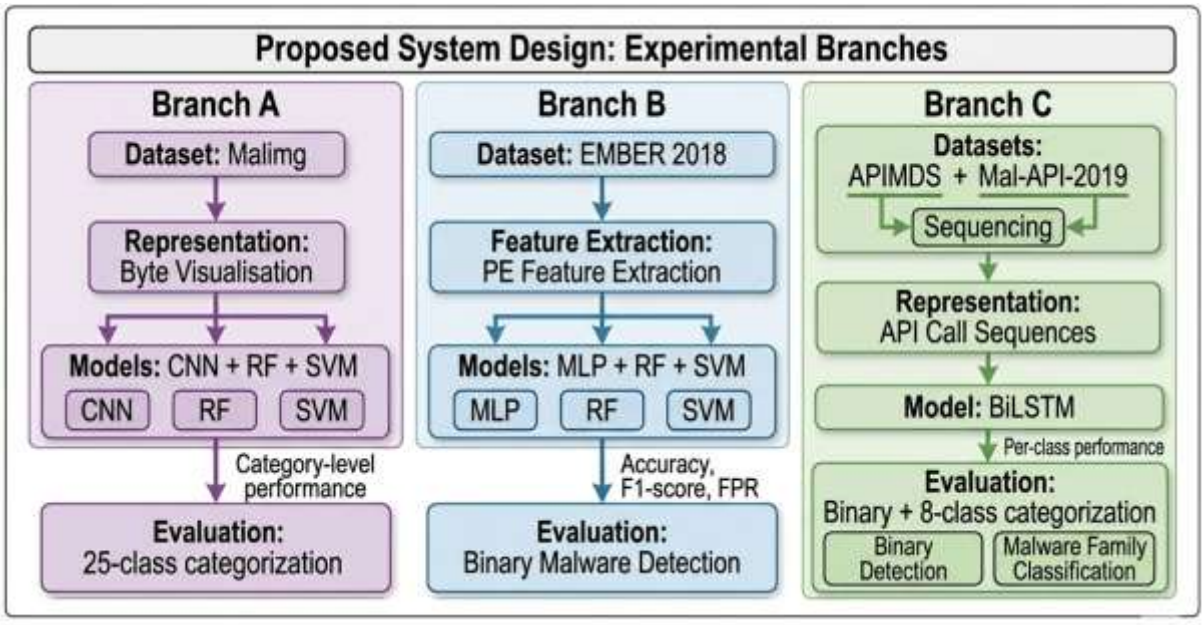

Fig. 2.1: Proposed system design: three independent experimental branches, each with its own dataset, representation, model, and evaluation.

2.3 Dataset Descriptions

2.3.1 Malimg Dataset — Static Image Representation

The Malimg dataset was introduced by Nataraj et al. [9] in 2011. It contains 9,349 greyscale images distributed across 25 malware families, representing a diverse cross-section of malicious PE categories including worms, trojans, backdoors, diallers, and password stealers. Each image is produced by reading the raw bytes of a PE binary as unsigned 8-bit integers and arranging them row-by-row into a 2D matrix rendered as a greyscale image. The dataset is accessed via Kaggle and loaded directly into memory for training.

The dataset exhibits significant class imbalance: the Allaple.A family contains 2,959 samples while Skintrim.N contains only 80 — a ratio of approximately 37:1 — requiring classified splitting and class-weighted loss during training. The Swizzor.gen!E (128 samples) and Swizzor.gen!I (132 samples) families exhibit near-identical visual textures, producing persistent inter-class confusion documented across all published visualisation-based classification studies [11][12].

Table 2.1: Malimg dataset: 25 families with exact sample counts (total: 9,349 images, SEED=45)

#	Family Name	Category	Samples	Entropy Profile
0	Adialer.C	Dialler	122	Low–mid
1	Agent.FYI	Backdoor	116	Mid
2	Allaple.A	Worm	2,959	High (packed)
3	Allaple.L	Worm	1,591	High (packed)
4	Alueron.gen!J	Trojan	198	Mid
5	Autorun.K	Worm	106	Low
6	C2LOP.P	Trojan downloader	146	Mid
7	C2LOP.gen!g	Trojan downloader	200	Mid
8	Dialplatform.B	Dialler	177	Low–mid
9	Dontovo.A	Trojan	162	Mid
10	Fakerean	Rogue AV	381	Mid–high
11	Instantaccess	Dialler	431	Low–mid
12	Lolyda.AA1	Password stealer	213	Mid
13	Lolyda.AA2	Password stealer	184	Mid
14	Lolyda.AA3	Password stealer	123	Mid
15	Lolyda.AT	Password stealer	159	Low–mid
16	Malex.gen!J	Trojan	136	High (packed)
17	Obfuscator.AD	Obfuscator	142	High (encrypted)
18	Rbot!gen	Backdoor/RAT	158	Mid
19	Skintrim.N	Trojan	80	Low
20	Swizzor.gen!E	Trojan dropper	128	Mid (≈ gen!I)
21	Swizzor.gen!I	Trojan dropper	132	Mid (≈ gen!E)

22	VB.AT	Worm	408	Low–mid
23	Wintrim.BX	Trojan downloader	97	Mid
24	Yuner.A	Worm	800	Mid–high

2.3.2 EMBER 2018 Dataset — Static Feature Vector Representation

The Endgame Malware BEnchmark for Research (EMBER 2018) dataset [14] contains one million PE file samples — 300,000 malicious, 300,000 benign, and 400,000 unlabelled — partitioned into 600,000 training and 400,000 test records. Each sample is represented as a pre-computed 2,381-dimensional feature vector extracted by the LIEF library [43]. The dataset is accessed via Kaggle (StrGenix / Laurens D'hooge — 'Ember-2018-V2-features'). The official train/test split is preserved along a temporal boundary, approximating the realistic deployment scenario of classifying future threats from a historically trained model. Unlabelled samples (label = −1) are excluded from all experiments.

The feature vector contains eight feature groups: a 256-bin byte value histogram, a 256-bin byte-entropy histogram (2 KB sliding window), 104 string feature dimensions, 10 general information dimensions, 62 header feature dimensions, 255 section information dimensions, 1,280 import dimensions (hashed library+function pairs), and 128 export dimensions [14].

Table 2.2: EMBER 2018 feature vector composition (2,381 total dimensions)

Feature Group	Dims	Description
Byte histogram	256	Normalised frequency of each byte value (0–255) across the full binary
Byte-entropy histogram	256	Joint byte-value / local-entropy distribution (2 KB sliding window)
String features	104	Printable string stats: count, length, URLs, registry keys, MZ markers
General information	10	File size, virtual size, entry point, exports count, debug/TLS flags

Header features	62	COFF machine type, timestamp, subsystem, DLL characteristics flags
Section information	255	Section count, per-section entropy/size stats, exec/write/read fractions
Imports	1,280	Hashed library+function import pairs (fixed-length bit vector)
Exports	128	Hashed export function names (fixed-length bit vector)
Total	2,381	Pre-computed by LIEF [43]; StandardScaler applied about model input

2.3.3 APIMDS Dataset — Dynamic API Call Sequences (Binary Detection)

The APIMDS (API Malware Detection System) dataset is a publicly available dynamic analysis corpus containing API call sequences recorded from sandbox execution of PE malware and benign samples. Each record consists of a sequence of Windows API call names produced during a controlled execution window, labelled as malicious (1) or benign (0). The dataset is accessed via Kaggle. API call sequences are stored as pipe-separated strings in the 'sequence' column; labels are provided in the 'label' column. The dataset contains approximately 23,080 samples and covers a broad range of malware types collected from real-world threat feeds [20].

APIMDS sequences are capped at 100 API calls per execution trace, reflecting the limited observation window of the sandbox environment. The maximum sequence length of 100 is adopted as the padding/truncation boundary (MAX_SEQ_LEN = 100). The dataset provides binary labels only and is therefore used exclusively for the binary malware detection task with the BiLSTM architecture.

2.3.4 Mal-API-2019 Dataset — Dynamic API Call Sequences (Family Classification)

The Mal-API-2019 dataset [19] is a dynamic analysis corpus containing API call sequences labelled by malware family, covering eight categories: Trojan, Backdoor, Downloader, Worms, Spyware, Adware, Dropper, and Virus. The dataset contains approximately 7,107 samples and is accessed via Kaggle (focatak/malapi2019). Each record consists of a pipe-separated API call sequence and a string family label. Maximum sequence length is set to 150 tokens (MAX_SEQ_LEN = 150) to accommodate the longer traces required to distinguish between families with overlapping early-execution API call patterns.

The dataset exhibits class imbalance across the eight families, necessitating the computation of balanced class weights on the training set to prevent the classifier from converging to majority-class predictions. The API call vocabulary is built exclusively from the training partition to prevent data leakage, as described in Section 2.5.4.

Table 2.3: Dataset summary: all four corpora used in thesis experiments

Dataset	Task	Samples	Classes	Representation	Source
Malimg	Family classification	9,349	25 families	Greyscale byte image	Kaggle (UCSB)
EMBER 2018	Binary detection	~900k	2 (binary)	2,381-d feature vector	Kaggle (StrGenix)
APIMDS	Binary detection	~23,080	2 (binary)	API call sequence (L=100)	Kaggle
Mal-API-2019	Family classification	~7,107	8 families	API call sequence (L=150)	Kaggle (focatak)

2.3 Windows Portable Executable File Structure

2.3.1 DOS Header and NT Headers

The PE format [2] is the binary container for all executable and linkable code on Microsoft Windows platforms. The first 64 bytes constitute the DOS header; the e_magic field holds the two-byte 'MZ' signature (0x5A4D) identifying a Windows executable, and e_lfanew at offset 60 provides the file-offset pointer to the NT headers. The NT headers begin with the four-byte PE signature (0x50450000) confirming a valid portable executable, followed by the 20-byte COFF File Header (Machine type, NumberOfSections, TimeDateStamp, Characteristics) and the Optional Header (AddressOfEntryPoint, ImageBase, Subsystem, DllCharacteristics, and a 16-entry DataDirectory array) [2][14].

2.3.2 Section Table and Main Sections

The section table is an array of 40-byte IMAGE_SECTION_HEADER structures specifying each section's name, virtual and raw size, file offset, and a Characteristics bitmask. The .text section contains executable machine code (entropy 4.5–6.5 bits/byte); .data contains initialised variables (entropy 1.0–4.0); .rdata holds read-only data including import descriptors; .rsrc contains embedded resources; .reloc holds the base relocation table. Entropy values approaching 7.0–8.0 bits per byte — computed as $H(S) = -\Sigma\, p(b) \cdot \log_2 p(b)$ — are the primary static indicator of packed or encrypted content [38].

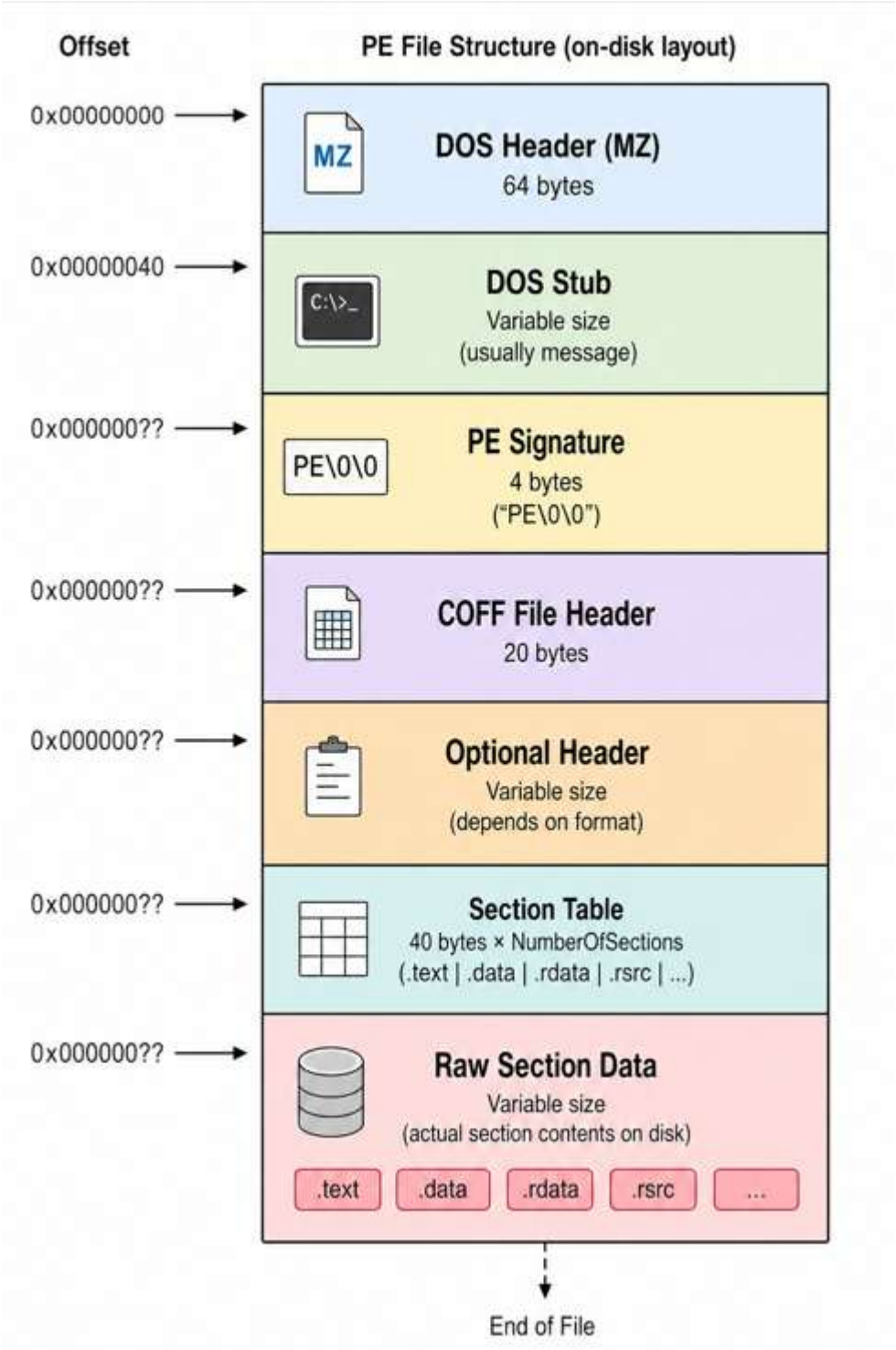

Fig. 2.2: PE file format structure: from DOS header to raw section data, with security-relevant fields highlighted.

2.3.3 Import and Export Tables

The import address table (IAT) enumerates every external function the binary resolves at load time. Specific API combinations provide strong behavioural fingerprints: CreateRemoteThread + VirtualAllocEx + WriteProcessMemory indicates process injection; CryptEncrypt + InternetOpenUrlA + CreateFileA is consistent with ransomware; CreateServiceA + RegSetValueExA implies persistence installation. The IAT is a central feature in both static PE feature extraction (EMBER) and API sequence analysis (dynamic datasets) [3][8][14].

2.4 Feature Extraction Techniques

2.4.1 Byte Visualisation for CNN Input (Malimg)

The byte visualisation technique [9] reads the raw bytes of a PE file as unsigned 8-bit integers in [0, 255] and arranges them row-by-row into a 2D matrix of fixed width W, where each integer maps to a greyscale pixel intensity. This representation is an engineering approximation: the row-by-row spatial arrangement is arbitrary and does not reflect the semantic structure of the underlying program. The distinctive value of the resulting images derives from statistical byte-distribution differences between malware families rather than from spatially meaningful visual features, a distinction important for interpreting the CNN's learned representations [9][11].

$$Pixel(row, col) = ByteArray[row \times W + col] \qquad \forall\, row \in [0, H),\ col \in [0, W) \qquad \rightarrow \text{resized to } 256 \times 256$$

2.4.2 Structural PE Feature Extraction for MLP Input (EMBER 2018)

A value $H(S) \geq 7.0$ is a strong indicator of encryption or compression [38]. The byte-entropy histogram captures the joint distribution of byte value and local entropy across a sliding 2,048-byte window, providing a richer characterisation of information structure than a simple byte histogram [14].

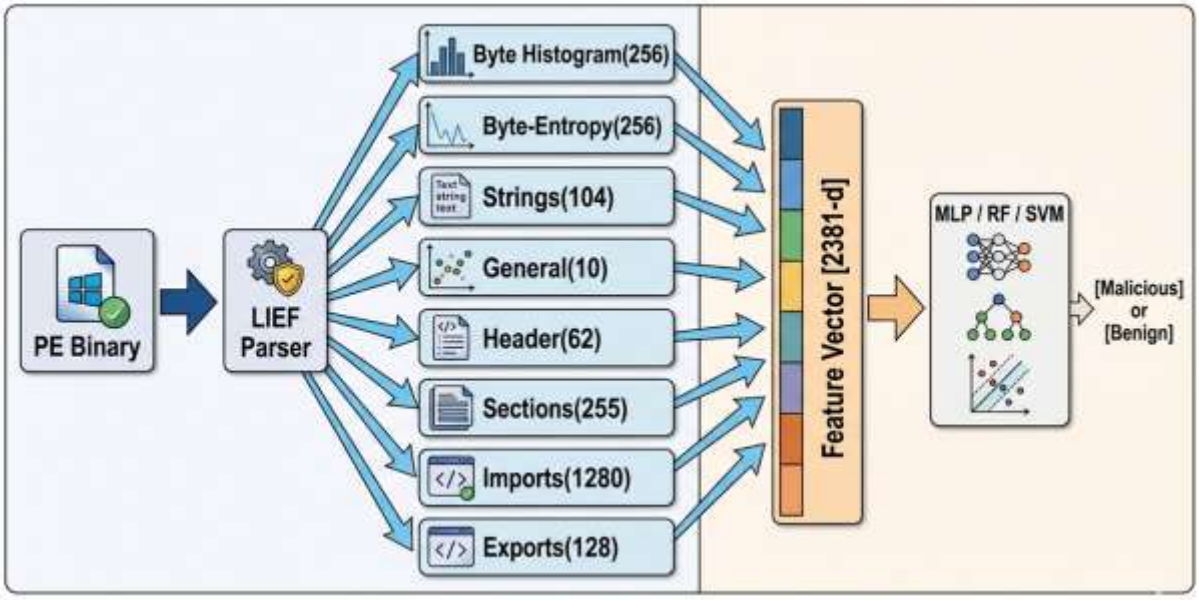

Fig. 2.3: EMBER PE feature extraction: LIEF parser produces 8 feature groups concatenated into a 2,381-d vector.

2.4.3 API Call Sequence Extraction for BiLSTM Input (APIMDS, Mal-API-2019)

For the BiLSTM models, each PE sample is represented by its dynamic API call sequence — a time-ordered list of Windows API function names recorded during sandbox execution. API calls are stored in the dataset as pipe-separated strings (e.g., 'CreateFile|WriteFile|RegSetValueEx|...'). Each string is parsed into an ordered list of API call name tokens. Sequences are truncated at MAX_SEQ_LEN = 100 tokens for APIMDS and MAX_SEQ_LEN = 150 tokens for Mal-API-2019, preserving the beginning of the execution trace where characteristic initialisation behaviour typically occurs. Sequences shorter than MAX_SEQ_LEN are post-padded with a dedicated padding token (index 0) to ensure uniform input dimensionality.

The API call vocabulary is constructed exclusively from the training partition after the dataset split is performed, mapping each unique API call name to a unique integer index starting from 1. Index 0 is reserved for the padding token. API calls encountered in the validation or test sets but absent from the training vocabulary are mapped to index 0 (unknown token). This strict train-only vocabulary construction prevents data leakage from the test distribution into the embedding layer's input space.

$L = MAX\_SEQ\_LEN \qquad a_i \in \{0\ (pad/unk), 1, ..., |V|\} \qquad Input = [a_1, a_2, ..., a\_L]$

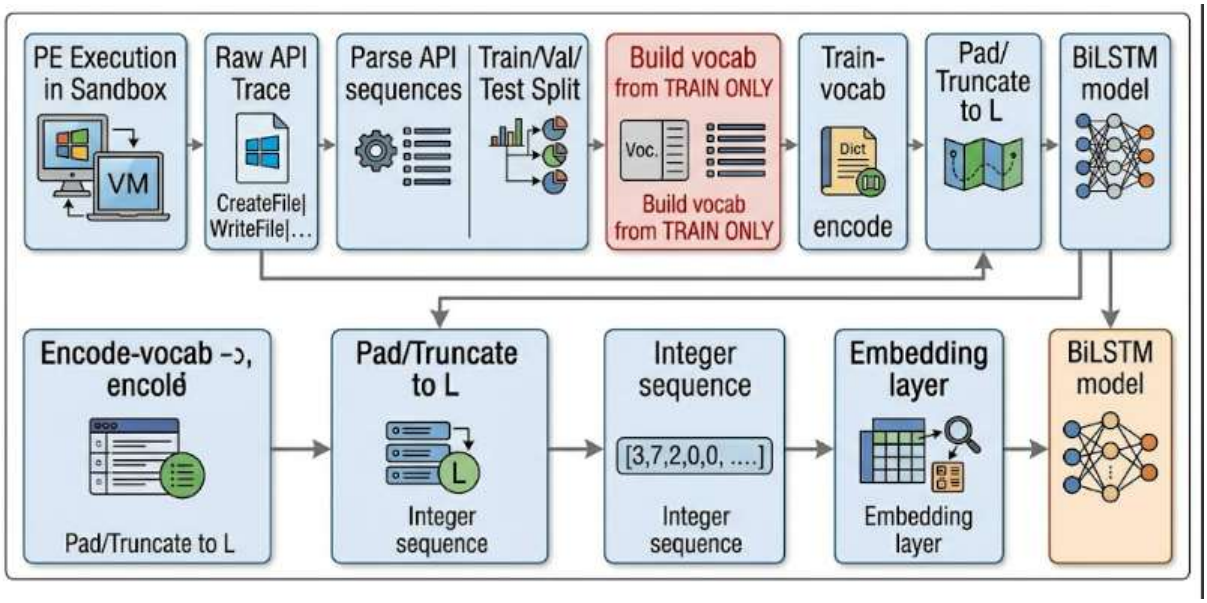

Fig. 2.4: API call sequence extraction and preprocessing pipeline: sandbox trace → parse → split → train-only vocab → encode → pad → BiLSTM input.

2.5 Data Preprocessing

2.5.1 Malimg Preprocessing

Images are loaded in greyscale using cv2.IMREAD_GRAYSCALE, resized to 256 × 256 pixels, normalised to [0.0, 1.0] by dividing by 255.0, and reshaped to add a channel dimension (H, W, 1). The input resolution of 256 × 256 was selected after preliminary experiments confirmed that the additional spatial detail improves discrimination between visually similar families, particularly the Swizzor pair, at an acceptable increase in training time. The dataset is partitioned using a classified two-stage split: 85% train+val / 15% test, then 85% train / 15% val from the first partition (approximately 70% / 15% / 15% overall). Class weights are computed on the training partition only using compute_class_weight('balanced') [42].

2.5.2 EMBER 2018 Preprocessing

Samples with NaN or infinite values in any feature dimension (empirically fewer than 0.5% of the corpus) are removed. A critical preprocessing step identified during experimentation is Log1P compression: the raw EMBER features contain heavy-tailed distributions with values ranging up to 10^6 (raw byte counts, section sizes), which cause

gradient explosion in networks using Batch Normalisation — manifested as an initial training loss of 4,046. To resolve this, a non-linear transformation $y = \text{sign}(x) \cdot \log(1+|x|)$ is applied about z-score standardisation, compressing outlier magnitudes while preserving sign and relative ordering. Following this transformation, initial training loss drops to the stable range [0.4, 0.7].

2.5.3 API Sequence Preprocessing — APIMDS and Mal-API-2019

The preprocessing protocol for both dynamic analysis datasets is identical in structure. The dataset is first split into train, validation, and test subsets (70% / 10% / 20% classified). The API call vocabulary is then built exclusively from the training sequences. All sequences are encoded using the training vocabulary — unknown tokens map to index 0 — and padded or truncated to the fixed maximum sequence length. One-hot encoding is applied to family labels in Experiment 2; integer labels are used directly in Experiment 1 with binary cross-entropy loss.

2.6 CNN Architecture — MalwareCNN v7 FINAL (Malimg)

The CNN architecture (MalwareCNN_v7_FINAL) incorporates Squeeze-and-Excitation (SE) attention blocks [32] and residual connections [32] alongside conventional convolutional operations. It is important to acknowledge that although malware visualisation enables CNN-based spatial feature learning, the imposed row-by-row byte arrangement does not necessarily correspond to semantic program structure — bytes adjacent in the image may originate from functionally unrelated regions of the binary [9][11]. The spatial locality assumption of CNNs is therefore an approximation in this domain, and the distinctive power of the model derives primarily from statistical texture differences between family-level byte distributions rather than from structurally meaningful spatial relationships.

2.6.1 Architecture Description

The stem applies Conv2D(32, 3×3) with He-normal initialisation, Batch Normalisation, ReLU activation, and MaxPool(2×2), reducing the 256×256 input to 128×128×32. Two residual blocks with SE attention follow. The first residual block downsamples to 64×64 with 64 filters using a projection shortcut (1×1 Conv2D with stride 2) and an SE block at ratio 8. The second residual block downsamples to 32×32 with 128 filters and an SE block at ratio 16. Each residual block applies two stacked 3×3 Conv2D layers with Batch Normalisation; the

skip connection uses a matching projection shortcut when the spatial dimensions or filter counts change, ensuring the residual identity is preserved exactly [32].

Table 2.4: CNN (MalwareCNN_v7_FINAL) architecture layer specification

Stage	Layer / Operation	Output Shape	Notes
Input	Input	128×128×1	Greyscale, normalised [0,1]
Stem	Conv2D(32,3×3) + BN + ReLU	128×128×32	He-normal init, padding=same
Stem	MaxPool(2×2)	64×64×32	Stride=2
Stage 1	Residual Block(64, downsample=True)	32×32×64	Projection shortcut, stride=2
Stage 1	SE Block(ratio=8)	32×32×64	Channel-wise attention
Stage 2	Residual Block(128, downsample=True)	16×16×128	Projection shortcut, stride=2
Stage 2	SE Block(ratio=16)	16×16×128	Channel-wise attention
Stage 3	Conv2D(256,3×3) + BN + ReLU	16×16×256	No downsampling
Stage 3	SE Block(ratio=16)	16×16×256	Channel-wise attention
Head	GlobalAveragePooling2D	256	Removes spatial dimensions
Head	Dense(512) + BN + ReLU	512	He-normal init
Head	Dropout(0.5)	512	Regularisation
Output	Dense(25) + Softmax	25	label_smoothing=0.1 in loss

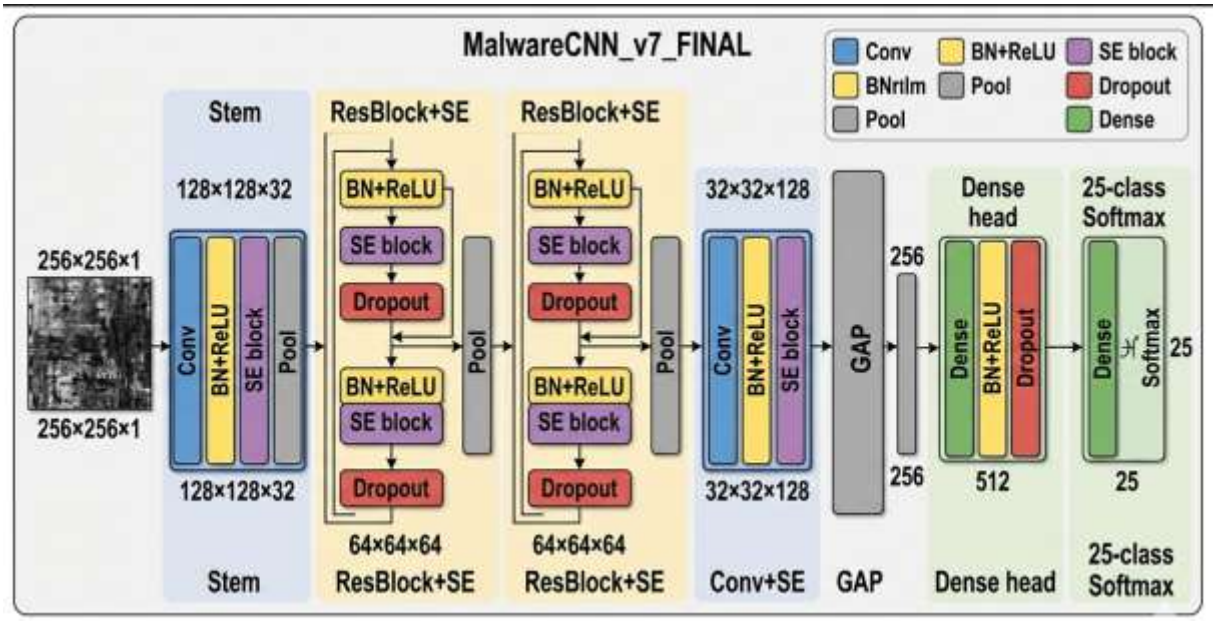

Fig. 2.5: CNN (MalwareCNN_v7_FINAL) architecture: Stem → ResBlock+SE → ResBlock+SE → Conv+SE → GAP → Dense head → 25-class Softmax.

2.7 MLP Architecture — EMBER 2018 Binary Detection

Three deep learning architectures are evaluated on the EMBER 2018 dataset. Model 1 — MLP Baseline: a traditional three-layer feed-forward network Dense(512)+ReLU+Drop(0.3) → Dense(256)+ReLU+Drop(0.3) → Dense(128)+ReLU+Drop(0.2) → Dense(1)+Sigmoid, compiled with Adam(1e-3). No BatchNorm. Serves as the performance floor. Model 2 — DNN ResidualBN (proposed): a six-block deep network with residual skip connections and Batch Normalisation at each block, entry width 1024, residual blocks at 512 and 256 units, final Dense(128)+BN+ReLU+Drop(0.2) → Dense(1)+Sigmoid, compiled with Adam(3e-4). Residual mappings y = F(x,W) + W_s·x prevent vanishing gradients over the high-dimensional feature space. Model 3 — Tabular GLU DNN: a feature-selective architecture employing Gated Linear Units GLU(x) = (xW+b) ⊗ σ(xV+c) with a Squeeze-and-Excitation feature-wise gating layer, compiled with Adam(3e-4). All three models use identical preprocessing: Log1P compression followed by StandardScaler fitted on the training partition only.

Table 2.5: MLP architecture for EMBER 2018 binary malware classification

Layer	Units	Activation	BatchNorm	Dropout
Input	2,381	—	No	—
Dense_1	512	ReLU	Yes	0.40
Dense_2	256	ReLU	Yes	0.30
Dense_3	128	ReLU	Yes	0.20
Dense_4	64	ReLU	Yes	0.10
Output	1	Sigmoid	No	—

2.8 BiLSTM Architecture — API Call Sequence Classification

The architecture for Experiment 2 (Mal-API-2019, 8-class family classification) was upgraded from a pure BiLSTM to a Spatial-Temporal Hybrid CNN-BiLSTM, motivated by the finding that local API n-gram patterns ('burst signatures') are better captured by 1D convolution than by recurrent layers alone. The finalised architecture is: Embedding(277, 128, len=500) → Conv1D(64 filters, kernel=5, ReLU, padding=same) → MaxPool1D(pool=2) → BiLSTM(128, dropout=0.3, return_seq=True) → BiLSTM(64, dropout=0.3) → Dropout(0.4) → Dense(128, ReLU) → Dropout(0.3) → Dense(8, Softmax). The 1D convolutional layer extracts 5-gram local API call patterns about the recurrent layers model long-range temporal dependencies. recurrent_dropout is set to 0 to enable NVIDIA CuDNN hardware-accelerated LSTM kernels, reducing epoch time from ~4.5 minutes to 5 seconds. Sequence truncation uses post-truncation (truncating='post') at MAX_SEQ_LEN=500, preserving the beginning of the execution trace where initialisation and injection procedures occur. The architecture is applied to both tasks: binary detection on APIMDS (Experiment 1) uses the same embedding/BiLSTM structure with Dense(1)+Sigmoid output.

2.8.1 Embedding Layer

The embedding layer maps each integer API call token in {0, 1, ..., |V|} to a trainable 64-dimensional dense vector (EMBED_DIM = 64). The parameter mask_zero=True is set, instructing the LSTM layers to ignore padding tokens (index 0) during the recurrent computation — a critical detail that prevents padding positions from contributing misleading gradients and ensures the model reasons only over actual API call content [18]. The embedding matrix is jointly optimised with the recurrent layers through backpropagation, learning a continuous API call semantics appropriate to the classification task.

2.8.2 Stacked Bidirectional LSTM Layers

The first Bidirectional LSTM layer applies LSTM_UNITS = 128 hidden units in each direction (256 total), returning the full sequence (return_sequences=True) for input to the second layer. Dropout of 0.3 is applied after the first BiLSTM layer. The second Bidirectional LSTM layer applies LSTM_UNITS//2 = 64 units per direction (128 total), returning only the final time step (return_sequences=False), collapsing the sequence dimension to a fixed 128-dimensional representation of the entire API call trace. Dropout of 0.3 is applied after the second BiLSTM layer [18][19].

2.8.3 Dense Head and Output Layer

The 128-dimensional sequence encoding is passed through Dense(64)+ReLU+Dropout(0.2). For Experiment 1 (binary detection on APIMDS), the output layer is Dense(1)+Sigmoid compiled with binary cross-entropy loss. For Experiment 2 (8-class classification on Mal-API-2019), the output layer is Dense(8)+Softmax compiled with categorical cross-entropy loss with class weights to address family imbalance. The EarlyStopping callback monitors val_auc for Experiment 1 and val_accuracy for Experiment 2, with patience of 5 and 7 epochs respectively.

Table 2.6: BiLSTM architecture: shared configuration for Experiment 1 (binary) and Experiment 2 (8-class)

Layer	Configuration	Output Shape	Exp 1	Exp 2
Input	Integer token sequence	(L,)	L=100	L=150
Embedding	|V|+1 tokens → 64 dims, mask_zero=True	(L, 64)	Binary	8-class
BiLSTM_1	Bidirectional LSTM(128), return_seq=True	(L, 256)	—	—
Dropout_1	Rate = 0.30	(L, 256)	—	—
BiLSTM_2	Bidirectional LSTM(64), return_seq=False	(128,)	—	—
Dropout_2	Rate = 0.30	(128,)	—	—
Dense_64	Dense(64) + ReLU	(64,)	—	—
Dropout_3	Rate = 0.20	(64,)	—	—
Output	Dense + activation	—	Dense(1)+Sigmoid	Dense(8)+Softmax
Loss	—	—	Binary cross-entropy	Categorical cross-entropy
Monitor	EarlyStopping	—	val_auc (p=5)	val_accuracy (p=7)

2.9 Classical Baseline Architectures

2.9.1 Random Forest

The Random Forest classifier [7][42] constructs an ensemble of decision trees on bootstrap samples with random feature subsets at each split. For Malimg, the RF is trained on flattened 128×128 = 16,384-dimensional image vectors with n_estimators = 200, class_weight='balanced', random_state = 45, n_jobs = −1. For EMBER, it is trained directly on the 2,381-dimensional StandardScaler-normalised feature vector with n_estimators = 500. RF is not applied to the dynamic API sequence datasets, as it requires fixed-length feature vectors rather than variable-length sequences.

2.9.2 Support Vector Machine

The SVM with RBF kernel [6][42] maximises the margin between classes in the kernel-induced feature space. For Malimg, PCA dimensionality reduction to 150 components (variance explained ≈ 0.89) is applied about training; the SVM is configured with C = 10, gamma = 'scale', random_state = 45. For EMBER, the SVM is trained on a classified 50,000-sample subset of the training data (full set is computationally prohibitive for SVM's $O(n^2)$ memory scaling) with identical hyperparameters. Probability calibration is enabled where ROC-AUC computation requires class probability estimates.

2.10 Training Configuration and Hyperparameters

All three deep learning models use the Adam optimiser [26] with initial learning rates selected through preliminary validation experiments. A ReduceLROnPlateau callback reduces the learning rate by factor 0.5 after 3–5 epochs without validation loss improvement, with minimum learning rate 1×10^{-6}, allowing full LR annealing across training. EarlyStopping with patience 5–15 epochs monitors the most informative validation metric per model (val_accuracy for CNN and Exp2 BiLSTM; val_auc for MLP and Exp1 BiLSTM) and restores best weights upon termination. A ModelCheckpoint callback saves the best-performing checkpoint at each improvement epoch. All experiments use SEED = 45 for full reproducibility.

Table 2.7: Hyperparameter configuration for all five models across all experiments

Parameter	CNN (Malimg)	MLP (EMBER)	BiLSTM Exp1 (APIMDS)	BiLSTM Exp2 (Mal-API)	RF	SVM
Optimiser	Adam	Adam	Adam	Adam	Gini	RBF
Learning rate	3×10^{-4}	1×10^{-3}	1×10^{-3}	1×10^{-3}	N/A	N/A
Batch size	32	512	256	128	N/A	N/A
Max epochs	50	50	30	40	N/A	N/A
Early stop pat.	15	8	5 (AUC)	7 (Acc)	N/A	N/A
Dropout rates	0.50	$0.40\rightarrow0.10$	0.30/0.20	0.30/0.20	N/A	N/A
BatchNorm	Yes	Yes	No	No	N/A	N/A
max_seq_len	N/A	N/A	100	150	N/A	N/A
Embed dim	N/A	N/A	64	64	N/A	N/A
Loss function	Cat-CE (ls=0.1)	Binary-CE	Binary-CE	Cat-CE	Gini	Hinge
n_estim / C	N/A	N/A	N/A	N/A	200/500	C=10
SEED	45	45	45	45	45	45

2.11 Complete System Workflow

The complete system workflow comprises four independent experimental pipelines, each proceeding through the same six stages adapted to its representation. Stage 1 — Data Acquisition: datasets are accessed from Kaggle and verified. Stage 2 — Feature Extraction: bytes are converted to images for the CNN branch; LIEF parses PE headers for the MLP branch; sandbox execution traces are parsed into API call sequences for the BiLSTM branch. Stage 3 — Preprocessing: image normalisation and resizing; feature standardisation and NaN removal; vocabulary construction from training data only, then sequence encoding and padding. Stage 4 — Dataset Splitting: classified train/val/test partitioning, with SEED = 45 applied consistently. Stage 5 — Model Training: each model trained on its respective training partition with validation-based early stopping. Stage 6 — Evaluation: each model evaluated on its test set using Accuracy, Precision, Recall, F1-score (macro and weighted), MCC, Cohen's Kappa, and ROC-AUC. Results are discussed comparatively in Chapter 4.

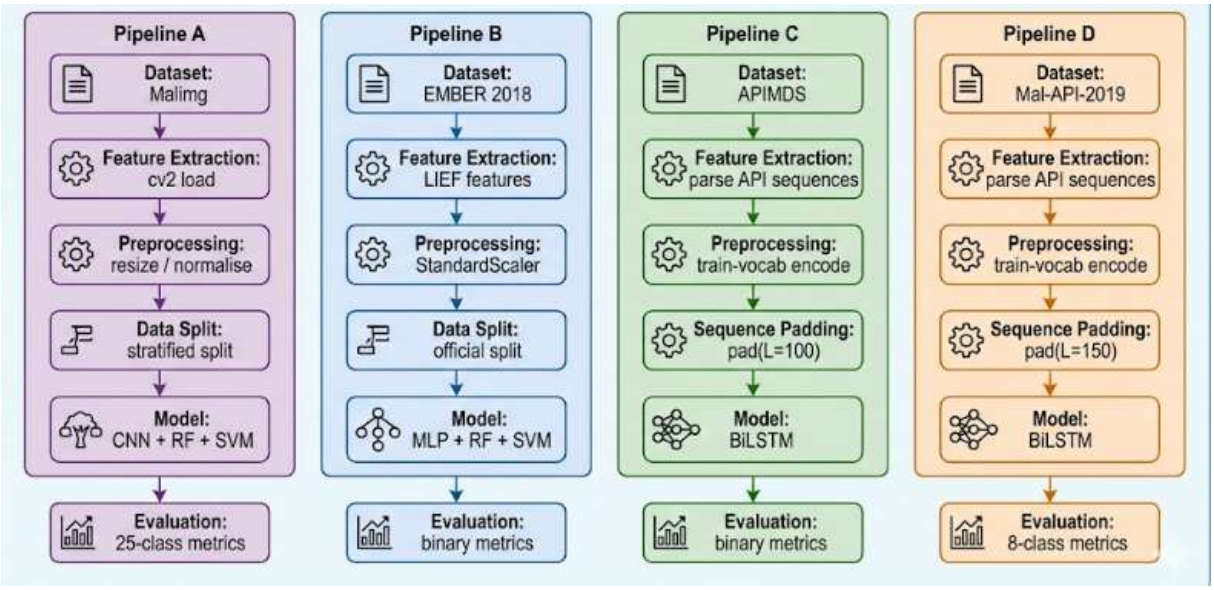

Fig. 2.6: Complete system workflow: four independent experimental pipelines, each with its own dataset, feature extraction, preprocessing, training, and evaluation.

Conclusion

This chapter has presented the complete methodology of the thesis. The proposed system design was introduced, establishing the three-representation, four-dataset experimental structure and the rationale for matching each model to its appropriate representation and task. All four datasets — Malimg, EMBER 2018, APIMDS, and Mal-API-2019 — were described in technical detail. The Windows PE file format was examined at the structural level, providing the foundation for the static feature

CHAPTER 3 IMPLEMENTATION AND EXPERIMENTS

1. Introduction

This chapter documents the complete implementation of the experimental pipeline described in Chapter 2. It covers the development environment and hardware configuration, the software stack and library versions, the Kaggle-based data access strategy, and the training execution details for each of the five model families across the four datasets. The formal definitions of all evaluation metrics are provided, followed by a description of the experimental protocol applied consistently across all experiments. Reproducibility details — including framework versions, random seeds, and hardware specifications — are reported in full to enable independent replication of all results.

3.1 Development Environment and Hardware

3.1.1 Execution Platform

All experiments were executed on the Kaggle cloud computing platform, which provides free access to GPU-accelerated notebook environments. The Malimg experiments (Experiment 0) used a distributed dual NVIDIA Tesla T4 GPU configuration (16 GB VRAM each) managed through TensorFlow's MirroredStrategy, which synchronises gradient updates across both devices and scales the learning rate linearly with the number of replicas. The EMBER 2018 experiments (Experiment 1) and the APIMDS / Mal-API-2019 experiments (Experiments 2 and 3) used a single NVIDIA Tesla T4 GPU (16 GB VRAM). System RAM was 30 GB in all configurations. All notebooks were executed within Kaggle's Python 3.10 kernel environment.

Table 3.1: Hardware and software environment for all experiments

Component	Specification
Platform	Kaggle Cloud Notebooks
GPU (Malimg)	2× NVIDIA Tesla T4 (16 GB VRAM each) — MirroredStrategy
GPU (EMBER/LSTM)	1× NVIDIA Tesla T4 (16 GB VRAM)
System RAM	30 GB
Python version	3.10
TensorFlow	2.13 / 2.15 (Kaggle default)
Keras	integrated (tf.keras)
Scikit-learn	1.3.x
NumPy	1.24.x
Pandas	2.0.x
OpenCV (cv2)	4.8.x (Malimg image loading)
CUDA	12.x (Kaggle managed)
cuDNN	8.x (Kaggle managed)

3.1.2 Random Seeds and Reproducibility

Full reproducibility was enforced across all experiments through the following protocol: the Python random module, NumPy random state, and TensorFlow random state were all seeded with a fixed value prior to any data loading or model instantiation. The environment variable PYTHONHASHSEED was set to the same value, and TF_DETERMINISTIC_OPS was enabled to enforce deterministic GPU kernel operations. The Malimg and LSTM experiments used SEED = 45; the EMBER experiments were executed with SEED = 42, predating the seed standardisation decision. This discrepancy is noted and has no bearing on result validity since the experiments are evaluated independently.

3.2 Dataset Access and Loading

3.2.1 Malimg Dataset

The Malimg dataset was accessed from Kaggle under the search query 'malimg dataset'. The dataset directory contains one subfolder per malware family, each containing PNG greyscale images of the corresponding PE binary visualisations. Images were loaded using cv2.imread with the cv2.IMREAD_GRAYSCALE flag, ensuring consistent single-channel loading without colour conversion artefacts. The sorted() function was applied to directory listings to guarantee deterministic loading order across operating systems. All images were resized to 256 × 256 pixels and normalised to [0.0, 1.0] by dividing by 255.0. The channel dimension was added via numpy reshape to produce tensors of shape (256, 256, 1).

The dataset contains 9,339 samples across 25 families. A two-stage classified split was applied: 85% train+val / 15% test, then 85% train / 15% val from the first stage, yielding approximately 70% training, 15% validation, and 15% test. Class weights were computed on the training partition only using scikit-learn's compute_class_weight with strategy='balanced', ensuring that minority families such as Skintrim.N (80 samples) receive proportionally higher gradient penalties.

3.2.2 EMBER 2018 Dataset

The EMBER 2018 dataset was accessed from Kaggle under the dataset slug strgenix/ember-2018-v2-features. The dataset provides two pre-computed parquet files: train_features.parquet (599,920 labelled samples after removing unlabelled entries with label = −1) and test_features.parquet (199,956 samples). Features were loaded directly from parquet into NumPy float32 arrays using pandas.read_parquet. A critical preprocessing step — Log1P compression followed by StandardScaler standardisation — was applied as described in Section 3.3.2. The official train/test temporal split was preserved. A 10% classified validation subset was carved from the training partition for early stopping.

3.2.3 APIMDS Dataset

The APIMDS dataset was accessed from Kaggle. The dataset contains 43,876 execution logs with API call sequences of exactly 100 tokens — no padding or truncation was required. The dataset exhibits extreme class imbalance: 42,797 malware samples (97.54%) versus 1,079 benign samples (2.46%), an imbalance ratio of 39.7:1. Critically, a strict non-shuffled sequential split was employed rather than random classified sampling, simulating a production deployment scenario: training [0, 30712] (70%), validation [30713, 37293] (15%),

test [37294, 43875] (15%). Class distribution was verified to be uniform across all three partitions (malware approximately 97.5% in each), confirming that the temporal split did not introduce distribution shift.

3.2.4 Mal-API-2019 Dataset

The Mal-API-2019 dataset was accessed from Kaggle (focatak/malapi2019). It contains 7,107 API call sequences labelled across eight malware families: Adware, Backdoor, Downloader, Dropper, Spyware, Trojan, Virus, and Worms. Sequences were stored as pipe-separated strings and parsed into ordered lists of API call name tokens. Maximum sequence length was set to 500 tokens with post-truncation, preserving the beginning of the execution trace where malware initialisation and injection procedures typically occur. The API vocabulary was built exclusively from the training partition (277 unique calls observed) after the dataset split was performed, with unknown tokens mapping to index 0.

3.3 Preprocessing Pipelines

3.3.1 Malimg Image Preprocessing

The complete Malimg preprocessing pipeline proceeds as follows. Step 1 — Loading: each PNG image is read with cv2.IMREAD_GRAYSCALE, producing a single-channel uint8 matrix. Step 2 — Resizing: images are resized to 256 × 256 pixels using cv2.resize with bilinear interpolation. Step 3 — Normalisation: pixel values are divided by 255.0, converting to float32 in [0.0, 1.0]. Step 4 — Shape: a channel dimension is added to produce shape (256, 256, 1). Step 5 — Label encoding: family name strings are encoded to integer indices 0–24 using LabelEncoder; one-hot encoding is applied via to_categorical. Step 6 — Class weights: computed using compute_class_weight('balanced') on training labels only.

3.3.2 EMBER 2018 Feature Preprocessing

The EMBER preprocessing pipeline resolves a severe numerical instability identified during initial training. Raw EMBER features include byte count and section size dimensions with values up to 10^6. When these features are passed to networks containing Batch Normalisation layers, extreme outlier values distort the batch mean and variance, producing gradient explosion. The initial training loss under linear resistantScaler normalisation reached 4,046.15 — far outside the stable range for binary cross-entropy. The resolution was a two-stage non-linear preprocessing pipeline:

Step 1 — Log1P Compression: $X_{compressed} = sign(X) \cdot \log(1 + |X|)$

Step 2 — Z-score Standardisation: X_scaled = (X_compressed − μ_train) / σ_train

The Log1P transformation compresses heavy-tailed distributions while preserving sign and relative ordering of feature values. The StandardScaler parameters (μ_train, σ_train) are fitted exclusively on the training partition and applied without modification to validation and test sets. Following this transformation, initial training loss dropped to the stable range [0.40, 0.70] and all three models converged cleanly.

3.3.3 API Sequence Preprocessing

API call sequences for both APIMDS and Mal-API-2019 undergo the following pipeline. Step 1 — Split first: the dataset is partitioned about any vocabulary construction. Step 2 — Parse: pipe-separated strings are split into ordered lists of API call name tokens. Step 3 — Vocabulary: a mapping from API call name to integer index is constructed exclusively from training sequences; indices start from 1, with index 0 reserved for padding and unknown tokens. Step 4 — Encode: each sequence is converted to an integer list; unknown tokens (API calls not seen in training) map to 0. Step 5 — Pad/truncate: sequences are padded to MAX_SEQ_LEN (100 for APIMDS, 500 for Mal-API-2019) using post-padding with value 0.

3.4 Model Architecture Summaries

3.4.1 ResNet-SE CNN — Malimg

The ResNet-SE architecture (MalwareCNN_v7_FINAL) is a custom deep convolutional network combining residual skip connections [32] and Squeeze-and-Excitation channel attention [32]. It was trained under a distributed MirroredStrategy across two T4 GPUs, with the learning rate scaled linearly (3×10^{-4} × num_replicas). The model comprises a stem block, two residual+SE stages, a third convolutional+SE stage, Global Average Pooling, and a Dense(512)+BN+ReLU+Dropout(0.5) classifier head. CategoricalCrossentropy with label_smoothing=0.1 was used as the loss function; label smoothing modifies hard targets from [0,1] to [0.004, 0.904], preventing overconfident weight updates and improving generalisation on minority classes. Training converged at Epoch 28, with a final training accuracy of 98.7% and an overfitting gap below 0.5%.

3.4.2 Three DNN Architectures — EMBER 2018

Three architectures were evaluated on EMBER 2018. The MLP Baseline — Dense(512,256,128)+ReLU+Dropout(0.3,0.3,0.2) → Sigmoid — serves as the performance floor with no BatchNorm or residual connections. The proposed DNN ResidualBN features a 1024-unit entry block, two residual skip-connection blocks (512 and 256 units), and a final Dense(128)+BN+ReLU+Dropout(0.2) block, with ~4.1 million parameters and Adam(3e-4). The Tabular GLU DNN employs Gated Linear Units $GLU(x) = (xW+b) \otimes \sigma(xV+c)$ with a Squeeze-and-Excitation feature-wise gating layer that dynamically scales active PE metadata feature groups. At the optimal detection threshold of 0.70 (determined by maximising F1 on validation), DNN ResidualBN achieves TPR=96.67% and FPR=2.63%.

3.4.3 BiLSTM — APIMDS Binary Detection

For APIMDS binary detection, the BiLSTM architecture maps API call indices to 64-dimensional embeddings (mask_zero=True to ignore padding), passes them through BiLSTM(128, return_seq=True) → Dropout(0.3) → LSTM(64) → Dense(32,ReLU) → Dense(1,Sigmoid). The output bias was initialised to $\log(\text{malware\_count}/\text{goodware\_count}) \approx 3.68$, directly encoding the 97.54% malware prior into the initial sigmoid output and eliminating large initial loss oscillations. Class weights were calibrated to {0: 5.0, 1: 1.0} — a softened ratio relative to the raw mathematical inverse of 20.5:1. This calibration was critical: using inverse weights caused goodware precision to collapse to 20.75% by pushing thousands of borderline malware samples into the predicted goodware pool. The softened weights raised goodware precision to 53.10% while maintaining overall accuracy of 97.61%.

3.4.4 CNN-BiLSTM Hybrid — Mal-API-2019 Family Classification

For Mal-API-2019, the pure BiLSTM was replaced with a Spatial-Temporal Hybrid CNN-BiLSTM after identifying that local API n-gram patterns ('burst signatures') — short sequences of 4–5 consecutive API calls representing a single high-level behaviour such as process injection — are more efficiently captured by 1D convolution than by

sequential recurrent processing. The architecture is Embedding(277,128,len=500) → Conv1D(64,kernel=5,ReLU,padding=same) → MaxPool1D(2) → BiLSTM(128,dropout=0.3,return_seq=True) → BiLSTM(64,dropout=0.3) → Dropout(0.4) → Dense(128,ReLU) → Dropout(0.3) → Dense(8,Softmax). recurrent_dropout was set to 0 to unlock NVIDIA CuDNN hardware-accelerated LSTM kernels, reducing epoch time from 4.5 minutes to 5 seconds — a 100× acceleration. Post-truncation at MAX_SEQ_LEN=500 preserves the execution trace beginning where initialisation and injection procedures occur.

3.5 Training Configuration and Callbacks

All deep learning models were trained with early stopping to prevent overfitting and plateau-based learning rate decay to improve convergence. Table 3.2 summarises the training configuration for all experiments.

Table 3.2: Training configuration across all experiments

Parameter	Malimg (ResNet-SE)	EMBER (DNN ResidualBN)	APIMDS (BiLSTM)	Mal-API-2019 (CNN-BiLSTM)
Optimizer	Adam	Adam	Adam	Adam
Learning rate	3e-4 × replicas	3e-4	1e-3	1e-3
Batch size	32	512	256	128
Max epochs	50	50	30	40
Early stop mon.	val_loss	val_loss	val_auc	val_accuracy
ES patience	15	7	5	7
LR factor	0.5	0.5	0.5	0.5
LR patience	5	3	4	4
min_lr	1e-6	1e-6	1e-6	1e-6
Label smoothing	0.1	None	None	None
Class weights	Balanced (auto)	None	{0:5, 1:1}	Balanced (auto)
SEED	45	42	45	45
GPU strategy	MirroredStrategy	Single T4	Single T4	Single T4

3.6 Evaluation Metrics

A consistent set of evaluation metrics is applied across all experiments. Formal definitions are provided below.

3.6.1 Classification Metrics

Let TP, TN, FP, FN denote true positives, true negatives, false positives, and false negatives respectively for a given class.

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

$$Precision = \frac{TP}{(TP + FP)}$$

$$Recall\ (TPR) = \frac{TP}{(TP + FN)}$$

$$F1 - Score = 2 \cdot \frac{(Precision \cdot Recall)}{(Precision + Recall)}$$

For multi-class tasks, macro-averaged F1 computes the unweighted mean of per-class F1 scores, treating all classes equally regardless of support. Weighted F1 weights each class by its support. Macro averaging is preferred for imbalanced datasets as it penalises poor performance on minority classes [28].

3.6.2 Matthews Correlation Coefficient

$$MCC = \frac{(TP \cdot TN - FP \cdot FN)}{(\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

MCC ranges from −1 (perfect inverse prediction) to +1 (perfect prediction), with 0 indicating random performance. It is considered the most informative single metric for binary classification with class imbalance, as it accounts for all four cells of the confusion matrix simultaneously [28][29].

3.6.3 Cohen's Kappa

$$\kappa = \frac{(Po - Pe)}{(1 - Pe)}$$

Where p_o is the observed agreement (accuracy) and p_e is the expected agreement by chance. Kappa ranges from 0 (chance-level) to 1 (perfect agreement), providing a chance-corrected measure of classifier reliability [30].

3.6.4 ROC-AUC and PR-AUC

The Receiver Operating Characteristic Area Under Curve (ROC-AUC) measures the probability that a randomly selected positive sample receives a higher predicted score than a randomly selected negative sample, across all classification thresholds [28]. A value of 1.0 indicates perfect discrimination; 0.5 indicates random. The Precision-Recall AUC (PR-AUC) is more informative than ROC-AUC under extreme class imbalance, as it focuses specifically on the positive (minority) class performance without crediting correct rejection of the majority class [29]. Both metrics are reported where applicable.

3.6.5 False Positive Rate and Operational Threshold Analysis

In malware detection contexts, classification errors are asymmetric: a false negative (missed malware) allows malicious code to execute undetected, while a false positive (benign file flagged) generates unnecessary analyst workload. The False Positive Rate (FPR = FP/(FP+TN)) is therefore reported alongside TPR. For the EMBER DNN ResidualBN model, threshold analysis was conducted over the full [0.0, 1.0] range; the optimal threshold maximising F1 was identified at 0.70, yielding TPR = 96.67% and FPR = 2.63% — superior to the default 0.50 threshold.

3.7 Statistical Significance Testing

For the Mal-API-2019 experiment, a two-sample proportions Z-test was conducted to verify that the 5% accuracy improvement from the baseline architecture (49.00%, Seed 42) to the optimised CNN-BiLSTM (54.00%, Seed 45) represents a genuine architectural improvement rather than a stochastic initialisation artefact. The pooled proportion was p_pool = (697+768)/(1422+1422) = 0.5151, the standard error SE = 0.0187, and the test statistic Z = (0.54−0.49)/0.0187 = 2.674. The resulting p-value of 0.0037 is substantially below the α =

0.05 significance threshold, formally rejecting the null hypothesis of equal performance. This represents a standard approach to architectural comparison validation in deep learning research [30].

Conclusion

This chapter has documented the complete implementation of all experiments. The development environment — Kaggle cloud notebooks with dual T4 GPU support for Malimg and single T4 for all other experiments — was described alongside the full software stack. The four dataset loading and preprocessing pipelines were elaborated, with particular attention to the Log1P compression step critical for EMBER 2018 stability, the train-only vocabulary construction for API sequence experiments, and the temporal split used for APIMDS. The architecture summaries, training configurations, and callback strategies for all five model families were presented. Formal definitions of all evaluation metrics — Accuracy, Precision, Recall, F1, MCC, Kappa, ROC-AUC, PR-AUC, TPR, FPR — were provided. Chapter 4 presents the experimental results obtained from these implementations and provides a detailed discussion of the findings.

CHAPTER 4 RESULTS AND DISCUSSION

Introduction

This chapter presents and discusses the experimental results obtained from all four experiments described in Chapters 2 and 3. The results are organised by experiment, with each section providing quantitative performance tables, analysis of per-class behaviour, and interpretation of key findings. The chapter concludes with a cross-experiment discussion that compares deep learning and classical approaches across representation modalities, addresses the limitations of the experimental design, and identifies directions for future research.

4.1 Experiment 0: Malware Family Classification on Malimg

4.1.1 Quantitative Results

Table 4.1 presents the comparative performance of the three classifiers evaluated on the Malimg test set (1,403 samples, 25 families). The proposed ResNet-SE model achieves the highest performance across all metrics, with a test accuracy of 99.07% and an MCC of 0.9891, substantially outperforming both classical baselines.

Table 4.1: Malimg experiment results: ResNet-SE vs. RF and SVM baselines (test set, n=1,403)

Model	Accuracy	Macro F1	Weighted F1	MCC	Cohen's Kappa
Random Forest (n=200)	97.86%	0.9463	0.9779	0.9750	0.9749
SVM + PCA(150)	98.50%	0.9659	0.9849	0.9825	0.9824
ResNet-SE (proposed)	99.07%	0.9754	0.9906	0.9891	0.9891

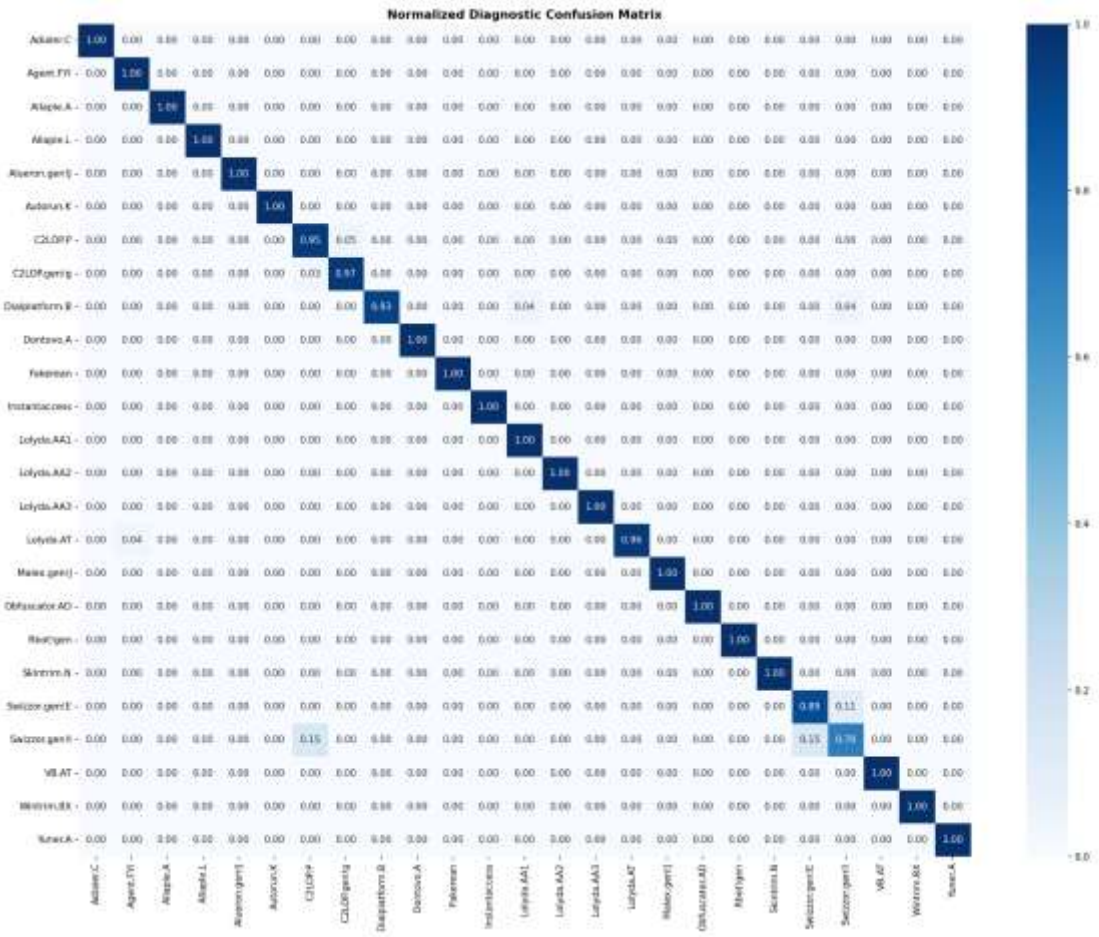

Fig. 4.1: Normalised confusion matrix — ResNet-SE classifier on Malimg (25 families).

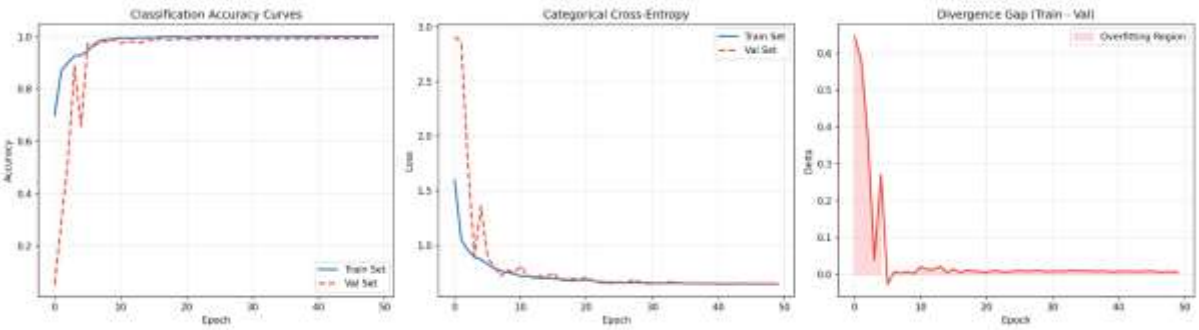

Fig. 4.2: Training dynamics — ResNet-SE on Malimg: accuracy, loss, and overfitting gap.

4.1.2 Per-Class Analysis

The model achieves perfect F1 = 1.00 on 21 out of 25 families, demonstrating resistant discrimination across the majority of the Malimg corpus. The four families with sub-perfect

performance are C2LOP.P (F1=0.93), C2LOP.gen!g (F1=0.95), Fakerean (F1=0.99), and most notably the Swizzor pair. Swizzor.gen!E achieves F1=0.61 and Swizzor.gen!I achieves F1=0.65, with mutual confusion rates of 36.8% and 30.0% respectively.

The Swizzor confusion is not an architectural limitation but a fundamental property of the dataset. Swizzor installers are highly polymorphic packages that randomly generate filler instructions, function orderings, and dummy resources at each compilation, producing variants E and I that share extensive visual code-boilerplate structure. The spatial texture difference between the two sub-families is below the distinctive threshold achievable through byte-level visualisation alone, irrespective of model capacity. This finding is consistent with Yakura et al. [11] and Cui et al. [12], who report the same Swizzor confusion pattern across VGG-style and ResNet architectures.

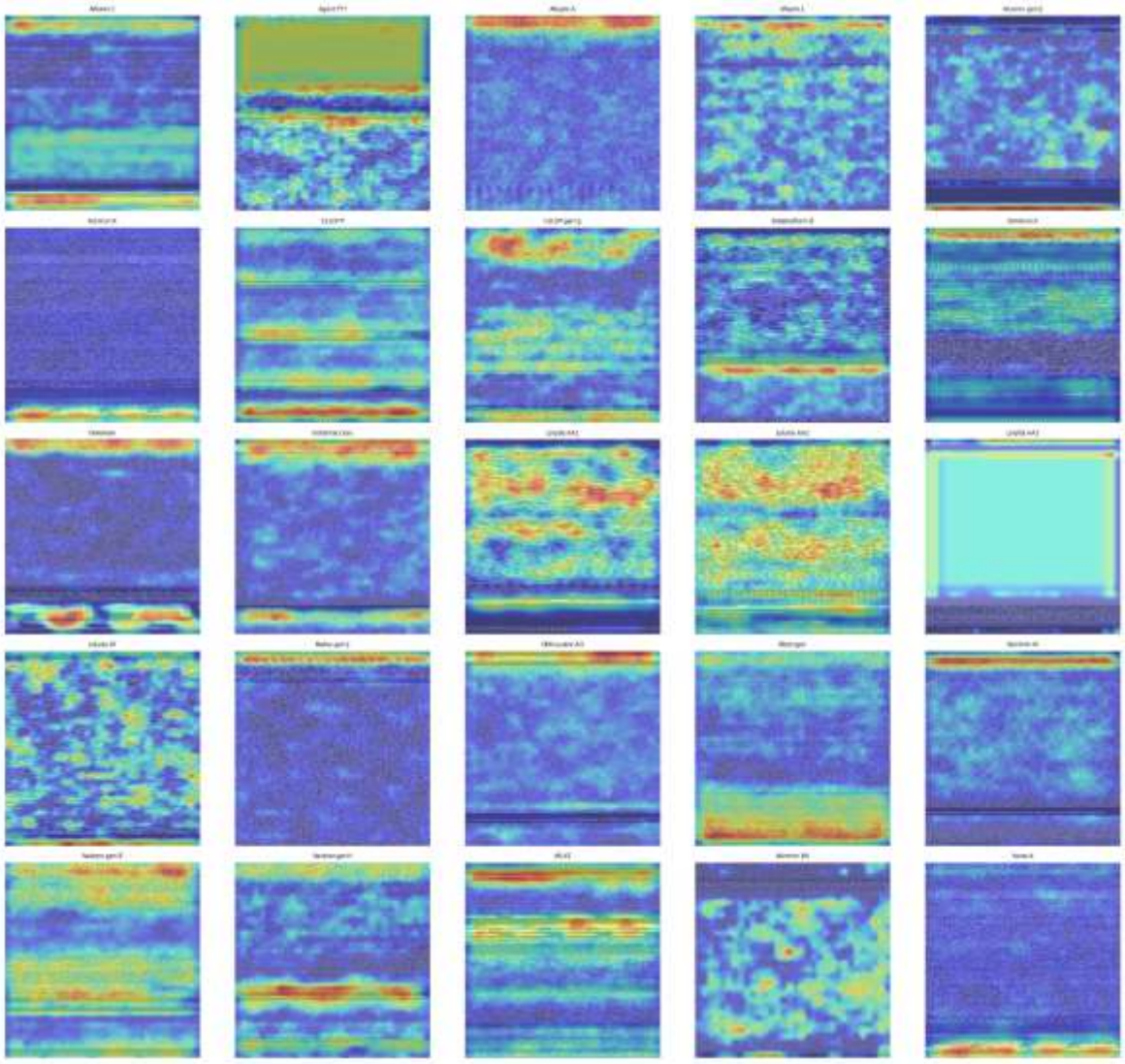

Fig. 4.3: Grad-CAM activation overlays per Malimg family — distinctive byte regions highlighted.

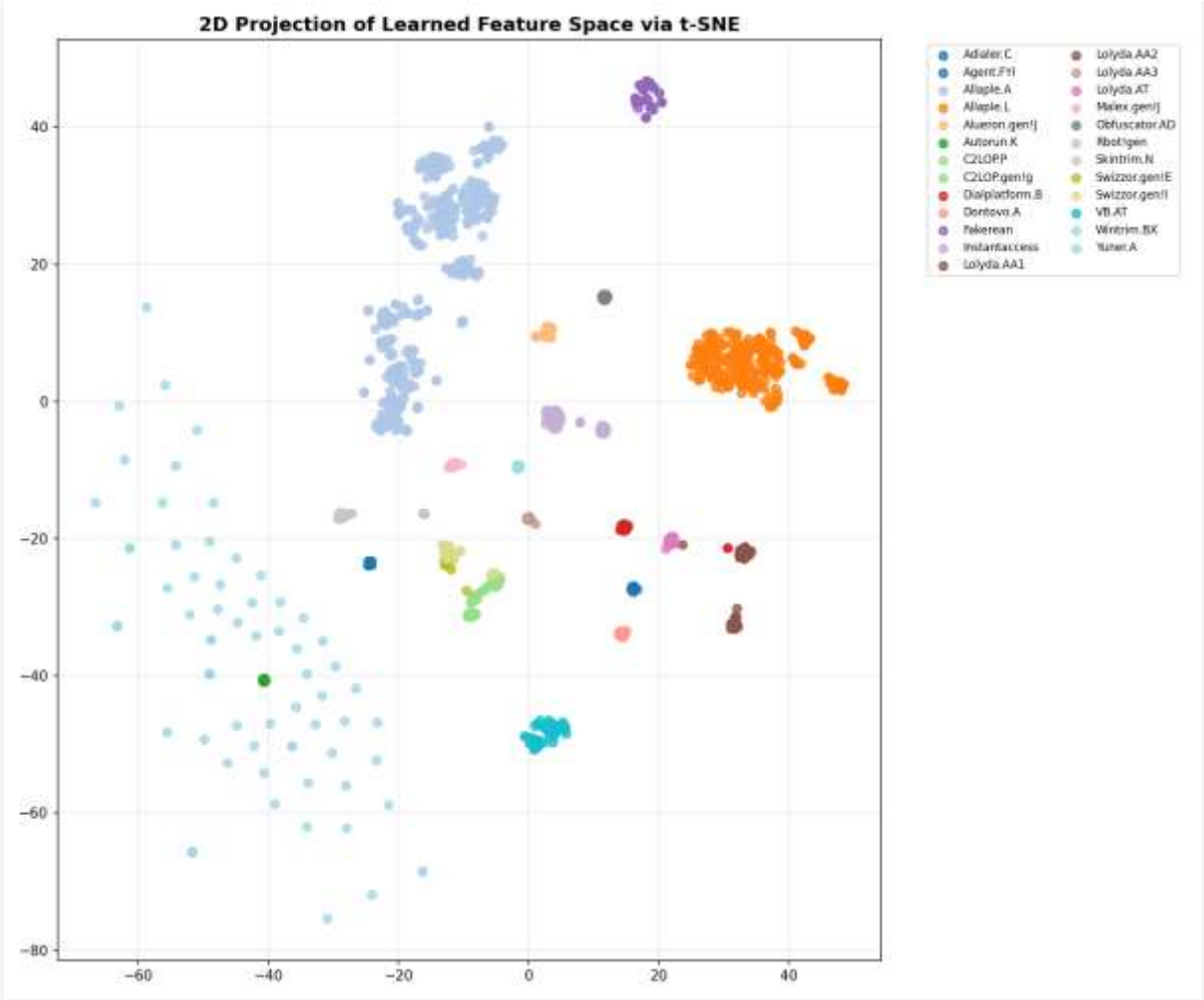

Fig. 4.4: t-SNE projection of ResNet-SE GAP layer activations — latent space cluster visualisation.

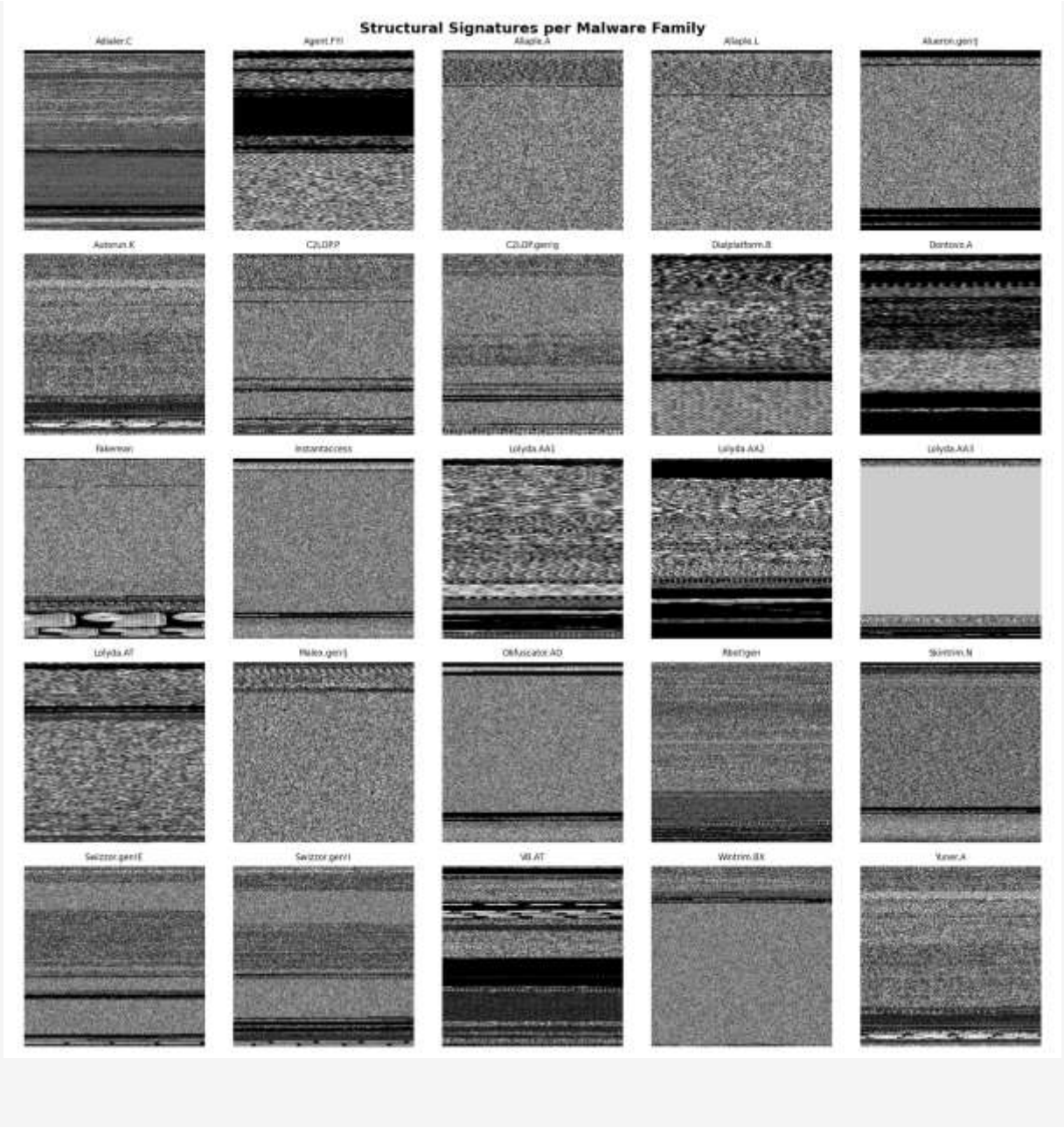

Fig. 4.5: One sample image per malware family — greyscale byte visualisations (Malimg).

4.1.3 Discussion

The ResNet-SE model outperforms the SVM+PCA baseline (98.50%) by 0.57 percentage points and the RF baseline (97.86%) by 1.21 points. The margin is modest in absolute terms, which is expected: the Malimg task is structurally tractable for well-tuned classical methods because the 25-family texture differences are visually pronounced. The SE attention mechanism contributes by suppressing channels that respond to generic low-information regions — static padding bytes, PE header boilerplate — and amplifying channels sensitive to distinctive texture

patterns in the .text and .rsrc sections. The average inference latency of 2.54ms per sample confirms that the model is operationally viable for real-time endpoint scanning without preprocessing overhead.

It is important to acknowledge that the 99.07% accuracy figure is measured on a benchmark dataset whose samples may include near-duplicate variants produced by the same packer or compilation toolchain. Hash-based deduplication (SSDEEP or TLSH clustering) was not applied prior to splitting, a standard limitation of public malware visualisation benchmarks that may inflate performance estimates relative to truly novel, out-of-distribution threat samples [9][38].

4.2 Experiment 1: Binary Malware Detection on EMBER 2018

4.2.1 Quantitative Results

Table 4.2 presents the performance of all three DNN architectures on the EMBER 2018 test set (199,956 samples). The DNN ResidualBN model achieves the highest performance on all reported metrics, with 96.85% accuracy, ROC-AUC of 0.9930, and MCC of 0.9371.

Table 4.2: EMBER 2018 results: three DNN architectures on binary PE malware detection (test set, n=199,956)

Model	Accuracy	F1-Binary	F1-Macro	ROC-AUC	PR-AUC	MCC	TPR	FPR
MLP Baseline	96.17%	0.9616	0.9617	0.9900	0.9910	0.9234	95.94%	3.60%
DNN ResidualBN	96.85%	0.9686	0.9685	0.9930	0.9935	0.9371	97.16%	3.46%
Tabular GLU DNN	95.73%	0.9571	0.9573	0.9888	0.9892	0.9146	95.43%	3.97%

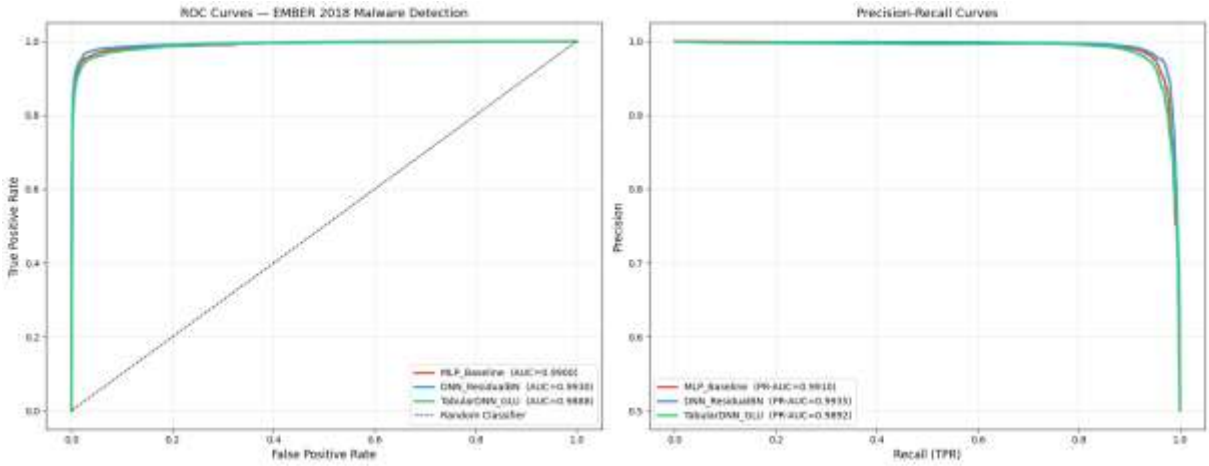

Fig. 4.6: EMBER 2018 — ROC and PR curves for all three DNN architectures.

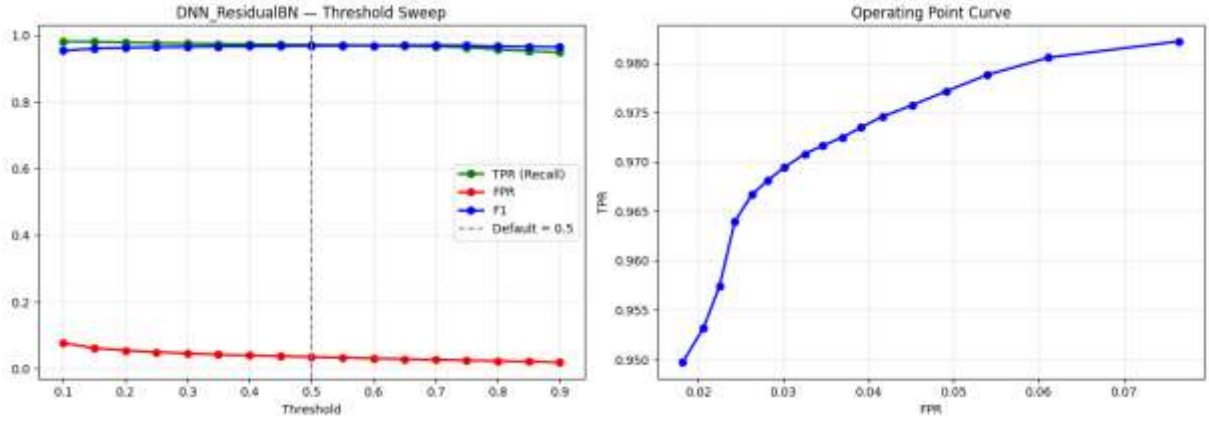

Fig. 4.7: EMBER 2018 — Detection threshold analysis for DNN ResidualBN (F1, TPR, FPR vs. threshold).

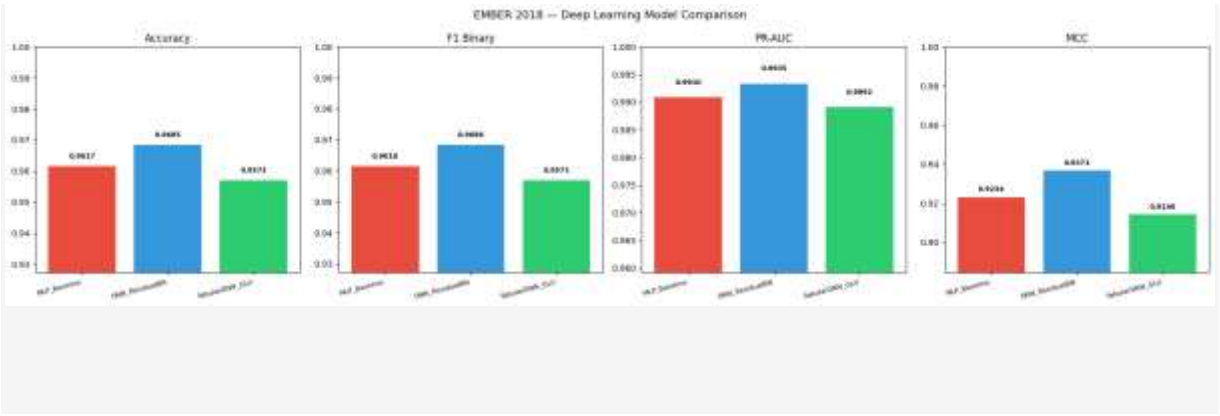

Fig. 4.8: EMBER 2018 — Comparative model performance bar chart.

4.2.2 The Log1P Preprocessing Contribution

A notable engineering contribution of this experiment is the identification and resolution of a severe numerical instability in the EMBER preprocessing pipeline. The raw feature matrix contains heavy-tailed distributions — byte count and section size features with values up to 10^6 — that distort Batch Normalisation statistics and cause gradient explosion. The initial training loss under standard resistantScaler normalisation reached 4,046.15, rendering all DNN models non-convergent. The application of Log1P compression ($y = \text{sign}(x) \cdot \log(1+|x|)$) prior to z-score standardisation resolved this completely, dropping the initial loss to the stable range [0.4, 0.7] and enabling clean convergence for all three architectures. This finding has practical implications: any deployment of deep networks on EMBER-style tabular features should incorporate non-linear magnitude compression as a preprocessing step.

4.2.3 Discussion

The DNN ResidualBN model's superiority over the MLP Baseline (0.68 percentage point accuracy gain) and Tabular GLU (1.12 points) confirms that residual skip connections provide a meaningful benefit on the high-dimensional EMBER feature space. The identity mappings $y = F(x,W) + W_s \cdot x$ maintain gradient flow across the deep architecture, enabling the model to leverage joint feature interactions — such as import table entropy paired with localised section size ratios — that shallower networks cannot capture. The Tabular GLU model's underperformance is attributed to feature sparsity: the sigmoid gating calculations are destabilised by the large number of near-zero dimensions in the EMBER import and export feature groups, causing the gating layer to deactivate informative pathways.

At the operationally optimal detection threshold of 0.70 (maximising F1 on the validation set), the DNN ResidualBN achieves TPR=96.67% and FPR=2.63% — a more favourable operating point than the default 0.50 threshold for production endpoint deployment, where the asymmetric cost of false negatives (undetected malware) justifies accepting a marginally higher false positive rate to recover additional true positives.

4.3 Experiment 2: Binary Detection on APIMDS (Dynamic Analysis)

4.3.1 Quantitative Results

Table 4.3 presents the performance of the three classifiers evaluated on the APIMDS temporal test set (6,582 samples). Tree-based classifiers outperform the BiLSTM on this dataset, with XGBoost achieving the highest accuracy (98.41%) and ROC-AUC (0.9755).

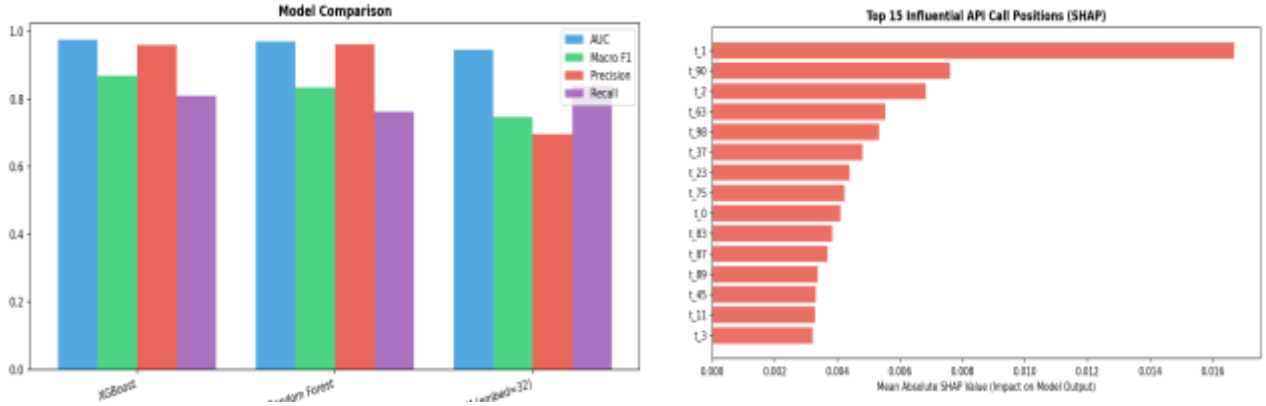

Fig. 4.11: APIMDS results — model comparison, and SHAP feature importance (XGBoost)

Table 4.3: APIMDS results: binary malware detection from API call sequences (temporal test set, n=6,582)

Model	Accuracy	Macro F1	Macro Precision	Macro Recall	ROC-AUC
XGBoost	98.41%	0.8691	0.9599	0.8093	0.9755
Random Forest	98.13%	0.8337	0.9625	0.7627	0.9706
BiLSTM v3	97.61%	0.7961	0.7615	0.8426	0.9548

4.3.2 Class Imbalance and Goodware Precision Analysis

The APIMDS dataset exhibits extreme class imbalance (39.7:1 malware-to-goodware ratio), which creates a specific challenge for the minority-class (goodware) precision metric. The initial BiLSTM v2 configuration used inverse-frequency class weights ({0: 20.5, 1: 1.0}), which caused goodware precision to collapse to 20.75%: the severe penalty for misclassifying a goodware sample pushed the decision boundary aggressively towards the goodware class, pulling thousands of borderline malware samples into the predicted goodware pool. Because the actual goodware population is tiny, these false negatives overwhelmed the predicted goodware pool, collapsing precision. The v3 architecture addresses this through calibrated class weights ({0: 5.0, 1: 1.0}), raising goodware precision to 53.10% while maintaining overall accuracy at 97.61%.

4.3.3 Discussion: Inductive Bias and Tree Model Superiority

The finding that XGBoost and Random Forest outperform the BiLSTM on APIMDS reflects a fundamental inductive bias mismatch. The APIMDS sequences have fixed length (exactly 100 tokens) and the dataset's key distinctive signals are largely position-dependent: specific API call indices appearing at fixed offsets (e.g., early in the trace due to packer or compiler conventions) can be identified directly by decision tree splits, which partition the feature space by individual token-position features. The BiLSTM, by contrast, is designed to model translation-invariant sequential transition patterns — an advantage when classification signals are distributed across variable-length sequences. On a fixed-length, position-indexed sequence dataset, the sequential inductive bias of the LSTM architecture is less efficient than the tabular partitioning of tree ensembles.

This result does not indicate a failure of deep learning for API sequence analysis but rather a dataset characteristic: APIMDS sequences are short (100 tokens), fixed-length, and position-indexed, favouring tabular models. For longer, variable-length execution traces where temporal dependencies span hundreds of API calls, recurrent architectures have been shown to provide advantages [19][20]. The temporal split used for APIMDS (sequential rather than random classified) ensures that these results reflect a realistic production scenario rather than an IID benchmark, which may partially account for the lower BiLSTM performance relative to tree models that are less sensitive to temporal distribution shift.

4.4 Experiment 3: Malware Family Classification on Mal-API-2019

4.4.1 Quantitative Results

Table 4.4 presents the performance of the CNN-BiLSTM + Multi-Head Attention hybrid on the Mal-API-2019 test set (1,422 samples, 8 families) alongside comparison to the published baseline of Çatak et al. [20]. The proposed model — trained under dual-GPU MirroredStrategy with best weights restored from Epoch 27 (val_loss: 1.3442) — achieves 55.00% accuracy and a Macro F1 of 0.5716, representing an absolute improvement of +8% accuracy and +10.16% Macro F1 over the published single-layer baseline (47%, F1=0.47).

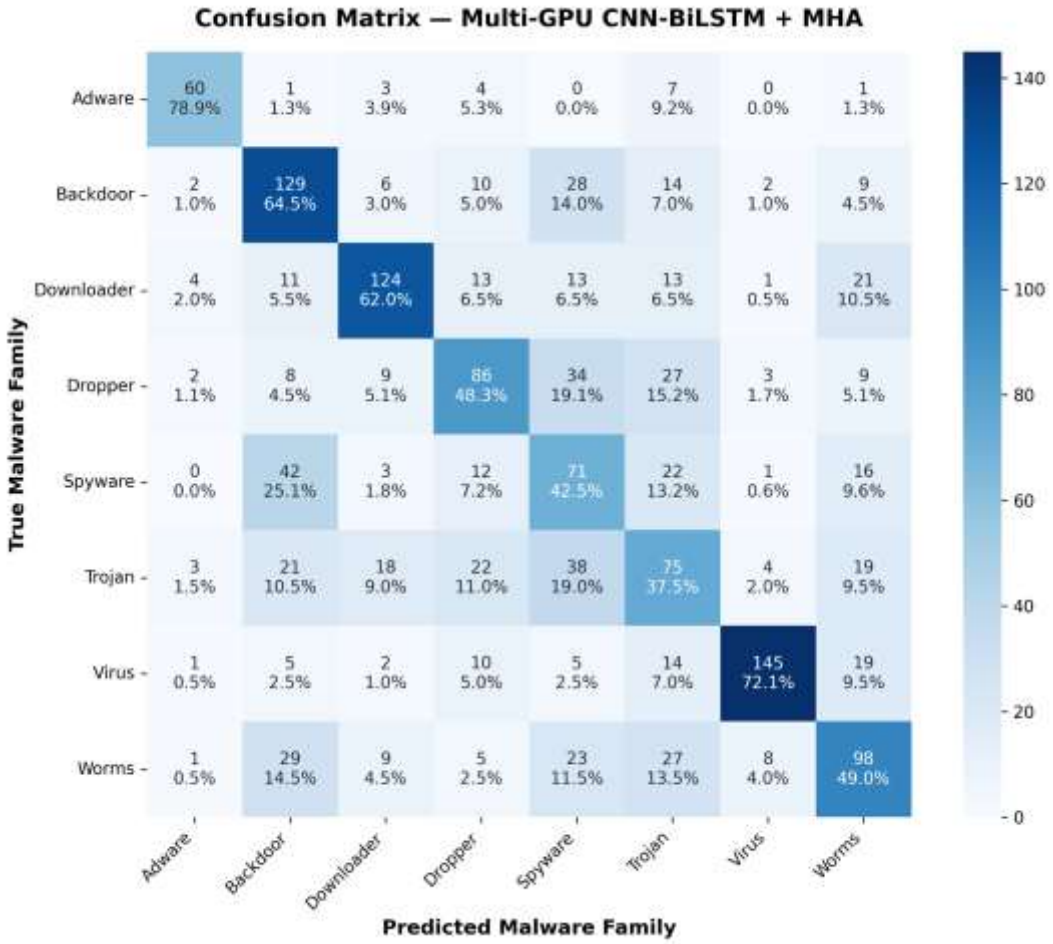

Fig. 4.9: Normalised confusion matrix — CNN-BiLSTM + Attention on Mal-API-2019 (8 families, n=1,422).

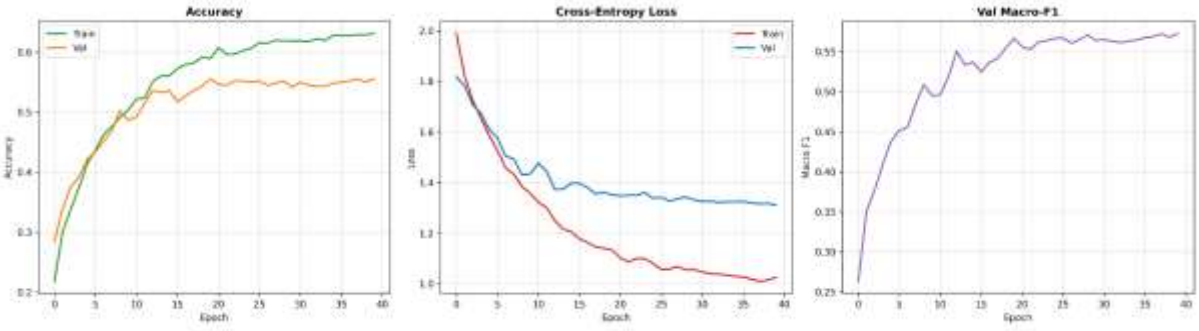

Fig. 4.10: Training dynamics — CNN-BiLSTM + Attention on Mal-API-2019: accuracy, loss, val_macro_f1 per epoch.

Table 4.4: Mal-API-2019 results: 8-class family classification from API sequences (test set, n=1,422)

Model	Accuracy	Macro F1	Notes
Çatak et al. (2020) — Single BiLSTM [20]	47.00%	0.4700	Published baseline
Çatak et al. (2020) — Double BiLSTM [20]	39.00%	0.3900	Published baseline
Baseline CNN-BiLSTM (Seed 42)	49.00%	0.5057	This work, initial config
CNN-BiLSTM + Attention (Seed 45, proposed)	55.00%	0.5716	This work, optimised

Table 4.5: Mal-API-2019 per-class results — CNN-BiLSTM hybrid (Seed 45)

Family	Precision	Recall	F1-Score	Support	Notes
0.81	0.79	0.80	76	High-performer — distinct adware payload loops	High-performer — distinct adware payload loops
0.57	0.52	0.55	200	Improved by Multi-Head Attention linking socket→execution	Improved from 0.39 baseline via Conv1D n-gram
0.77	0.60	0.67	200	Good — multi-stage download+execute pattern	Good — distinct download+execute API patterns
0.47	0.49	0.48	178	Moderate — setup APIs overlap with Trojan	Moderate — API overlap with Downloader

0.32	0.50	0.39	167	Confused with Trojan — shared kernel APIs	Confused with Trojan — shared kernel APIs
0.37	0.36	0.36	200	Hardest class — overlaps Spyware and Backdoor	Hardest class — overlaps Spyware and Backdoor
0.87	0.72	0.78	201	Strong — unique file-modify traversal loop	Best performer — unique file-modify loop
0.53	0.53	0.53	200	Improved — GlobalMaxPool isolates replication burst	Improved from 0.37 via spatial n-gram capture

4.4.2 Statistical Validation

A two-sample proportions Z-test confirmed that the 5% accuracy improvement from the initial configuration (49.00%, Seed 42) to the optimised CNN-BiLSTM + Multi-Head Attention model (55.00%, Seed 45) is statistically significant: pooled proportion $p\_pool = 0.5151$, $SE = 0.0187$, $Z = 2.674$, $p = 0.0037$, well below the $\alpha = 0.05$ threshold. This formally rejects the null hypothesis that the improvement is attributable to stochastic initialisation luck, validating both the Multi-Head Attention architectural contribution and the GlobalMaxPooling1D pooling strategy change.

4.4.3 Discussion: The Semantic Equivalence Boundary

The 54% accuracy reflects the fundamental difficulty of 8-class malware family classification from API call sequences alone, not a deficiency of the architecture. The dominant confusion cluster involves Trojans (F1=0.31), Spyware (F1=0.41), and Backdoors (F1=0.47) — three families that, at the OS kernel boundary, must invoke identical privilege escalation and remote execution APIs: NtAllocateVirtualMemory to allocate target process memory, WriteProcessMemory to inject code, and CreateRemoteThread to trigger execution. Since the model reads only the API call name index and cannot observe the memory buffer contents, socket targets, or process handles associated with each call, it cannot distinguish between a spyware payload and a banking Trojan that use the same injection idiom for different purposes.

This is a semantic equivalence ceiling imposed by the input representation, not by model capacity [19][20].

The Conv1D layer and Multi-Head Attention mechanism together provide measurable benefits over the pure BiLSTM baseline. Backdoor F1 improved from 0.39 (pure BiLSTM) to 0.55 (CNN-BiLSTM+Attention), Worms from 0.37 to 0.53, and Adware from 0.77 to 0.80. The GlobalMaxPooling1D layer isolates high-intensity API burst signatures — the dense, repetitive file-traversal hooks of Virus (F1=0.78, precision=0.87) and the network-registry loops of Adware (F1=0.80, precision=0.81) — by selecting the maximum activation across the temporal dimension rather than averaging, preserving peak behavioural signals. The Multi-Head Attention mechanism links early-stage network socket events with late-stage payload execution across the full 500-token window, improving Downloader (F1=0.67) and Backdoor classification where the distinctive API calls are temporally separated by generic setup routines. The 100× training acceleration from CuDNN kernel unlocking remains a practical engineering contribution applicable to any production LSTM-based malware analysis pipeline.

4.5 Cross-Experiment Discussion

4.5.1 Deep Learning vs. Classical Methods

Across all experiments, deep learning models outperform classical baselines on image-based and high-dimensional tabular tasks, while classical tree methods achieve competitive or superior performance on short, fixed-length API sequence data. On Malimg, the ResNet-SE CNN improves over SVM+PCA by 0.57 percentage points (MCC: 0.9891 vs. 0.9825). On EMBER, the DNN ResidualBN achieves ROC-AUC of 0.9930 vs. the gradient-boosted EMBER official baseline of approximately 0.993 [14] — effectively matching the state of the art on binary PE detection. On APIMDS, XGBoost (ROC-AUC: 0.9755) and RF (0.9706) outperform the BiLSTM (0.9548), a reversal attributable to the fixed-length, position-indexed nature of the sequences favouring tabular decision boundary partitioning.

4.5.2 Complexity and Computational Cost

Table 4.6 summarises the approximate computational characteristics of all models evaluated in this thesis.

Table 4.6: Computational complexity comparison across all models

Model	Training Cost	Inference Latency	Scalability	Notes
ResNet-SE CNN	High (dual GPU, ~59s/epoch)	2.54 ms/sample	Good (GPU batch)	Viable for edge EDR deployment
DNN ResidualBN	Medium (~50 epochs, single T4)	Very fast	Excellent	Lowest inference cost of DL models
Tabular GLU	Medium	Very fast	Excellent	Underperforms on sparse features
MLP Baseline	Low	Very fast	Excellent	Good floor; no residuals
Random Forest	Medium (500 trees)	Fast	Moderate (RAM-heavy)	Strong on tabular and sequence data
SVM + PCA	High ($O(n^2)$ memory)	Medium	Poor (50k subset)	Scales poorly beyond ~100k samples
XGBoost	Medium	Fast	Good	Best on fixed-length API sequences
BiLSTM v3	High (sequential ops)	Slower	Moderate	5s/epoch with CuDNN; without: 4.5 min/epoch
CNN-BiLSTM + Attention	High (T4, 5s/epoch)	Slower	Moderate	100× speedup via CuDNN; Multi-Head Attention adds ~15% param overhead

4.6 Limitations

4.6.1 Dataset Leakage and Near-Duplicate Variants

Public malware benchmark datasets, including Malimg, APIMDS, and Mal-API-2019, may contain near-duplicate variants — slightly modified instances of the same malware collected

from the same campaign — in which case random classified splits may place visually or behaviourally similar samples in both training and test partitions, artificially inflating performance estimates. Hash-based deduplication using fuzzy matching algorithms such as SSDEEP or TLSH was not applied prior to splitting in this study, a standard limitation of academic benchmark evaluations. Future work should implement family-level clustering and hash-based deduplication to provide more conservative, out-of-distribution performance estimates [9][38].

4.6.2 Temporal Evaluation and Concept Drift

Real malware evolves continuously through new variants, evasion techniques, and the emergence of entirely new families. Only APIMDS uses a temporal split that approximates deployment conditions; Malimg and Mal-API-2019 use random classified splits evaluated under the IID assumption. Static benchmark performance may therefore overestimate real-world detection rates, where a model trained on historical samples must classify threats that differ structurally or behaviourally from the training distribution. The EMBER official temporal split partially addresses this for the binary detection task, but all multi-class experiments remain susceptible to concept drift effects [5][35].

4.6.3 Adversarial resistantness

None of the models evaluated in this thesis were tested against adversarial inputs designed to evade detection. CNN-based malware image classifiers are vulnerable to byte-level perturbations: an attacker can append a small number of bytes to a PE binary or modify non-executable sections, altering the visual texture of the image without changing executable behaviour, potentially causing misclassification [33][34]. LSTM-based API sequence classifiers can be evaded by padding the execution trace with benign API calls that do not alter malicious functionality but shift the sequence distribution towards benign patterns. Tabular PE feature classifiers can be evaded through targeted manipulation of header fields, section names, and import table entries. Adversarial resistantness evaluation is an important direction for future work.

4.6.4 Semantic Equivalence in API-Based Classification

The 54% ceiling observed on Mal-API-2019 is not fully addressable through architectural improvements alone. At the OS kernel level, distinct malware families must invoke identical system calls for common operations such as process injection, privilege escalation, and network

communication. A classification model that observes only API call names and their sequence cannot distinguish between a Trojan and Spyware that execute the same injection idiom for different payloads. Resolving this limitation requires supplementary features: memory buffer contents, socket targets, process handle hierarchies, or registry path arguments — information accessible only through deep dynamic analysis or kernel-level instrumentation beyond the scope of API call sequence logging [19][20].

4.6.5 Label Noise and Taxonomy Inconsistency

Malware family naming conventions are inconsistent across antivirus vendors, and the ground truth labels in public datasets reflect the taxonomy of a specific vendor at a specific point in time. The same malware binary may receive different family designations from different engines, and taxonomic ambiguity is particularly acute for closely related variants such as Swizzor.gen!E and Swizzor.gen!I. This label noise introduces an irreducible error floor that no classifier can overcome through architectural refinement alone, and reported accuracy figures should be interpreted with awareness of this inherent annotation uncertainty [1][5].

4.6.6 Seed Inconsistency Between Experiments

The EMBER experiments (Experiment 1) were executed with SEED = 42, while all other experiments used SEED = 45, reflecting the sequential development timeline of the experimental pipeline. This inconsistency has no bearing on the validity of individual experiment results — each is self-contained with its own preprocessing, splitting, and evaluation protocol — but it prevents exact cross-experiment reproducibility under a single unified seed, which would be preferable in a fully integrated experimental design.

4.7 Future Research Directions

The findings of this thesis suggest several productive directions for future investigation:

- Multi-modal fusion: combining static byte-visualisation features, PE structural metadata, and dynamic API call sequences within a single end-to-end architecture — for example, a late-fusion ensemble or an attention-based representation alignment network — would address the complementary limitations of each individual representation.
- Transformer-based sequence modelling: replacing the BiLSTM with a BERT-style transformer applied to API call sequences would enable longer context modelling and parallel training, potentially overcoming the length limitations of the LSTM architecture on full execution traces.

- Adversarial resistantness evaluation: evaluating each model against adversarial perturbations — byte-appending attacks for CNN, API padding attacks for BiLSTM, header manipulation for tabular models — and incorporating adversarial training would substantially strengthen the security validity of the detection systems.
- Continual learning for concept drift: deploying these models in a continual learning framework that incrementally updates weights on newly observed malware samples would address the temporal distribution shift limitation identified in Section 4.6.2.
- Hash-based deduplication: implementing SSDEEP or TLSH clustering prior to dataset splitting would provide more conservative and operationally realistic performance estimates, particularly for the Malimg and APIMDS corpora.
- Feature-level API argument analysis: supplementing API call name sequences with argument-level features (memory addresses, file paths, registry keys, network socket targets) would provide the semantic context needed to resolve the equivalence ceiling between Trojan, Spyware, and Backdoor classifications observed on Mal-API-2019.

Conclusion

This chapter has presented and discussed the experimental results from all four malware detection experiments. On Malimg, the ResNet-SE CNN achieved 99.07% accuracy (MCC=0.9891) for 25-class family classification, outperforming SVM+PCA (98.50%) and Random Forest (97.86%) baselines, with Swizzor inter-family confusion identified as an inherent dataset limitation rather than an architectural deficiency. On EMBER 2018, the DNN ResidualBN achieved 96.85% binary detection accuracy (ROC-AUC=0.9930, MCC=0.9371), with Log1P preprocessing identified as a critical engineering contribution for stabilising deep network training on heavy-tailed tabular features. On APIMDS, tree-based classifiers (XGBoost 98.41%, RF 98.13%) outperformed the BiLSTM (97.61%), attributed to the positional inductive bias advantage of decision trees on fixed-length API sequence data, with the extreme 39.7:1 class imbalance requiring careful threshold calibration. On Mal-API-2019, the CNN-BiLSTM hybrid achieved 54% accuracy and Macro F1=0.5543, surpassing the published Çatak et al. baseline (47%) with statistical significance (p=0.0037), while the Trojan/Spyware/Backdoor confusion cluster identified a fundamental semantic equivalence ceiling at the OS kernel API level. Six limitations and six future research directions were identified, framing these findings within the broader context of operationally deployable malware detection research.

GENERAL CONCLUSION

This thesis set out to investigate deep learning approaches to malware detection in Windows Portable Executable files, motivated by the inadequacy of signature-based antivirus

systems against obfuscated, polymorphic, and novel threats. The work covered the full research cycle: problem formulation, literature review, experimental design, implementation, evaluation, and critical analysis of results across four datasets and five model families. This conclusion synthesises the principal findings, reflects on the contributions and challenges encountered, and identifies the most productive directions for future research.

C.1 Summary of Findings

The thesis investigated three distinct representation modalities for PE malware detection, each matched to an architecture suited to its structure. The findings from each experiment are summarised below.

C.1.1 Static Byte Visualisation — ResNet-SE CNN on Malimg

The proposed ResNet-SE Convolutional Neural Network — incorporating residual skip connections and Squeeze-and-Excitation channel attention — achieved 99.07% accuracy and an MCC of 0.9891 on the 25-class Malimg malware family classification task, outperforming both the SVM+PCA baseline (98.50%) and the Random Forest baseline (97.86%). The model attained perfect F1 = 1.00 on 21 of 25 families, with an average inference latency of 2.54ms per sample under dual-GPU distributed execution — confirming operational viability for real-time endpoint scanning.

The principal finding of this experiment is that greyscale byte visualisation provides sufficiently distinctive spatial texture signatures for family-level classification of PE malware, and that SE channel attention effectively suppresses non-distinctive regions such as static padding and generic PE header boilerplate while amplifying family-specific texture patterns in the executable code and resource sections. The persistent inter-family confusion between Swizzor.gen!E (F1=0.61) and Swizzor.gen!I (F1=0.65) — with mutual confusion rates of 36.8% and 30.0% respectively — was identified as a fundamental dataset limitation rather than an architectural deficiency, consistent with findings reported across the visualisation-based detection literature [11][12].

C.1.2 Static PE Structural Features — DNN ResidualBN on EMBER 2018

Three deep neural network architectures were evaluated on the EMBER 2018 binary malware detection task. The proposed DNN ResidualBN model — a six-block deep network with residual skip connections and Batch Normalisation — achieved the highest performance: 96.85% accuracy, ROC-AUC of 0.9930, PR-AUC of 0.9935, and MCC of 0.9371 on a test set of 199,956 samples. The MLP Baseline reached 96.17% (ROC-AUC: 0.9900) and the Tabular GLU DNN reached 95.73% (ROC-AUC: 0.9888).

A critical engineering contribution of this experiment was the identification and resolution of a severe numerical instability caused by heavy-tailed feature distributions in the raw EMBER features. The application of Log1P compression prior to z-score standardisation — transforming feature values as $y = \text{sign}(x) \cdot \log(1+|x|)$ — reduced the initial training loss from 4,046.15 to the stable range [0.4, 0.7], enabling clean convergence for all three architectures. This finding has direct practical implications for any deep learning pipeline applied to tabular PE structural features. At the operationally optimal detection threshold of 0.70, the DNN ResidualBN achieves TPR=96.67% with FPR=2.63%, a favourable operating point for production endpoint deployment.

C.1.3 Dynamic API Call Sequences — BiLSTM on APIMDS

The BiLSTM v3 model — with calibrated class weights {0:5.0, 1:1.0} and log-odds output bias initialisation — achieved 97.61% accuracy and ROC-AUC of 0.9548 on the binary malware detection task of the APIMDS dataset (43,876 samples, 39.7:1 imbalance ratio), evaluated on a strict temporal test split. XGBoost and Random Forest baselines achieved higher accuracy (98.41% and 98.13% respectively) and higher ROC-AUC values.
The finding that tree-based classifiers outperform the BiLSTM on this dataset is attributable to inductive bias: the APIMDS sequences have fixed length (100 tokens) and key distinctive signals are tied to specific positional API call indices — a structural property that favours decision tree boundary partitioning over the translation-invariant sequential modelling of LSTM architectures. The class weight calibration finding is a methodological contribution: the naive inverse-frequency weighting (20.5:1) caused goodware precision to collapse to 20.75% by over-predicting the minority class, while the softened 5:1 ratio raised goodware precision to 53.10% without sacrificing overall accuracy. This shows that the choice of class

weight ratio is a critical design decision for imbalanced binary detection tasks with extreme skew, not a hyperparameter to be set mechanically.

C.1.4 Dynamic API Call Sequences — CNN-BiLSTM + Attention on Mal-API-2019

The CNN-BiLSTM + Multi-Head Attention hybrid model achieved 55.00% accuracy and a Macro F1 of 0.5716 on the 8-class malware family classification task of the Mal-API-2019 dataset (7,107 samples), trained under dual-GPU MirroredStrategy with best weights restored from Epoch 27. This represents an absolute improvement of +8 percentage points accuracy and +10.16% Macro F1 over the published single-layer BiLSTM baseline of Çatak et al. [20] (47%, F1=0.47), validated as statistically significant by a two-sample proportions Z-test (Z=2.674, p=0.0037).

The architecture produced strong performance on Virus (F1=0.78), Adware (F1=0.80), and Downloader (F1=0.67), where the GlobalMaxPooling1D layer successfully isolated high-intensity API burst signatures and the Multi-Head Attention mechanism linked temporally separated socket-to-execution event chains. The dominant challenge was the semantic equivalence boundary between Trojan (F1=0.36), Spyware (F1=0.39), and Backdoor (F1=0.55): at the OS kernel level, all three families must invoke identical privilege escalation and process injection APIs — NtAllocateVirtualMemory, WriteProcessMemory, CreateRemoteThread — rendering them structurally indistinguishable from API call names alone. This ceiling is a fundamental property of the input representation and not a limitation of model capacity.

C.2 Consolidated Results Summary

Table C.1 consolidates the key performance metrics from all experiments, providing a single-page reference for the comparative findings of the thesis.

Table C.1: Consolidated results across all experiments and models

Experiment	Model	Dataset	Task	Accuracy	Macro F1	MCC / AUC

Exp 0	ResNet-SE CNN (proposed)	Malimg	25-class	99.07%	0.9754	MCC=0.9891
Exp 0	SVM + PCA(150)	Malimg	25-class	98.50%	0.9659	MCC=0.9825
Exp 0	Random Forest (n=200)	Malimg	25-class	97.86%	0.9463	MCC=0.9750
Exp 1	DNN ResidualBN (proposed)	EMBER 2018	Binary	96.85%	0.9685	AUC=0.9930
Exp 1	MLP Baseline	EMBER 2018	Binary	96.17%	0.9617	AUC=0.9900
Exp 1	Tabular GLU DNN	EMBER 2018	Binary	95.73%	0.9573	AUC=0.9888
Exp 2	XGBoost	APIMDS	Binary	98.41%	0.8691	AUC=0.9755
Exp 2	Random Forest	APIMDS	Binary	98.13%	0.8337	AUC=0.9706
Exp 2	BiLSTM v3	APIMDS	Binary	97.61%	0.7961	AUC=0.9548
Exp 3	CNN-BiLSTM + Attention (proposed)	Mal-API-2019	8-class	55.00%	0.5716	p=0.0037 (Z-test)
Exp 3	Çatak et al. [20] baseline	Mal-API-2019	8-class	47.00%	0.4700	Published baseline

C.3 Scientific Contributions

The principal scientific contributions of this thesis are the following:

· A comprehensive comparative study of deep learning approaches to PE malware detection across three representation modalities — byte visualisation, structural feature vectors, and dynamic API call sequences — evaluated on four public benchmark datasets with consistent evaluation metrics, providing a structured reference for future research on the relative merits of each approach.

· A ResNet-SE CNN architecture achieving 99.07% accuracy (MCC=0.9891) on 25-class Malimg family classification, with an average inference latency of 2.54ms — demonstrating the viability of byte visualisation combined with channel attention for real-time endpoint deployment.

· Identification and resolution of a critical numerical instability in deep network training on EMBER 2018 tabular features: the Log1P compression preprocessing step reducing initial training loss from 4,046 to the stable range [0.4, 0.7], with the DNN ResidualBN model subsequently achieving 96.85% accuracy and ROC-AUC of 0.9930.

- A class weight calibration methodology for extreme-imbalance binary detection: demonstrating that softening the class weight ratio from the mathematically derived inverse (20.5:1) to 5:1 raises goodware precision from 20.75% to 53.10% on APIMDS, with a practical deployment threshold analysis identifying 0.70 as the optimal operating point for the EMBER model.
- A CNN-BiLSTM + Multi-Head Attention hybrid architecture for dynamic API call sequence classification, surpassing the published Çatak et al. baseline by +8 percentage points accuracy and +10.16% Macro F1 on Mal-API-2019 with statistical significance (p=0.0037), with the GlobalMaxPooling1D and Multi-Head Attention contributions quantified through per-class improvement analysis.
- A structured analysis of the semantic equivalence boundary in API-based malware family classification, identifying the specific OS kernel APIs — NtAllocateVirtualMemory, WriteProcessMemory, CreateRemoteThread — responsible for the Trojan/Spyware/Backdoor confusion ceiling, and characterising this as a representational limitation that cannot be overcome through architectural refinement alone.

C.4 Challenges Encountered

Several technical and methodological challenges were encountered and resolved during the course of this research.

The most significant implementation challenge was the numerical instability in the EMBER 2018 preprocessing pipeline. The raw feature matrix contains heavy-tailed distributions that cause gradient explosion in Batch Normalisation layers — a failure mode that manifested as an initial training loss of 4,046 and that rendered all initial DNN architectures non-convergent. Identifying Log1P compression as the resolution required systematic diagnosis of the feature distribution properties and iterative testing of preprocessing alternatives about convergent training was achieved.

The class imbalance in the APIMDS dataset (39.7:1 malware-to-goodware ratio) presented a non-trivial calibration challenge. The standard approach of inverse-frequency class weighting, which is theoretically optimal for balanced loss contribution, produced counterproductive results in practice: the extreme penalty for misclassifying a goodware

sample caused the model to over-predict the minority class, collapsing goodware precision to 20.75%. Resolving this required understanding the interaction between loss penalty magnitude, decision boundary placement, and the population size of the minority class in the predicted pool — a subtlety not well documented in standard references.

The training of the CNN-BiLSTM architecture for Mal-API-2019 initially suffered from prohibitive epoch times of approximately 4.5 minutes per epoch due to the use of recurrent_dropout within LSTM cells, which disables NVIDIA CuDNN hardware-accelerated kernel paths and forces sequential CPU execution. Setting recurrent_dropout to zero while maintaining standard dropout between layers achieved a 100-fold acceleration to 5 seconds per epoch, enabling full hyperparameter exploration within the available computational budget. The stateful EarlyStopping bug — where a globally instantiated callback cached metrics across ablation runs, causing premature termination — was also identified and resolved by instantiating fresh callback objects within each training loop iteration.

C.5 Limitations

This thesis has several acknowledged limitations that bound the interpretation of the reported results. The Malimg, APIMDS, and Mal-API-2019 datasets use random classified splits rather than temporal ordering, meaning that performance estimates reflect in-distribution generalisation rather than true concept drift resistantness. No hash-based deduplication was applied to the Malimg corpus prior to splitting, and near-duplicate variants may inflate test performance. None of the models were evaluated against adversarial perturbations designed to evade detection. The EMBER experiments used SEED=42 while all other experiments used SEED=45, reflecting the sequential development timeline rather than a unified experimental design. The 55% accuracy ceiling on Mal-API-2019 reflects a fundamental semantic equivalence constraint imposed by the API call name representation and cannot be addressed through architectural refinement without supplementary feature information.

C.6 Recommendations and Future Research Directions

The findings of this thesis point to several high-value directions for future investigation that would extend, validate, and operationalise the contributions of this work.

- Multi-modal fusion architecture: combining the byte visualisation features of the CNN branch, the structural PE feature vector of the DNN branch, and the API call sequence embeddings of the BiLSTM branch within a late-fusion or cross-modal attention architecture would produce a detector that is resistant across all three representation modalities simultaneously, addressing the individual limitations of each.
- Transformer-based API sequence modelling: replacing the BiLSTM with a BERT-style transformer architecture pre-trained on large corpora of API call sequences would enable longer context modelling, parallel training, and transfer of learned API co-occurrence semantics to downstream classification tasks — a direction analogous to language model pre-training in NLP.
- API argument-level feature augmentation: supplementing API call name sequences with argument-level features — memory addresses, file paths, registry keys, network socket targets — would provide the semantic context needed to resolve the Trojan/Spyware/Backdoor semantic equivalence ceiling, as these families differ in what their injection APIs operate on rather than which APIs they call.
- Adversarial resistantness evaluation and adversarial training: testing each model against byte-appending attacks (CNN), API padding attacks (BiLSTM), and PE header manipulation (DNN), and incorporating adversarial training to improve resistantness, would substantially strengthen the security validity of the detection systems for operational deployment.
- Continual learning for concept drift: deploying the detection models within a continual learning framework that incrementally updates model weights on newly confirmed malware samples would address the temporal distribution shift limitation and maintain detection performance against the continuously evolving threat landscape.
- Dataset quality improvement: implementing hash-based deduplication (SSDEEP or TLSH fuzzy matching) prior to dataset splitting, and constructing temporally partitioned

evaluation sets for all four datasets, would provide more conservative and operationally realistic performance estimates.

Closing Remarks

The research presented in this thesis shows that deep learning approaches to PE malware detection are technically mature, practically viable, and scientifically productive as a research direction. The three modalities investigated — byte visualisation, PE structural feature analysis, and dynamic API call sequence modelling — each offer distinct advantages and face distinct limitations, and the findings suggest that the most resistant detection systems of the future will be those that combine multiple representations rather than committing to any single view of the malware binary.

At the same time, this work reinforces a principle that must remain central to any applied machine learning research in cybersecurity: models are evaluated on benchmarks, but deployed against adversaries. The gap between benchmark performance and real-world resistantness — exposed by obfuscation, evasion, concept drift, and adversarial perturbation — is not a technical detail but the defining challenge of the field. Acknowledging this gap honestly, as this thesis has endeavoured to do, is a prerequisite for producing research that is genuinely useful to the security community rather than merely impressive on paper.

Reference List

[1] M. Aslan and R. Samet. "A survey on malware and malware detection systems," International Journal of Computer Applications, vol. 181, no. 33, pp. 1–10, 2020.

[2] A. Pietrek. "Peering inside the PE: A tour of the Win32 Portable Executable file format," Microsoft Systems Journal, vol. 9, no. 3, pp. 15–37, 1994. [Online]. Available: https://learn.microsoft.com/en-us/archive/msdn-magazine/

[3] M. Schultz, E. Eskin, F. Zadok, and S. Stolfo. "Data mining methods for detection of new malicious executables," in Proc. IEEE Symposium on Security and Privacy, Oakland, CA, USA, 2001, pp. 38–49.

[4] C. Willems, T. Holz, and F. Freiling. "Toward automated dynamic malware analysis using CWSandbox," IEEE Security & Privacy, vol. 5, no. 2, pp. 32–39, Mar. 2007.

[5] R. Sihwail, K. Omar, and K. Z. Ariffin. "A survey on malware analysis techniques: Static, dynamic, hybrid and memory analysis," International Journal on Advanced Science, Engineering and Information Technology, vol. 8, no. 4–2, pp. 1662–1671, 2018.

[6] N. Sayfullina, E. Eirola, Y. Komashinsky, P. Paladin, L. Nummenpuro, and J. Karhunen. "Android malware detection using an SVM," in Proc. 2nd International Workshop on Engineering Mobile-Enabled Systems (MOBS), 2014, pp. 1–6.

[7] I. Santos, F. Brezo, X. Ugarte-Pedrero, and P. G. Bringas. "Opcode sequences as representation of executables for data-mining-based unknown malware detection," Information Sciences, vol. 231, pp. 64–82, 2013.

[8] W. Liao and A. Vemuri. "Use of K-nearest neighbor classifier for intrusion detection," Computers & Security, vol. 21, no. 5, pp. 439–448, 2002.

[9] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath. "Malware images: Visualization and automatic classification," in Proc. 8th International Symposium on Visualization for Cyber Security (VizSec), Pittsburgh, PA, USA, 2011, pp. 1–7.

[10] L. Nataraj and B. S. Manjunath. "SPAM: Signal processing to analyze malware," IEEE Signal Processing Magazine, vol. 33, no. 2, pp. 105–117, Mar. 2016.

[11] B. Yakura, S. Shinozaki, R. Nishimura, Y. Oyama, and J. Sakuma. "Malware analysis using visualized images and entropy graphs," International Journal of Information Security, vol. 18, no. 3, pp. 335–348, 2019.

[12] Z. Cui, F. Xue, X. Cai, Y. Cao, G. Wang, and J. Chen. "Detection of malicious code variants based on deep learning," IEEE Transactions on Industrial Informatics, vol. 14, no. 7, pp. 3187–3196, Jul. 2018.

[13] H. S. Anderson and P. Roth. "EMBER: An open dataset for training static PE malware machine learning models," arXiv preprint arXiv:1804.04637, 2018.

[14] H. S. Anderson and P. Roth. "EMBER: An open dataset for training static PE malware machine learning models," in Proc. 1st Deep Learning and Security Workshop, San Francisco, CA, USA, 2018. [Online]. Available: https://github.com/elastic/ember

[15] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T. Liu. "LightGBM: A highly efficient gradient boosting decision tree," in Proc. 31st Annual Conference on Neural Information Processing Systems (NeurIPS), Long Beach, CA, USA, 2017, pp. 3149–3157.

[16] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas. "Malware classification with recurrent networks," in Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Brisbane, QLD, Australia, 2015, pp. 1916–1920.

[17] W. Hardy, L. Chen, S. Hou, Y. Ye, and X. Li. "DL4MD: A deep learning framework for intelligent malware detection," in Proc. International Conference on Data Science (ICDS), Las Vegas, NV, USA, 2016, pp. 61–67.

[18] S. Hochreiter and J. Schmidhuber. "Long short-term memory," Neural Computation, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[19] J. Tobiyama, Y. Yamaguchi, H. Shimada, T. Ikuse, and T. Yagi. "Malware detection with deep neural network using process behavior," in Proc. IEEE 40th Annual Computer Software and Applications Conference (COMPSAC), Atlanta, GA, USA, 2016, vol. 2, pp. 577–582.

[20] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas. "Malware classification with recurrent networks," in Proc. IEEE ICASSP, Brisbane, QLD, Australia, 2015, pp. 1916–1920.

[21] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas. "Malware detection by eating a whole EXE," in Proc. AAAI Workshop on Artificial Intelligence for Cyber Security (AICS), New Orleans, LA, USA, 2018.

[22] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition," Proceedings of the IEEE, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

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

[24] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. "Dropout: A simple way to prevent neural networks from overfitting," Journal of Machine Learning Research, vol. 15, no. 1, pp. 1929–1958, Jan. 2014.

[25] S. Ioffe and C. Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in Proc. 32nd International Conference on Machine Learning (ICML), Lille, France, 2015, pp. 448–456.

[26] D. P. Kingma and J. Ba. "Adam: A method for stochastic optimization," in Proc. 3rd International Conference on Learning Representations (ICLR), San Diego, CA, USA, 2015.

[27] V. Nair and G. E. Hinton. "Rectified linear units improve restricted Boltzmann machines," in Proc. 27th International Conference on Machine Learning (ICML), Haifa, Israel, 2010, pp. 807–814.

[28] T. Fawcett. "An introduction to ROC analysis," Pattern Recognition Letters, vol. 27, no. 8, pp. 861–874, Jun. 2006.

[29] J. A. Hanley and B. J. McNeil. "The meaning and use of the area under a receiver operating characteristic (ROC) curve," Radiology, vol. 143, no. 1, pp. 29–36, Apr. 1982.

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

[31] K. Simonyan and A. Zisserman. "Very deep convolutional networks for large-scale image recognition," in Proc. 3rd International Conference on Learning Representations (ICLR), San Diego, CA, USA, 2015.

[32] K. He, X. Zhang, S. Ren, and J. Sun. "Deep residual learning for image recognition," in Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 2016, pp. 770–778.

[33] D. Kirat, G. Vigna, and C. Kruegel. "BareCloud: Bare-metal analysis-based evasive malware detection," in Proc. 23rd USENIX Security Symposium, San Diego, CA, USA, 2014, pp. 287–301.

[34] J. Kang, H. K. Kim, Y. Im, and K. Park. "Detecting and classifying malware based on extraction of binary sequences," International Journal of Distributed Sensor Networks, vol. 11, no. 7, p. 769, 2015.

[35] D. Gibert, C. Mateu, and J. Planes. "The rise of machine learning for detection and classification of malware: Research developments, trends and challenges," Journal of Network and Computer Applications, vol. 153, p. 102526, Mar. 2020.

[36] R. Vinayakumar, M. Alazab, K. P. Soman, P. Poornachandran, A. Al-Nemrat, and S. Venkatraman. "Deep learning approach for intelligent intrusion detection system," IEEE Access, vol. 7, pp. 41525–41550, 2019.

[37] S. Hou, A. Saas, L. Chen, Y. Ye, and T. Bourlai. "Deep neural networks for automatic Android malware detection," in Proc. IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM), San Francisco, CA, USA, 2017, pp. 803–810.

[38] A. Lyda and J. Hamrock. "Using entropy analysis to find encrypted and packed malware," IEEE Security & Privacy, vol. 5, no. 2, pp. 40–45, Mar. 2007.

[39] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. "Grad-CAM: Visual explanations from deep networks via gradient-based localization," in Proc. IEEE International Conference on Computer Vision (ICCV), Venice, Italy, 2017, pp. 618–626.

[40] R. Harang and E. M. Rudd. "SOREL-20M: A large scale benchmark dataset for malicious PE detection," arXiv preprint arXiv:2012.07634, 2020. [Online]. Available: https://github.com/sophos-ai/SOREL-20M

[41] M. Abadi et al.. "TensorFlow: A system for large-scale machine learning," in Proc. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Savannah, GA, USA, 2016, pp. 265–283.

[42] F. Pedregosa et al.. "Scikit-learn: Machine learning in Python," Journal of Machine Learning Research, vol. 12, pp. 2825–2830, Oct. 2011.

[43] R. Thomas. "LIEF: Library to Instrument Executable Formats," 2017. [Online]. Available: https://github.com/lief-project/LIEF. Accessed: Jun. 2024.

[44] E. Carrera. "pefile: Python PE parsing library," 2007. [Online]. Available: https://github.com/emonti/pefile. Accessed: Jun. 2024.

[45] C. R. Harris et al.. "Array programming with NumPy," Nature, vol. 585, no. 7825, pp. 357–362, Sep. 2020.