博 士 論 文

Semantic Code Learning for Binary Analysis: Applications to Malware Classification and Code Understanding

Minami Someya

染谷　実奈美

情報セキュリティ大学院大学
情報セキュリティ研究科
情報セキュリティ専攻

2025 年 9 月

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Day in and day out, security analysts confront an asymmetric battle: they must interpret compiled binary code that conceals its true intent, while attackers need only exploit a single vulnerability to succeed. This fundamental challenge—deciphering machine code without access to source code—defines modern cybersecurity. As malware grows increasingly sophisticated and proliferates at unprecedented rates, the gap between defensive capabilities and offensive threats continues to widen.

Binary code analysis serves as a cornerstone of defense in cybersecurity, focusing on understanding machine-executable code in the absence of human-readable source code. It has two primary objectives: detecting malicious software and uncovering vulnerabilities in legitimate programs. The task is complicated by both the overwhelming volume of samples and the technical difficulty of reverse engineering optimized, often obfuscated, machine code. Traditional manual analysis, though thorough, cannot scale to the demands of modern threats, underscoring the need for automated approaches that approximate human expertise.

Recent advances in machine learning, particularly deep learning, have achieved notable success in automating binary analysis. Neural networks can extract complex patterns from large datasets, attaining classification accuracies exceeding 98% on malware detection benchmarks [1, 2]. Even more impressively, Large Language Models (LLMs) have demonstrated strong code understanding capabilities, such as recovering meaningful function names from stripped binaries [3]. These advances point toward a future where AI systems augment human analysts—handling routine analysis tasks while flagging sophisticated threats for expert review.

However, several fundamental challenges limit the practical deployment of these technologies. First, the **interpretability gap**: deep learning models typically function as black boxes, offering predictions without explanations. For example, when a model labels a binary as ransomware, analysts cannot determine which code patterns influenced the decision, making it difficult to validate results or extract reusable insights. Second, the **deployment constraint**: LLMs generally require cloud-based infrastructure, rendering them unsuitable for sensitive malware samples that must remain in secure, offline envi-

ronments. Third, the **data scarcity problem**: malware source code is rarely available, since most samples exist only as compiled binaries. This scarcity severely restricts the development of specialized models for malware understanding.

This dissertation addresses these challenges through three complementary contributions. We develop **FCGAT**, an interpretable malware classification method that provides function-level explanations. We introduce **RevLlama**, which enables local deployment of LLM capabilities through knowledge distillation. Finally, we propose a **synthetic data generation framework** that produces diverse training samples without requiring actual malware source code. Together, these methods form a comprehensive framework for practical and interpretable binary code analysis.

1.2 Research Overview

This dissertation develops machine learning–based methods for functional understanding of binary code, with a particular emphasis on interpretability and practical deployment. The research addresses the critical need for automated analysis tools that security professionals can both trust and effectively apply in real-world environments.

We propose three approaches:

- **FCGAT**: An interpretable malware classification method leveraging function call graphs and attention mechanisms
- **RevLlama**: A knowledge distillation framework that enables local deployment of LLM capabilities
- **Synthetic data generation**: A systematic strategy for creating diverse malware training data

Together, these methods overcome the limitations of existing approaches, enabling accurate and interpretable binary analysis while respecting the operational constraints of security environments.

1.3 Interpretable Malware Classification Method

FCGAT (Function Call Graph with Attention) addresses the interpretability challenge in malware classification. Unlike traditional black-box approaches, FCGAT processes structural information from function call graphs while providing transparent decision-making through attention mechanisms.

The key contributions are:

- A malware classification method that explains decisions at the function level, allowing analysts to understand which program functions contribute to malicious behavior
- Experimental validation achieving F1-scores of 98.15% and 98.45% on benchmark datasets, on par with state-of-the-art methods
- Interpretability analysis that identifies characteristic functions for different malware families, such as encryption routines in ransomware samples

1.4 LLM-based Binary Code Function Understanding

The second contribution leverages Large Language Models (LLMs) for binary code understanding while addressing deployment constraints through two complementary approaches.

1.4.1 RevLlama: Knowledge Distillation for Function Name Recovery

RevLlama enables the local deployment of LLM capabilities for function name recovery—a critical task in reverse engineering, where meaningful names must be reconstructed for stripped binaries.

Key achievements include:

- A knowledge distillation framework that transfers reasoning capabilities from cloud-based LLMs to compact local models
- A performance improvement of 20.3% over GPT-4o while enabling fully offline operation
- Open-source release of models, training code, and datasets to support reproducible research

1.4.2 Synthetic Malware Generation for Data Augmentation

To address the problem of data scarcity, we develop a framework for generating synthetic malware samples that capture essential behavioral patterns without requiring access to real malware source code.

The main contributions are:

- A multi-category augmentation strategy covering file operations, network communication, encryption, and anti-analysis techniques
- **MalFuncBench**, a benchmark dataset containing expert-annotated functions from real malware families
- Experimental validation demonstrating that models trained on synthetic data achieve performance comparable to GPT-4o

1.5 Structure of the Thesis

Chapter 2 provides background on binary analysis, machine learning approaches, and related work in malware classification and code understanding. Chapter 3 introduces technical preliminaries, including graph neural networks and large language models. Chapter 4 details the architecture, training methodology, and interpretability analysis of FCGAT, with case studies on real malware families. Chapter 5 presents RevLlama ' s knowledge distillation framework and a comprehensive experimental evaluation. Chapter 6 describes the synthetic data generation approach and validates its effectiveness through extensive benchmarking. Finally, Chapter 7 concludes with a summary of contributions, limitations, and directions for future research.

Chapter 2

Background and Motivation

This chapter provides an overview of the technical challenges in binary analysis for cybersecurity and reviews relevant prior research. It highlights key gaps in current machine learning-based approaches.

2.1 Binary Analysis for Security

2.1.1 The Role of Binary Analysis in Cybersecurity

In modern cybersecurity, analysts rarely encounter malware in its original source code. Instead, they must analyze compiled binaries—machine code stripped of variable names, function names, comments, and high-level structures. This creates a fundamental asymmetry: attackers develop malware with well-documented source code, while defenders must reconstruct intent from opaque sequences of instructions.

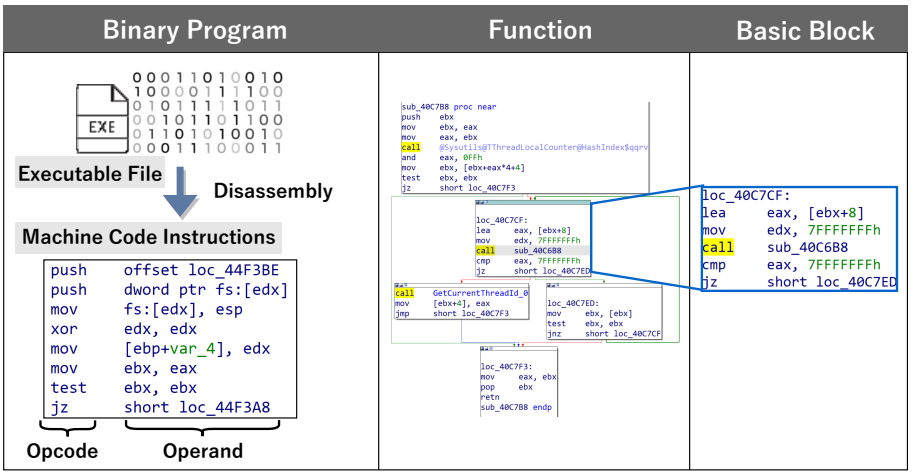

Figure 2.1: Hierarchical representation of a binary program

Figure 2.1 illustrates how binary programs can be represented at different abstraction levels. At the lowest level, binaries consist of machine code instructions composed of opcodes and operands. These instructions are grouped into basic blocks—sequences with

single entry and exit points—that form the atomic units of control flow. Basic blocks are then organized into functions, which encapsulate higher-level behaviors such as encryption, network communication, or file manipulation.

This hierarchical decomposition guides reverse engineering. Analysts move between levels of abstraction depending on their objectives: understanding detailed instruction-level behavior, reconstructing function-level semantics, or examining overall program organization.

2.1.2 Static vs. Dynamic Analysis

Binary analysis approaches fall into two fundamental categories, each with distinct advantages and limitations. Static analysis examines code without executing it, providing safety when dealing with potentially destructive malware and enabling comprehensive analysis of all code paths. Analysts can methodically examine every function, understand the overall program structure, and identify suspicious patterns without risk. However, static analysis struggles with packed or encrypted malware, where the true code is hidden until runtime, and with self-modifying code that changes its own instructions during execution.

Dynamic analysis takes the opposite approach, executing the binary in a controlled environment—typically a sandbox or virtual machine—to observe its actual behavior. This method reveals the program's true actions, including network communications, file operations, and system modifications. Dynamic analysis can automatically unpack encrypted code and observe runtime behaviors that would be difficult to predict from static examination alone. Yet it suffers from significant limitations: each execution reveals only one possible path through the program, sophisticated malware can detect virtualized environments and alter its behavior accordingly, and the analysis overhead makes it impractical for processing large volumes of samples.

In practice, effective malware analysis combines both approaches. Security teams often begin with static analysis to quickly triage large numbers of samples and identify interesting specimens for deeper investigation. They then employ dynamic analysis to confirm hypotheses and observe actual malicious behaviors. This dissertation focuses primarily on static analysis techniques, as they form the foundation for scalable, automated binary understanding systems.

2.2 Representations of Binary Code

The choice of how to represent binary code fundamentally shapes what machine learning models can learn and how effectively they can analyze malware. Different representations capture different aspects of program behavior, from low-level instruction patterns to high-level structural relationships.

2.2.1 Byte Sequences and OpCode Analysis

At the most basic level, a binary file consists of a sequence of bytes that encode both data and instructions. Some analysis approaches treat these raw bytes as input, applying

techniques from natural language processing to identify patterns. For instance, n-gram analysis can reveal common byte sequences associated with specific packers or compilers. However, byte-level analysis suffers from a fundamental limitation: not all bytes represent code, and the meaning of a byte depends heavily on its context within the instruction stream.

OpCode (operation code) analysis provides a more semantically meaningful representation by focusing on the instruction mnemonics after disassembly. Instead of raw bytes like `0x55 0x89 0xe5`, the analysis works with human-readable opcodes like `push`, `mov`, and `call`. This abstraction removes some architecture-specific details while preserving the essential program operations. Researchers have successfully applied various feature extraction techniques to opcode sequences, including frequency analysis, sequential pattern mining, and embedding methods borrowed from natural language processing.

2.2.2 Control Flow Graph (CFG)

Control Flow Graphs represent programs as directed graphs where nodes correspond to basic blocks—sequences of instructions with single entry and exit points—and edges represent possible control flow transfers between blocks. This representation captures the program's execution structure, including loops, conditionals, and function calls. CFGs provide rich structural information that helps distinguish between different algorithmic patterns and programming constructs.

However, CFGs of real-world programs present significant challenges for analysis. A typical malware sample might contain tens of thousands of basic blocks, resulting in graphs too large for efficient processing or human interpretation. The complexity increases further when considering indirect jumps and calls, which create uncertainty in the graph structure. Additionally, compiler optimizations can produce CFGs that bear little resemblance to the original source code structure, making interpretation difficult.

2.2.3 Function Call Graph (FCG)

Function Call Graphs abstract away intra-procedural details to focus on inter-procedural relationships. In an FCG, nodes represent functions and edges represent call relationships between them. This higher level of abstraction dramatically reduces graph complexity —a program with 50,000 basic blocks might have only 500 functions—while preserving important structural information about program organization.

FCGs offer several advantages for malware analysis. They capture the modular structure of programs, where specific functions implement distinct behaviors like file encryption, network communication, or system manipulation. The reduced graph size makes both machine learning and human interpretation more tractable. Furthermore, function-level analysis aligns well with how reverse engineers think about programs, as they typically analyze one function at a time before understanding inter-function relationships.

Our work leverages FCGs as the primary representation for malware classification, as they provide an optimal balance between expressiveness and tractability. The function-level granularity also enables meaningful explanations of classification decisions, as we can

identify which specific functions contribute to malicious behavior.

2.3 Machine Learning for Malware Classification

The evolution of malware classification techniques reflects broader trends in machine learning, progressing from simple statistical methods to sophisticated deep learning approaches. This section examines the major paradigms in automated malware classification, with particular attention to methods that process Windows PE (Portable Executable) format binaries.

2.3.1 Image-based Classification Methods

A surprising yet effective approach treats malware binaries as images. Pioneered by Nataraj et al. [4], this method converts binary files into grayscale images where each byte value (0-255) corresponds to a pixel intensity. The resulting images often display distinctive visual patterns: different sections of the executable appear as distinct textures, and similar malware families produce visually similar images.

The appeal of image-based methods lies in their ability to leverage powerful computer vision models. Researchers have successfully applied architectures like ResNet-50 [5], VGG-16 [6], and Inception-V3 [7] to malware classification, achieving impressive accuracy rates above 96% on benchmark datasets. The IMCFN model [8] further refined this approach by incorporating texture features specifically designed for malware images.

Despite their success, image-based methods suffer from a critical limitation: lack of interpretability. When a model classifies malware based on pixel patterns, it cannot explain which program behaviors or code structures led to its decision. A bright region in the image might correspond to encrypted data, repeated instructions, or even padding bytes —the visual representation obscures the actual program semantics. This opacity makes image-based methods unsuitable for applications requiring explainable decisions.

2.3.2 Sequence-based Methods

Binary-based methods treat executable files as sequences of bytes or opcodes, applying techniques from natural language processing and time series analysis. MalConv [1] pioneered end-to-end learning directly from raw bytes, using a specialized CNN architecture that processes entire executables without feature engineering. The model learns to identify malicious patterns from millions of training samples, achieving competitive performance while maintaining the flexibility to detect novel malware variants.

Other approaches preprocess the binary data before analysis. Qiao et al. [9] applied Word2Vec [10] to create embeddings of byte sequences, treating bytes as "words" in a specialized language. After filtering meaningless byte sequences, they fed the resulting 256-dimensional embeddings into a multi-layer perceptron for classification. This hybrid approach combines the representation learning power of Word2Vec with the classification capabilities of neural networks.

The primary advantage of sequence-based methods is their minimal preprocessing requirements—they can operate directly on executable files without disassembly or other costly transformations. However, they face a fundamental challenge: sequential proximity in the binary does not necessarily imply semantic relationship. Jump and call instructions can transfer control to distant addresses, breaking the sequential flow that these models assume. This limitation motivates the use of graph-based representations that explicitly capture control flow relationships.

2.3.3 Graph-based Methods

Disassembly-based methods first convert binaries into assembly code or graph structures before analysis. These approaches better capture program semantics but require more sophisticated preprocessing. MAGIC [11] constructs Control Flow Graphs from disassembled code and applies Deep Graph Convolutional Neural Networks (DGCNN) [12] for classification. Each basic block becomes a node with manually engineered features capturing instruction types, API calls, and structural properties.

Recent work by Wu et al. [2] introduced GEMAL, which operates on Function Call Graphs rather than CFGs. GEMAL first converts individual instructions to vector representations using Word2Vec, then aggregates these instruction vectors to create function embeddings. A graph attention network processes the FCG to produce a final malware classification. On the Microsoft Malware Classification Challenge dataset [13], GEMAL achieved 99.81% accuracy, demonstrating the effectiveness of function-level analysis.

The success of graph-based methods stems from their ability to capture program structure explicitly. Unlike sequence-based approaches, they naturally handle control flow transfers and preserve the hierarchical organization of real programs. However, existing graph-based methods, including GEMAL, operate as black boxes—they provide accurate classifications but cannot explain which parts of the program graph contributed to their decisions. This limitation motivates our work on interpretable graph-based classification.

2.3.4 Explainable Malware Classification

The need for explainable AI in security applications has driven research into interpretable malware classification methods. Yakura et al. [14] applied attention mechanisms to image-based classification, generating heat maps that highlight important byte regions. While innovative, their approach achieved only 49% accuracy—far below the requirements for practical deployment. The poor performance illustrates a common challenge: adding interpretability often degrades classification accuracy.

More recently, Herath et al. [15] developed CFGExplainer, which identifies important subgraphs in CFG-based classifiers through node pruning. Their method can isolate the 20% of nodes most relevant to classification decisions. However, even this reduced subgraph typically contains thousands of basic blocks—far too many for manual analysis. The sheer size of CFG explanations limits their practical utility for malware analysts who need to quickly understand malicious behaviors.

These limitations motivate our focus on function-level explanations. By operating on

FCGs rather than CFGs, we can provide explanations at a granularity that aligns with how analysts think about programs. Instead of highlighting thousands of basic blocks, our method identifies specific functions implementing malicious behaviors—a level of detail that enables practical analysis while maintaining high classification accuracy.

2.4 Large Language Models for Binary Code Understanding

The emergence of Large Language Models has created new opportunities for automated code understanding, including the challenging domain of binary analysis. This section examines how LLMs can be applied to reverse engineering tasks and the unique challenges that arise when processing decompiled code.

2.4.1 Code Understanding Tasks in Reverse Engineering

Reverse engineering encompasses several interconnected tasks that transform low-level binary representations into human-understandable formats. Variable name recovery attempts to assign meaningful names to memory locations and registers based on their usage patterns. Type inference reconstructs data structure definitions from memory access patterns. Code summarization generates natural language descriptions of function behavior. Among these tasks, function name recovery stands out as particularly challenging and valuable—it requires understanding the entire function's behavior and condensing it into a descriptive name that captures its purpose.

Traditional approaches to these tasks relied on pattern matching and heuristics. For example, a function that calls `CreateFile` followed by `WriteFile` might be named `write_to_file`. However, real-world binaries contain complex interactions, optimized code patterns, and obfuscated logic that confound simple rules. Modern malware actively obscures its behavior through indirect calls, API hashing, and control flow flattening, making pattern-based approaches ineffective.

2.4.2 Deep Learning Approaches to Function Naming

The application of deep learning to function name recovery began with relatively simple sequence-to-sequence models. Nero [16] pioneered the use of neural networks for this task, encoding control flow graphs into vector representations and generating function names through a decoder network. The key insight was treating the problem as a form of translation—from the "language" of assembly code to the "language" of function names.

SymLM [17] advanced this approach by incorporating broader program context. Instead of analyzing functions in isolation, SymLM considers how functions are called and what other functions they invoke. This contextual information proves crucial for disambiguation—a function that manipulates strings might be `decode_config` in one context but `decrypt_password` in another. By training on large corpora of open-source binaries, SymLM learned to recognize common patterns and generate appropriate names.

HexT5 [18] represents the current state-of-the-art in specialized models for binary analysis. Built on the CodeT5 architecture [19], HexT5 was specifically trained on decompiled

code, learning to handle the peculiarities of decompiler output. The model addresses multiple tasks simultaneously—function naming, summarization, and similarity detection—through a unified transformer architecture. This multi-task learning approach allows the model to develop richer internal representations that benefit all tasks.

2.4.3 Large Language Models in Binary Analysis

Recent work by Shang et al. [20] showed that general-purpose LLMs such as GPT-4 can perform competitively on binary understanding tasks, sometimes exceeding specialized models without additional training. This result indicates that the broad knowledge encoded in LLMs can transfer to the analysis of decompiled code. In particular, LLMs can draw on their understanding of programming patterns, API usage, and natural language to propose meaningful and descriptive function names.

LLMs also offer advantages beyond predictive accuracy. They can produce reasoning traces that clarify why a particular name is suitable, highlight operations that define a function ' s role, and suggest alternative names at different levels of abstraction. Such interpretability is useful in security analysis, where analysts must review and validate automated outputs.

At the same time, applying LLMs to malware analysis raises practical challenges. Most LLMs are provided as cloud services, requiring users to upload code to external servers. Security policies generally prohibit sharing malware samples with third parties, as the code may contain zero-day exploits, proprietary attack methods, or information about ongoing investigations. In addition, transferring malware outside secure environments increases the risk of accidental exposure or misuse, even if service providers apply strict safeguards.

2.5 Research Gaps and Thesis Contributions

The preceding sections outlined the state of research on binary analysis, malware classification, and code understanding. Despite significant progress, several gaps remain that hinder practical deployment in security operations.

2.5.1 The Interpretability Gap in Malware Classification

High-performance malware classifiers often function as black boxes, returning labels without explanations. Although many achieve accuracies above 98% on benchmark datasets, they do not reveal which code features influenced their decisions. This lack of transparency poses challenges for security teams: false positives or negatives cannot be reliably verified, analysts cannot build on model insights to deepen their understanding of malware, and explainability requirements in regulated environments may not be met.

The proposed FCGAT method addresses this issue by providing explanations at the function level. Rather than abstract activations, it identifies specific functions associated with malicious behavior, offering outputs that analysts can examine and validate.

2.5.2 The Deployment Gap for LLM-based Analysis

While LLMs perform well on code understanding tasks, their reliance on cloud infrastructure limits their use in security contexts. Malware must remain in secure environments, attack techniques should not be disclosed, and ongoing investigations cannot be exposed externally. Using small local models reduces accuracy, but uploading samples to the cloud conflicts with operational requirements.

RevLlama addresses this limitation by applying knowledge distillation to create compact models that approximate or surpass cloud-based LLM performance while running fully offline. This enables security teams to apply advanced AI techniques without compromising data security.

2.5.3 The Data Scarcity Problem

Both classification and code understanding tasks are constrained by the limited availability of training data. Malware source code is seldom available, and collecting or labeling samples is resource-intensive and legally sensitive. As a result, existing datasets are typically small, biased toward older families, and may not capture recent techniques.

To mitigate this problem, this dissertation introduces a synthetic data generation framework. It produces diverse and realistic training samples without requiring access to actual malware. By systematically generating variations across different behavioral categories, the framework expands available datasets and improves the robustness of trained models.

Chapter 3

Preliminaries

This chapter introduces the technical background necessary to understand the methods presented in this dissertation. It begins with natural language processing (NLP) techniques applied to code, then reviews fundamental machine learning concepts including neural networks and graph-based models. The chapter concludes with an overview of Large Language Models (LLMs) and knowledge distillation methods that support practical deployment.

3.1 Natural Language Processing for Code

3.1.1 Code as Language

Programming languages, although formally defined and semantically precise, share several properties with natural languages that make NLP techniques useful for code analysis. Code has hierarchical structure, from tokens to statements, functions, and entire programs. It follows grammatical rules (syntax) and conveys meaning (semantics) through symbol combinations. Importantly, even compiled binaries originate from source code that reflects human design choices and naming conventions.

This analogy leads to useful insights. Like words in natural language, programming constructs derive meaning from their usage. A function invoked alongside file I/O routines likely handles file operations, while one called in cryptographic contexts may implement security-related functionality. Code also contains recurring patterns and conventions that statistical models can learn to recognize.

At the same time, code differs from natural language in ways that affect model design. It must be syntactically valid to compile and semantically correct to execute. Minor edits, such as a missing semicolon, can change behavior entirely. Long-range dependencies —through function calls, variable scope, or control flow—often extend beyond typical sentence lengths in natural language. These characteristics require careful adaptation of NLP techniques for code.

3.1.2 Embeddings and Representation Learning

Machine learning models require numerical input, so code must be transformed into vector representations that preserve semantics. The aim is for semantically similar code fragments to map to similar embeddings, while dissimilar code maps farther apart.

Token-level embeddings are the starting point. Early methods used one-hot encodings or learned embedding matrices for code tokens. Word2Vec [10] improved this by learning dense representations from context. Applied to code, it can group functions such as `malloc` and `calloc`, or I/O routines like `printf` and `scanf`, based on similar usage patterns.

A challenge in code embeddings is the open vocabulary. Identifier names, numeric constants, and string literals can vary widely. Subword methods like byte-pair encoding (BPE) address this by breaking rare tokens into frequent subunits. For example, `calculateHashValue` can be split into `calculate`, `Hash`, and `Value`, whose embeddings combine to represent the full identifier.

Binary analysis presents additional challenges. Decompiled code often uses generic variable names such as `var_1` and `var_2`, so semantics must be inferred from usage. At the assembly level, instruction embeddings must capture both the operation (opcode) and operands, which may include registers, memory addresses, or constants. Our work builds on these ideas, applying embedding methods adapted to the specific properties of disassembled and decompiled code.

3.2 Foundations of Machine Learning

3.2.1 Deep Neural Networks

Deep neural networks (DNNs) are widely applied to tasks such as malware classification and function name recovery. They learn hierarchical representations through stacked layers of linear transformations with non-linear activations:

$$\mathbf{h}^{(l+1)} = f(\mathbf{W}^{(l)}\mathbf{h}^{(l)} + \mathbf{b}^{(l)})$$

where $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$ are the learnable weight matrix and bias, and f is a non-linear activation function such as ReLU. Early layers capture local features such as token patterns or instruction sequences, while deeper layers capture higher-level structures including control flow or function interactions.

Training DNNs introduces issues such as vanishing gradients, overfitting, and class imbalance. Common countermeasures include residual connections, normalization layers (e.g., BatchNorm, LayerNorm), dropout, weight decay, and data augmentation.

Different architectures emphasize different strengths. Convolutional Neural Networks (CNNs) detect local byte patterns, Recurrent Neural Networks (RNNs) and LSTMs model sequential dependencies, and Transformer architectures have become dominant for code understanding due to their ability to represent long-range relationships and support efficient parallelization.

3.2.2 Graph Neural Networks (GNNs)

Graph Neural Networks (GNNs) [21] have gained significant attention in recent years for their ability to model data with graph structures, such as social networks, biological pathways, and program call graphs [22]. GNNs are applicable to various tasks, including node classification, graph classification, and link prediction. In this study, we construct Function Call Graphs (FCGs) by treating each function as a node and connecting functions with edges based on their call relationships. We then perform graph classification on the resulting graphs.

(1) Graph Definitions

A graph $G = (V, E)$ consists of a set of nodes $V = \{v_1, ..., v_n\}$ and a set of edges $E = \{e_1, ..., e_m\}$. Each edge $e \in E$ connects two nodes u and v as $e = (u, v)$. The neighborhood of a node v is defined as $\mathcal{N}(v) = \{u \mid (u, v) \in E\}$. Each node v is associated with an initial feature vector $\mathbf{x}_v \in \mathbb{R}^k$, and the entire graph can be represented with a feature matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$. The latent representation of a node in a hidden layer is denoted $\mathbf{h}_v$, and the representation of the entire graph is $\mathbf{h}_G$.

(2) Graph Classification with GNNs

Graph classification using GNNs generally consists of three phases:

1. **AGGREGATE**: Aggregate the features of neighboring nodes.
2. **COMBINE**: Update the node's feature vector using the aggregated information.
3. **READOUT**: Aggregate all node vectors into a graph-level vector.

The node update formula at the k-th layer is as follows:

$$\mathbf{h}_v^k \leftarrow \mathrm{COM}\left(\mathbf{h}_v^{k-1}, \mathrm{AGG}\left(\left\{\mathbf{h}_u^{k-1} \mid u \in \mathcal{N}(v)\right\}\right)\right)$$

In Figure 3.1, node v_1 aggregates the features of its neighbors v_2, v_3, and v_4. These vectors are processed by the AGGREGATE function and then combined with v_1 's own vector via the COMBINE function to update its features. This process is repeated for all nodes, and the READOUT function is used to obtain the graph-level vector $\mathbf{h}_G$, which is then passed through a fully connected layer for classification.

3.2.3 Attention Mechanisms and Transformers

Attention mechanisms are widely used in modern machine learning because they allow models to focus on the most relevant parts of the input rather than processing all information uniformly. This improves predictive performance and also provides a degree of interpretability, since the learned attention weights indicate which parts of the input contributed most to the decision.

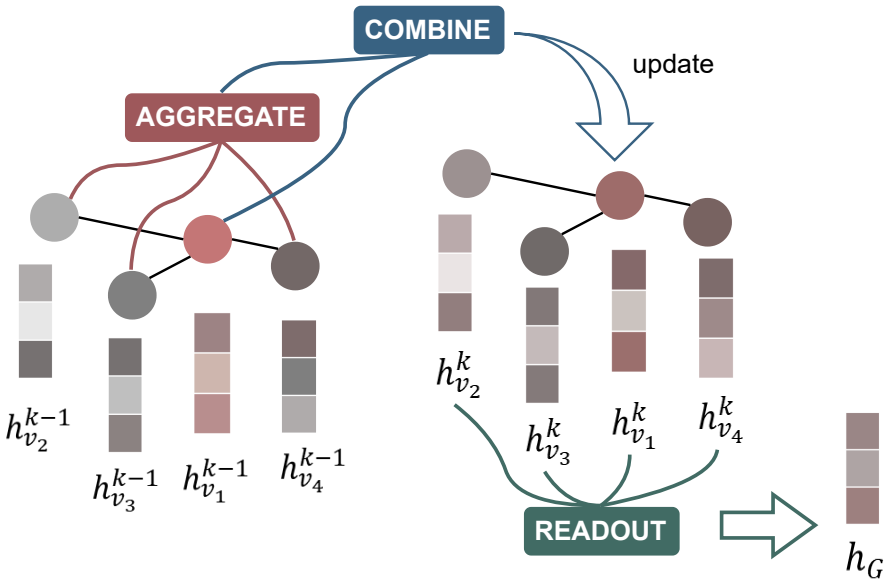

Figure 3.1: Overview of graph classification using GNNs

(1) Fundamental Attention Mechanism

The basic idea of attention is to weight different input elements according to their importance for the current task. Given a query $\mathbf{q}$, keys $\mathbf{K}$, and values $\mathbf{V}$, attention is defined as:

$$\text{Attention}(\mathbf{q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

Here, the query represents what the model is trying to find, the keys indicate what information is available, and the values contain the content to be combined. The dot product $\mathbf{q}\mathbf{K}^T$ measures similarity: if a key is highly relevant to the query, it will receive a larger weight. The softmax operation normalizes these similarity scores into probabilities, and the final output is a weighted sum of the values.

The scaling factor $\sqrt{d_k}$ ensures stable training. Without this normalization, the dot products could grow too large in high-dimensional spaces, pushing the softmax into regions where gradients vanish. In practice, this detail makes the difference between unstable and effective training.

(2) Multi-Head Attention

A single attention operation focuses on one type of relationship at a time. However, real-world data often contains multiple patterns. Multi-head attention addresses this by applying several independent attention operations in parallel:

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, ..., \text{head}_h)\mathbf{W}^O$$

with each head defined as:

$$\text{head}_i = \text{Attention}(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V)$$

Each head uses its own learned projections, allowing specialization. For example, in text processing one head may focus on word order (syntax), another on semantic similarity, and another on long-range dependencies. By combining their outputs, the model captures diverse aspects of the input.

(3) Transformer Architecture

The Transformer architecture builds layers by stacking multi-head attention with position-wise feed-forward networks. Residual connections and layer normalization are used to stabilize training:

$$\text{LayerOutput} = \text{LayerNorm}(\mathbf{x} + \text{MultiHeadAttention}(\mathbf{x}))$$

$$\text{Output} = \text{LayerNorm}(\text{LayerOutput} + \text{FeedForward}(\text{LayerOutput}))$$

This architecture offers two main advantages. First, self-attention captures long-range dependencies efficiently: a word or token at the beginning of a sequence can directly attend to another far downstream, without step-by-step propagation as in RNNs. Second, the computations are parallelizable, which enables faster training on modern hardware. These properties explain why Transformers have become the dominant model across domains such as natural language processing, program analysis, and many others.

3.2.4 Large Language Models (LLMs)

Large Language Models extend the Transformer architecture to very large scales, achieving state-of-the-art results in both natural language and programming language tasks.

3.2.5 Large Language Models and Capabilities

LLMs are generally trained with an autoregressive objective: predicting the next token given its preceding context:

$$p(x_t \mid x_1, ..., x_{t-1}) = \text{softmax}(\mathbf{W}\mathbf{h}_t + \mathbf{b})$$

where $\mathbf{h}_t$ is the hidden representation at position t. The training objective is to maximize the likelihood of observed token sequences:

$$\mathcal{L} = -\sum_{t=1}^{T} \log p(x_t \mid x_1, ..., x_{t-1})$$

Scaling these models to billions of parameters and training them on massive datasets allows them to generalize across a wide range of tasks, including translation, summarization, code generation, and reasoning.

Empirical studies have identified scaling laws that relate performance to model size N and dataset size D:

$$L(N, D) = \left(\frac{N_c}{N}\right)^{\alpha_N} + \left(\frac{D_c}{D}\right)^{\alpha_D}$$

with N_c, D_c, α_N, and α_D determined empirically. These laws provide guidance on how to balance model capacity and dataset size for efficient training.

As LLMs grow larger, they often display new behaviors that smaller models lack. For example, the ability to perform multi-step reasoning or infer meaningful function names may appear only once the model exceeds a certain scale. These scale-dependent improvements make LLMs powerful tools across domains, though they also raise challenges related to efficiency, deployment constraints, and responsible use.

(1) Training Methods for LLMs

Modern LLMs are typically trained in several stages, each addressing a different objective:

1. **Pretraining**: Large-scale unsupervised learning on massive text corpora (hundreds of billions of tokens). The model learns general linguistic and semantic patterns.
2. **Instruction Tuning**: Supervised fine-tuning on curated datasets where inputs are paired with explicit instructions and outputs, making the model more responsive to user queries.
3. **Reinforcement Learning from Human Feedback (RLHF)**: Aligning outputs with human preferences through reinforcement learning, where annotators provide feedback on which responses are more helpful or appropriate.

For specialized domains such as code analysis, continued pretraining or fine-tuning on domain-specific corpora can yield significant improvements. However, the resource cost is very high—full pretraining may require thousands of GPUs and training budgets of millions of dollars—making it impractical for most research groups. As a result, lightweight adaptation methods such as parameter-efficient fine-tuning are often preferred in practice.

(2) Prompting Techniques

One distinctive property of LLMs is their ability to perform new tasks without additional training, if guided by well-designed prompts. This practice, known as *prompt engineering*, exploits the model ' s internal knowledge. Two widely used strategies are outlined below.

Few-Shot Learning By including a handful of input-output examples in the prompt, the model can infer the pattern and apply it to a new case. For example, when generating function descriptions, the prompt might include:

```
Q: Which country does Tokyo belong to?
A: Tokyo → Japan

Q: Which country does Cairo belong to?
A: Cairo → Egypt

Q: Which country does Sydney belong to?
A: Sydney → [MODEL PREDICTS]
```

The model imitates the format and semantic style of the examples, producing a description for the target function.

Chain-of-Thought Prompting For tasks requiring reasoning, prompting the model to explicitly "think step by step" can improve accuracy. This technique encourages the model to generate intermediate reasoning before a final answer, for instance:

```
Let's solve this step by step:
1. The listed price is $50. A 20% discount applies → 50 ×
   0.8 = 40.
2. A 10% tax is added to the discounted price → 40 × 1.10 = 44.
Therefore, the final price is $44.
```

This structured reasoning mimics human analysis and often yields more interpretable and correct outputs.

In our work, such prompting methods are used to extract high-quality reasoning traces from large teacher models. These traces then serve as supervision signals when training smaller student models, enabling them to inherit both predictive performance and reasoning ability.

3.2.6 Knowledge Distillation

Knowledge distillation transfers the capabilities of a large, high-performance *teacher* model into a smaller, more efficient *student* model. This is especially important in constrained environments, where the deployment of cloud-scale models is impractical or prohibited for security reasons.

The most common approach trains the student to mimic the teacher's probability distribution over outputs, not just the final label. By learning from the teacher's confidence scores, the student captures subtle distinctions between classes, which improves generalization.

Beyond output imitation, *feature distillation* aligns intermediate representations of the student with those of the teacher. In this way, the student learns not only what the teacher predicts but also how it arrives at those predictions. This can embed reasoning patterns into compact models, enhancing interpretability and robustness.

Through distillation, large LLMs can be used to supervise smaller, locally deployable models that retain much of their performance while being computationally cheaper and safer to use in sensitive domains.

3.3 Summary

This chapter introduced the technical background for the methods developed in later chapters. We examined how natural language processing techniques apply to code, reviewed deep learning models from standard networks to graph-based architectures, and discussed attention mechanisms and Transformers. We also outlined Large Language Models, their training methods, prompting strategies, and knowledge distillation.

These foundations support the main contributions of this dissertation: FCGAT for interpretable malware classification, RevLlama for secure deployment of LLM capabilities, and a synthetic data generation framework for training. The next chapters present each contribution in detail.

Chapter 4

FCGAT: Interpretable Malware Classification Method using Function Call Graph and Attention Mechanism

This chapter presents FCGAT (Function Call Graph with Attention), our interpretable malware classification method that addresses the critical need for explainable AI in security applications. We demonstrate how combining graph neural networks with attention mechanisms enables both high classification accuracy and function-level interpretability, providing security analysts with actionable insights into model decisions.

4.1 Introduction

Malware classification—categorizing malicious software into families based on shared characteristics and behaviors—forms a cornerstone of modern cyber defense. When confronted with new malware samples, security teams must quickly understand their capabilities and develop appropriate countermeasures. While machine learning has shown remarkable success in automating this classification, existing approaches suffer from a fundamental limitation: they cannot explain their decisions in terms meaningful to security analysts.

The challenge extends beyond academic interest. In operational security environments, analysts need to understand why a sample was classified as ransomware versus a banking trojan. They must validate model predictions against their domain expertise, extract insights about emerging threats, and justify response actions to stakeholders. Black-box classifiers, despite their high accuracy, fail to meet these practical requirements.

FCGAT addresses this interpretability gap through a novel architecture that operates on Function Call Graphs. By applying attention mechanisms at the function level, FCGAT not only achieves state-of-the-art classification performance but also identifies which specific functions contribute most to malicious behavior. When FCGAT classifies a sample as ransomware, it simultaneously highlights the encryption routines, file enumeration

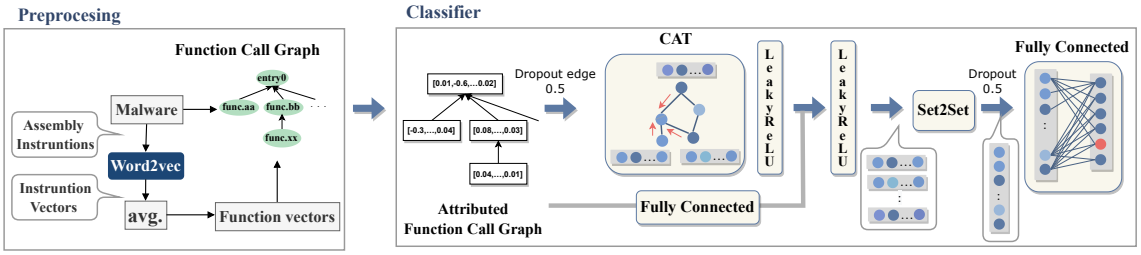

Figure 4.1: Overview of our proposed method FCGAT

functions, and ransom note generation code that led to this decision.

The key insight underlying our approach is that malware analysts naturally think in terms of functions. When reverse engineering malware, analysts examine individual functions to understand their purpose, then trace function calls to comprehend overall behavior. By aligning our model's decision-making process with this analytical workflow, we create explanations that directly support human understanding and validation.

4.2 Proposed Method

An overview of the proposed method FCGAT is shown in Fig. 4.1. FCGAT consists of three parts: FCG creation, function feature creation, and malware classification. FCGAT takes a binary file as input. A FCG is created from a binary file, and feature vectors of functions extracted using Word2vec are assigned as the node features in the graph. Finally, FCGAT classifies malware class by GNN using the FCG as input.

4.2.1 Creation of Function Call Graph

FCG is a directed graph where nodes represent functions and edges represent function calls. Although a general FCG connects edges from the calling function to the direction of called function, FCGAT reverses this arrow. In programs, the called function can be nested within the calling function. Therefore, the features of the called function should be aggregated in the calling function, thus reverse FCG is used. In a general FCG, the ends of the arrows usually represent API functions (often API wrappers). That is, FCGAT aggregate the features of API functions into the calling functions.

4.2.2 Function feature creation

This module converts assembly functions into function vectors that can be handled by machine learning. These function vectors need to reflect the semantics of the functions in order to classify malware with high performance. FCGAT uses CBOW model of Word2vec [10] to obtain instruction vectors corresponding to assembly instructions. This method is based on natural language processing techniques, where instructions correspond to words and functions correspond to sentences. CBOW model obtains instruction vectors

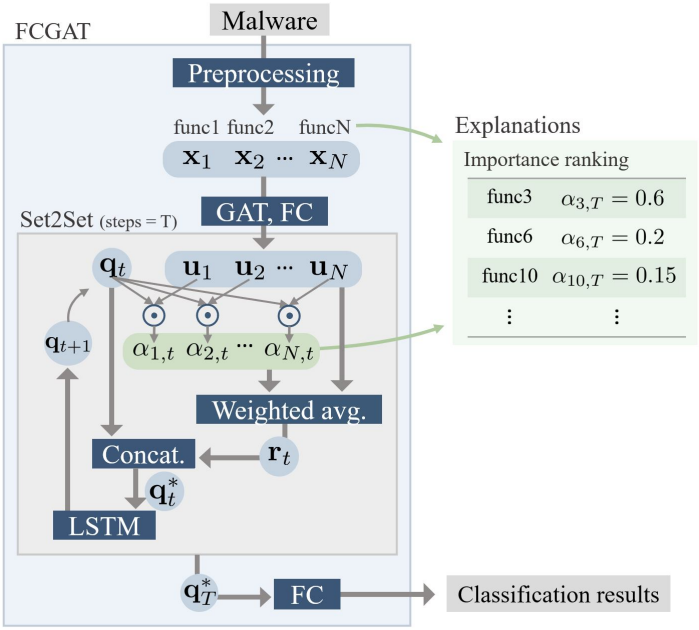

Figure 4.2: An overview of Set2Set process and how it explains the classification results

by predicting the current instruction from surrounding instructions. Word2vec uses unsupervised learning and does not require any prior knowledge. This method is well suited for learning features of malware functions that are difficult to label, unlike open-source programs.

Since assembly instructions contain numerical values such as memory addresses and offset values, the vocabulary becomes excessively large if used as is. Therefore, the assembly instructions are filtered as follows:

1. Remove ”,” and comments after ”;”.
2. Replace the number with ”N”.
3. Connect operand and opcode with ”_”.

This filtering makes use of the analysis results of analysis tool. In our implementation, we use IDA Pro as the analysis tool. IDA Pro converts well-known functions, such as API functions, from addresses to function names so that instruction vectors can reflect the characteristics of well-known function names.

After creating the instruction vectors in the CBOW model, FCGAT averages the instruction vectors in the function to obtain a function vector. This vector is assigned as the feature of a node in a FCG.

4.2.3 Malware classification model

The structure of the classification model is shown in Fig. 4.1. The inputs are FCGs with function feature vectors as node features. The FCGs are convolved with *Graph Attention*

Network (GAT) [23]. The features of the nodes are updated by the following:

$$\mathbf{u}_i = \text{LeakyReLU}(\mathbf{W_1}\mathbf{x}_{v_i} + \text{LeakyReLU}(\|_{k=1}^{K} \sum_{\mathbf{x}_j \in N(v_i)} \alpha_{ij}^k \mathbf{W}^k \mathbf{x}_j)),$$

where $\mathbf{x}_{v_i}$ is the initial function vector at a node to be updated, $\mathbf{x}_j \in N(v_i)$ is the neighbor of the node, $\mathbf{W}_1$ and $\mathbf{W}^k$ are the learning parameters. Multi-head Attention is used to concatenate the results from K-independent Attention mechanisms. The α_{i_j} represents the importance between nodes. α_{i_j} is learned by the following:

$$\alpha_{ij} = \text{softmax}_j \left(\text{LeakyReLU} \left(\mathbf{a}^\top \left[\mathbf{W}_2 \mathbf{x}_{v_i} \| \mathbf{W}_2 \mathbf{x}_{v_j} \right] \right) \right).$$

The GAT layer's attention mechanism is not used for interpretation. This is because α_{ij} given for each edge depends on the number of neighbors, making it hard to reflect the importance of a node. For example, if there is only one neighbor, α_{ij} will be the maximum possible value of 1, even if the node is not important. In this layer, Multi-head Attention has the effect of expanding representation, which improves performance.

FCGAT does not convolve the features of the updating node but only aggregates the features of the neighbors. The features of the updating node are converted to the hidden layer dimension at the fully connected layer (corresponding to $\mathbf{W_1}\mathbf{x}_{v_i}$) and added to the output of GAT.

Next, the readout process uses Set2Set [24], which is the key to interpretability, to obtain the feature vector of the entire graph as follows:

$$\begin{aligned}
\mathbf{q}_t &= \text{LSTM}(\mathbf{q}^*_{t-1}) \\
\alpha'_{i,t} &= \text{softmax}(\mathbf{u}_i \cdot \mathbf{q}_t) \\
\mathbf{r}_t &= \sum_{i=1}^{N} \alpha'_{i,t} \mathbf{u}_i \\
\mathbf{q}^*_t &= \mathbf{q}_t \| \mathbf{r}_t.
\end{aligned}$$

Set2Set is an extension of the seq2seq [25] approach to support unordered sets. That is, permuting $\mathbf{u_i}$ and $\mathbf{u}_{i'}$ has no effect on the read vector $\mathbf{r}_t$. The ordering of the nodes in FCG depends on the starting address of the function, but this ordering is usually meaningless. Thus FCG is considered suitable to be processed by Set2Set. Fig. 4.2 shows an overview of Set2Set process and how it explains the classification results. FCGAT can apply the attention mechanism to the variable number of nodes by using LSTM, and we expect that the features of nodes that are important for classification are given high attention. In other words, we can obtain the important functions by analyzing the value of $\alpha'_{i,t}$. Based on attention weights in the final step of Set2Set, function name and importance $\alpha'_{i,t}$ are provided as an importance ranking. In order to reflect the importance for classification to the attention weight, simple network structures are adopted for the other parts. For example, only one convolution in GAT is used to avoid excessively diluting the original function's features.

Finally, classification is performed in a Fully Connected layer. This layer takes a graph vector $\mathbf{h}_G$ as input and outputs a vector representing the classification probabilities of the

Table 4.1: Classification model parameters

Layer	(In, Out)	Activation	Remark
GAT	(100, 192)	LeakyReLU	K = 3
FC	(100, 192)	LeakyReLU	
Set2Set	(192, 384)		steps = 4
FC	(384, Number of classes)	Softmax	

classes as follows:

$$\mathbf{p} = \text{softmax}(\mathbf{W}_3\mathbf{h}_G),$$

where $\mathbf{W}_3$ is a learninig parameter. The index with the highest value among the $\mathbf{p}$ elements indicates the predicted class.

We adopt Dropout [26] to prevent overfitting and stabilize learning. Dropout randomly drops layer outputs to 0 during training to enable correct recognition even if some data are lost. It helps avoid overfitting some local features and improves the robustness of the model. In Fig. 4.1, Dropout 0.5 means dropping the node's output to 0 with 0.5 probability. In addition, cross-entropy is used as the error function, and AdamW [27] is used as the optimization algorithm.

4.3 Evaluation

We evaluated FCGAT in terms of classification performance (in Section 4.3.2) and classification interpretability (in Section 4.3.3). Before delving into them, we describe the experimental setup common to both. Our research artifacts are available on Github [1].

4.3.1 Experimental setup

We implemented FCGAT using Python, PyTorch 1.11.0, and PyTorch Geometric 2.0.2, a library for GNN. FCGAT uses a reverse engineering tool IDA Pro for preprocessing. IDA Python is used to automate operations in CUI, and *ida_gdl.gen_simple_call_chart* function is used to create FCG.

For function feature creation, Gensim library is used to create the CBOW model. When evaluating performance, only train data is used to create CBOW training models, which are then used for the evaluation of test data. We choose parameters embedding size 100, windowsize 2, epochs 100 for the CBOW model. Where, embedding size is the dimension of the feature vector of the function, and window size is the number of surrounding words to be considered. In other words, instruction vectors are learned from the two instructions before and after.

For training the classification model, we choose learning rate 0.001, mini-batch size 256, and epochs 700. The model parameters are shown in TABLE 4.1.

[1] https://github.com/som3ya/FCGAT

4.3.2 Classification performance evaluation

In this experiment, we evaluate the performance of FCGAT on two datasets and compare it with the demonstration results of existing research by Ma et al. [28]. We also compare with GEMAL by Wu et al. [2]. We implemented and performed a replication experiment with GEMAL to compare on the same dataset.

(1) Datasets

We used two datasets that Ma et al. used in their experiments on existing studies. We created FCGs and the function vectors for each dataset using FCGAT. Then, the function vectors were assigned to the nodes of the FCG, and datasets with graph format were created. We used the following datasets.

MalwareBazaar dataset MalwareBazaar dataset was published by Ma et al. [28]. MalwareBazaar [2] is a website that provides malware for research purposes. Ma et al. created the dataset using the following procedure. First, they selected the top six malware families from the malware released in 2020 on MalwareBazaar and downloaded 1000 malware samples from each family. Next, they filtered out samples that were not in PE format and used Joe Security [3] and AVClass [29] to check the label of each sample and to determine whether the label removed inconsistent samples. As a result, they obtained 3,971 samples from 6 families. Ma et al. published a list of hash values and family names of the samples on their GitHub [4]. We downloaded 3,971 malware samples using this list and MalwareBazaar's API. We used 3,968 samples (TABLE 4.2) that were successfully preprocessed for the evaluation experiments.

BIG-2015 The second dataset is BIG-2015 [13], provided by Microsoft, which consists of 10,868 samples from 9 families. Each sample consists of .byte files, a binary representation of a hexadecimal number, and .asm files, the disassembly outputs of IDA Pro. In this experiment, we created FCGs from .asm files and extracted function binaries. 10,810 samples (TABLE 4.3) that were successfully preprocessed were used in the evaluation experiment.

[2]https://bazaar.abuse.ch
[3]https://www.joesecurity.org
[4]https://github.com/MHunt-er/Benchmarking-Malware-Family-Classification

Table 4.2: MalwareBazaar

Family	Counts
Gozi	767
GuLoader	589
Heodo	214
IcedID	578
njrat	939
Trickbot	881
Total	3,968

Table 4.3: BIG-2015

Family	Counts
Ramnit	1,533
Lollipop	2,475
Kelihos_ver3	2,938
Vundo	453
Simda	42
Tracur	751
Kelihos_ver1	387
Obfuscator.ACY	1,217
Gatak	1,012
Total	10,808

(2) Evaluation metrics

We use the same evaluation metrics as in the experiment of the comparator Ma et al. Four evaluation metrics are used: Accuracy, Precision (P_{macro}), Recall (R_{macro}), and F1-Score ($F1_{\text{macro}}$). Each formula is expressed as following:

$$Accuracy = (\sum_{n=1}^{N} \sum_{m=1}^{M} TP_{mn})/S$$

$$P_{\text{macro}} = \left(\sum_{n=1}^{N} P_n\right)/N$$

$$R_{\text{macro}} = \left(\sum_{n=1}^{N} R_n\right)/N$$

$$F1_{\text{macro}} = \left(\sum_{n=1}^{N} F1_n\right)/N$$

where

$$P_n = \left(\sum_{m=1}^{M} TP_{mn}\right) / \left(\sum_{m=1}^{M} (TP_{mn} + FP_{mn})\right)$$

$$R_n = \left(\sum_{m=1}^{M} TP_{mn}\right) / \left(\sum_{m=1}^{M} (TP_{mn} + FN_{mn})\right)$$

$$F1_n = (2 * P_n * R_n) / (P_n + R_n)$$

N is the number of all classes, S is the number of all samples, and M is the number of cross-validation splits. TP_{mn}, FP_{mn} and FN_{mn} represent the true positive, false positive, and false negative of malware family n in the m-fold.

(3) Experimental results

TABLE 4.4 shows the results of the evaluation metrics computed and compared with the existing studies. For GEMAL, we adopt the results of replication experiments to evaluate performance instead of the paper values because we cannot compare them with MalwareBazaar dataset based on the paper values.

Table 4.4: Comparison of experimental results with those in existing studies

Category	Model	MalwareBazaar dataset				BIG-2015			
		Accuracy	P_{macro}	R_{macro}	$F1_{macro}$	Accuracy	P_{macro}	R_{macro}	$F1_{macro}$
Image	ResNet-50 [5] *	96.68	96.91	96.75	96.83	98.42	96.57	95.68	96.08
	VGG-16 [6] *	96.35	96.58	96.54	96.56	93.94	90.32	81.89	87.27
	Inception-V3 [7] *	95.83	95.67	95.79	95.73	96.99	93.67	94.46	94.03
	IMCFN [8] *	97.38	97.53	97.41	97.47	97.77	95.93	94.81	95.13
Binary	CBOW+MLP [9] *	**97.81**	**97.92**	**98.08**	**98.00**	98.41	97.63	96.67	97.12
	MalConv [1] *	95.92	96.04	96.43	96.20	97.02	94.34	92.62	93.33
Disassembly	MAGIC [11] *	92.82	88.03	87.36	87.45	98.05	96.75	94.03	95.14
	Word2vec+KNN [30] *	95.64	93.34	94.29	93.79	98.07	96.41	96.51	96.45
	MCSC [31] *	96.80	94.97	94.51	94.70	97.94	95.97	96.17	96.06
	FCGAT(proposed method)	**98.11**	**98.03**	**98.27**	**98.15**	**99.27**	**97.93**	**98.45**	**98.18**
	GEMAL(replication) [2]	97.71	97.65	98.00	97.82	**99.37**	**98.26**	**98.48**	**98.37**
	GEMAL(paper) [2] †	-	-	-	-	99.81	-	-	99.81

∗ and † mean the experimental results by Ma et al. [28] and Wu et al. [2], respectively. The top two in each evaluation metric are shown in **bold**. In MalwareBazaar dataset, FCGAT outperformed all other methods on all metrics. In BIG-2015, the F1-Score of FCGAT is 98.18%, which is about equivalent to 98.37% in the replication experiment of GEMAL.

In MalwareBazaar dataset, FCGAT successfully classifies with an F1-Score of 98.15% and outperforms all other methods on all metrics. In BIG-2015, the F1-Score of FCGAT is 98.18%, which is about equivalent to 98.37% in the GEMAL replication experiment.

As noted in Section 2.3.3, the disassembly-based methods perform poorly in the Ma et al. experiments, especially in MalwareBazaar dataset. The reason is considered that these methods do not use the PE header feature of executables. It has been noted that models that take the entire executable file as input, such as MalConv [1], tend to mainly pay attention to the PE header in malware classification [32]. However, classifying malware by focusing on PE headers is risky because PE headers contain information that can be spoofed (e.g., timestamps). FCGAT outperforms the methods that use the entire files, even though it only uses code segment features. We think that FCGAT is able to represent malware features more accurately by extracting features on a function basis.

4.3.3 Classification interpretability

In this experiment, we perform malware category classification by FCGAT and extract the importance ranking of the functions as explanations, and verify the effectiveness of these explanations. Malware category indicates malware features such as Downloader, Ransomware, and Trojan. We expect FCGAT to focus on malware features by classifying them into malware categories rather than families. That is, we predict that the important function will be network communication functions for Downloader and encryption functions for Ransomware.

(1) Datasets

We used a dataset named BODMAS-8cat, modified from BODMAS [33]. BOSMAS was published by Yang et al. and contains 57,293 executables labeled by 581 families and 14 categories. In this experiment, we excluded as many packed files as possible to focus on the malware features. We excluded from BODMAS those samples that have been

pack-detected by PEiD [5]. Then, the category with more than 90 samples in each malware category was selected among the successfully preprocessed samples. As a result, BODMAS-8cat was created, containing 23,369 samples in 8 categories, as shown in TABLE 4.5.

Table 4.5: Details of BODMAS-8cat

Category	Family Counts	Sample Counts
backdoor	31	598
downloader	19	967
dropper	17	397
informationstealer	19	347
ransomware	18	169
trojan	282	15,674
virus	8	93
worm	87	5,124
total	481	23,369

(2) Contributions of important functions

In this section, we examine how much the important functions contribute to the classification. Before that, we evaluate FCGAT using BODMAS-8cat with 8:2 hold-out validation. The classification performance of the test data was accuracy of 95.04%. Malware category classification is a more difficult than family classification due to labeling difficulties. However, FCGAT is able to classify with sufficiently high performance, and the explanations for classification are considered reliable.

To verify that the attention weights reflect the contributing functions for the classification, we evaluate classification performance using only important functions with higher attention weights. This evaluation method is based on the method of Herath et al. [15]. We create subgraph datasets consisting of functions with high attention weights by varying the number of nodes, and evaluate their performance. If FCGAT focuses on functions that highly contribute to the classification, the performance of the subgraph should be close to that of the original graph (100%subgraph).

As mentioned in Section 4.2.3, we can obtain the important functions by analyzing attention weights in Set2Set. The specific procedures are as follows. First, we create a trained model using all BODMAS-8cat samples. Next, we perform category classification of BODMAS-8cat samples with the trained model and calculate the attention weights corresponding to each function in each sample. We use attention weight $\alpha'_{i,4}$ of the fourth step in Set2Set. Finally, we create subgraphs consisting of nodes from 1 to 100% of the original graph, ordered by attention weight, and calculate the accuracy of the malware categories as predicted by the trained model.

Fig. 4.3 shows the classification accuracy against Graph Size for the subgraphs, and TABLE 4.6 shows the average number of nodes and classification accuracy for the subgraphs. While the accuracy in the original graph is 95.06%, the 1% subgraph with only

[5]https://www.aldeid.com/wiki/PEiD

6 nodes on average achieves 69.67%, higher than our intuition. This result supports the claim that FCGAT focuses on functions that highly contribute to classification and implies that only about 6 functions can characterize malware.

In the experiment by Herath et al. [15], the 10% subgraph created by CFGExplainer showed 52.39% accuracy. In contrast, the 10% subgraph created by FCGAT achieves 71.73% accuracy. Note that CFGExplainer uses CFG, which has a different graph structure from our FCG. The nodes in CFG are basic blocks, which are less granular than functions, making it difficult to characterize malware with fewer basic blocks. We believe that FCGAT can reduce the analysis workload by providing the analysts with reliable explanations with fewer functions.

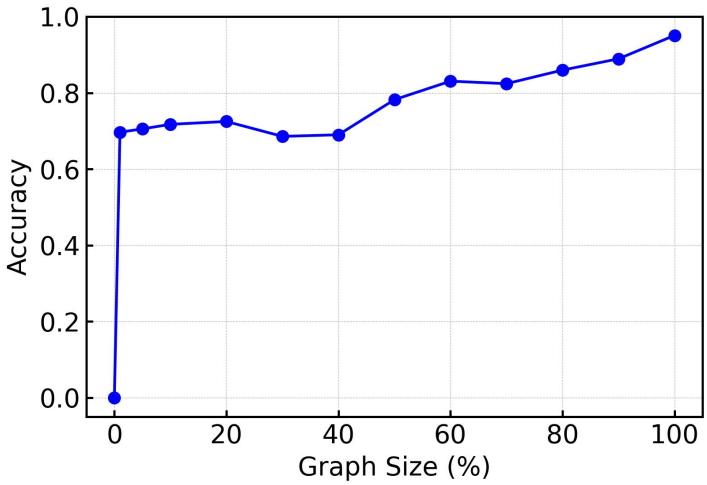

Figure 4.3: The classification accuracy of subgraphs

Table 4.6: Average number of nodes and classification accuracy of subgraphs

Graph Size (%)	Average Number of Nodes	Accuracy
1	6.2	69.67
5	28.8	70.53
10	56.9	71.73
20	113.4	72.48
30	170.1	68.57
40	226.5	68.99
50	283.0	78.20
60	339.6	83.06
70	396.2	82.40
80	452.7	85.96
90	509.3	88.92
100	565.4	95.06

Even though the average number of nodes in the 1% subgraph is only about 6, it achieves 69.67% accuracy. This means that only about 6 functions can characterize malware.

(3) Trend analysis of malware categories

In this section, we analyze the important functions and show that it is possible to analyze trends in the malware category by identifying typical functions for each category.

We extracted the most important functions for each sample by analyzing the attention weights as follows:

1. Create a trained model of FCGAT using all samples in BODMAS-8cat.
2. Perform inference on all samples with the trained model and calculate the attention weight $\alpha'_{i,4}$ of the fourth step in Set2Set.
3. Functions with the max $\alpha'_{i,4}$ are aggregated for each category. In this process, function names generated by IDA Pro are used to determine the function identities. IDA Pro uses the symbol name of the function if symbol data is available. Otherwise, the function name is generated based on the address reference. For *sub_[start address]* and *start*, which are considered to have no correspondence between function name and code, the md5 hash of the function vector is calculated and the function name is changed to *sub_[hash value]* and *start_[hash value]* before aggregation.

In this way, each sample's functions with the max attention weight are summarized for each category, and the top eight are shown in TABLE 4.7.

Table 4.7: Aggregate results for functions with max attention weights

backdoor		downloader		dropper		informationstealer	
CreateFileA	119	InternetOpenW	260	sub_460d8c2	278	GetFocus	202
fclose	105	MessageBoxA	130	GetWindowThreadProcessId	48	mciSendStringA	47
sub_4f847a1	83	mciSendStringA	77	CheckSumMappedFile	17	__dllonexit	11
SetWindowLongA	74	sub_f9da9b9	61	IIDFromString	14	free	10
GetCommandLineA	47	sub_1236153e	48	IsProcessorFeaturePresent	4	CreateFileA	9
sub_a9f1051	29	DispatchMessageA	34	sub_62922e7	3	sub_d0f95e9	8
sub_cf1fee5	18	UpdateWindow	29	__imp_VirtualProtect	2	lstrcatA	7
_fdopen	17	CoUninitialize	27	ShellExecuteA	2	GdipCreateFromHDC	7
ransomware		trojan		virus		worm	
CoRegisterMallocSpy	34	sub_a9f1051	1,316	GetActiveWindow	38	__abnormal_termination	1,314
CoReleaseMarshalData	22	__vbaUI1I4	1,036	strcpy	31	EnterCriticalSection	892
InternetReadFile	19	CloseHandle	709	start_80bc47b	15	sub_f7f9a83	407
ReadFile	12	InternetOpenW	633	?ProcessWndProcException	2	__ZNSt6locale5_ImplC2Ej	255
SetPropA	11	GetSystemDirectoryA	512	nullsub_3	2	__ZNSt6locale5_ImplC1Ej	205
SwitchDesktop	11	sub_acc2cf0	510	CoUninitialize	1	sub_d0f95e9	188
IsProcessorFeaturePresent	10	OleSetMenuDescriptor	472	GetFileVersionInfoSizeW	1	__ZNSt6locale6globalERKS_	182
CreateOleAdviseHolder	7	rtcLowerCaseVar	388	EnableMenuItem	1	SetWindowsHookExA	158

This table shows the most important functions and their counts, aggregated by category. The results provide insights into the functions typical of the malware category.

The results provide insight into the functions that are typical for each category. Especially in the case of API functions, it is easy to infer behavior from the function name. We will focus on some characteristic functions.

Backdoor is placed on the victim machine by attackers, enabling it to be controlled remotely. Thus, *GetCommandLineA* reflects the feature of reading command line parameters. This function is found in common with the malware families gbot and zegost. The user-defined function *sub_4f847a1* performs XOR operations, which Malware often uses XOR operations to obfuscate malicious data or code.

In downloader, we can see that FCGAT focused on *InternetOpenW* which is used when downloading another malware from the Internet. This result highly reflects the characteristics of downloader.

Dropper is a program that creates and drops files containing malicious code from itself. *CheckSumMappedFile* calculates a new checksum for the file and returns it with the CheckSum parameter. Malware sometimes copies a binary executable to a file in a temporary path, maps the file to memory, and then calls *CheckSumMappedFile* to verify the checksum. *__imp_VirtualProtect* and *ShellExecuteA* are used to execute shell code.

Informationstealer includes the blocker, zbot, and other families. textitGetFocus, typical of this class, gets a handle to the window that has keyboard focus. This function is found in common with blocker.

In trojan, the samples in which *sub_a9f1051* is the most important function are packed. We used PEiD to exclude as many packed samples as possible but could not exclude packers that did not match the PEiD signatures. However, FCGAT captures the characteristics of packers and tends to classify the same packers into the same class. Dealing with packed samples is future work.

We have seen important functions for malware classification. FCGAT seems to identify particularly similar samples within classes and focuses on functions common to them. As a result, we have found many API functions that were often common to multiple samples. By utilizing this method, it will be possible to comprehensively analyze the characteristics

Table 4.8: Importance ranking of function

GandCrab#1		GandCrab#2	
function	$\alpha'_{i,4}$	function	$\alpha'_{i,4}$
aes_encrypt	0.5984	aes_encrypt	0.0331
aes_decrypt	0.4015	aes_decrypt	0.0084
sub_10007BB0	2.002e-06	SetHandleInformation	0.0080
sub_10003F70	4.166e-07	GetTickCount	0.0080
sub_10004C20	7.423e-10	InitializeCriticalSection	0.0080

These samples are common to the top two important functions, *aes_encrypt* and *aes_decrypt*, which are characteristic of ransomware.

of malware categories.

(4) Case study: Ransomware/ GandCrab

GandCrab is ransomware that appeared in 2018. Here we will discuss two samples of GandCrab, GandCrab#1 and GandCrab#2, and examine which functions FCGAT focuses on.

The importance ranking of the functions and their attention weights are shown in TABLE 4.8. The functions *aes_encrypt* and *aes_decrypt* are at the top of the ranking for both samples. As the name implies, *aes_encrypt* and *aes_decrypt* functions perform encryption and decryption of AES ciphers, and well represent features of ransomware. These functions were named by IDA Pro's Lumina server, which stores metadata for well-known functions.

GandCrab#1 is a DLL file, and GandCrab#2 is an EXE file. Their program structures are quite different. However, they were correctly classified by focusing on the common functions, *aes_encrypt* and *aes_decrypt*. FCGAT seems to be able to capture the characteristics of programs that are semantically similar but structurally and syntactically different. In this case, the function names were named by the Lumina server, so it is easy to guess the behavior from the function names. However, even if it were difficult to guess from the function names (e.g., *sub_420000*), FCGAT can focus on the characteristic functions because it uses features of assembly code. We consider that these explanations can be useful for malware analysis.

4.4 Discussion

4.4.1 Effectiveness of our method

We will discuss the effectiveness of our proposed method by comparing it with existing studies. In the same way as GEMAL [2], FCGAT uses FCG for malware classification, and both have succeeded in high-performance classification in evaluation experiments (Section (3)). They differ mainly in the readout process. GEMAL uses a fully connected layer

to learn the attention weight and thus requires a constant number of nodes in the input graph. That is, the number of functions must be constant for all samples. GEMAL truncates nodes greater than 200 and zero-padded for fewer. However, the maximum number of nodes in the MalwareBazaar Dataset is 10,789 and in BIG-2015 it is 14,716. For samples with many functions, functions that characterize malware may be truncated. On the other hand, FCGAT adopts Set2Set for the readout process and uses LSTM to achieve an attention mechanism for a variable number of nodes. Therefore, we believe that FCGAT is superior to GEMAL in terms of model interpretability.

Yakura et al.'s method [14] and CFGExplainer [15] have different feature units from our method. Yakura et al. state that the byte sequence of the original executable can be identified by extracting the pixel regions of the image. However, the classified images are compressed to 64 $\times$ 64 resolution, and it is suspicious whether the corresponding byte sequence can be identified from the pixel area. In addition, this method had a classification error of 50.97% for malware families, and low classification performance was mentioned as a problem. On the other hand, our method successfully classified malware with high performance, with an F1-Score of over 98%, and both high-performance and interpretable malware classification were achieved.

CFGExplainer obtains explanations for classification in units of basic blocks. As seen in Section (2), the classification performance of CFGExplainer is significantly degraded for subgraphs with a small number of nodes because basic blocks are less granular than functions. Therefore, to identify subgraphs that highly contribute to classification, it is necessary to increase the number of nodes in the subgraph, which increases the effort of manual analysis. In this regard, FCGAT can identify subgraphs with fewer nodes and higher contribution to classification, with an average of 6 nodes still performing 69.67% of accuracy.

These existing studies provide analysis cases for individual samples, but they do not analyze to characterize the class of the entire dataset. That is, some samples can provide interpretable explanations for classification, but others may not be well explained. We have confirmed that the important functions reflect the trend of malware categories in Section (3). FCGAT extracts the explanations on a per-function basis, which enables aggregating the characteristic functions for each class and identifying trends.

4.4.2 Limitations

Similar to existing static analysis methods, our proposed method is challenging to perform malware classification for the malware obfuscated by packers. The packed program contains a decompression processing code for each type of packer. Therefore, it is likely to be able to capture the characteristics of packers and thus be applicable to packer estimation task.

4.5 Conclusion

We have presented FCGAT, a malware classification method that can explain classifications by functions. Evaluation experiments showed that FCGAT classifies malware with

sufficiently high performance compared to the latest methods. We then analyzed the explanations for the classification results based on the attention weight of the classification model. The results show that these explanations provide functions that reflect the malware features. In addition, we performed malware classification on the subgraphs with top important functions. Surprisingly, our result shows that only top 6 (average per sample) of important functions contribute to the classification result with almost 70% of accuracy. Malware families can be characterized by considerably fewer functions than our intuition. FCGAT is expected to improve the efficiency of malware analysis and comprehensive malware trend analysis. For example, it can be combined with existing tools, such as IDA Pro, to highlight functions specific to the malware.

Chapter 5

RevLlama: Recovering Function Names via Rationale Distillation from Large to Small Language Models

5.1 Introduction

Function name recovery—the task of restoring meaningful names to stripped binary functions—is a critical step in binary reverse engineering. Unlike source code, which contains human-readable identifiers and documentation, compiled binaries often lack semantic information, forcing analysts to infer function behavior from cryptic, low-level code.

Recent advances in Large Language Models (LLMs) such as GPT-4 and Gemini have demonstrated impressive code understanding capabilities, raising the possibility of using LLMs to support reverse engineering tasks. However, two fundamental obstacles hinder the direct application of LLMs to binary analysis. First, most LLMs are trained primarily on source code from open repositories such as GitHub, and thus lack exposure to decompiled or low-level code with obfuscated patterns. Second, LLMs are typically accessed through cloud-based APIs, making them unsuitable for sensitive use cases like malware analysis, where uploading proprietary or malicious binaries to external servers is unacceptable.

To bridge this gap, this work proposes **RevLlama**, a locally deployable LLM specialized for function name recovery from decompiled binary code.

5.2 Method

We construct a high-performance function name prediction model by efficiently transferring knowledge from an LLM to an SLM. This section first presents an overview of our approach, followed by detailed explanations of each component.

5.2.1 Overview

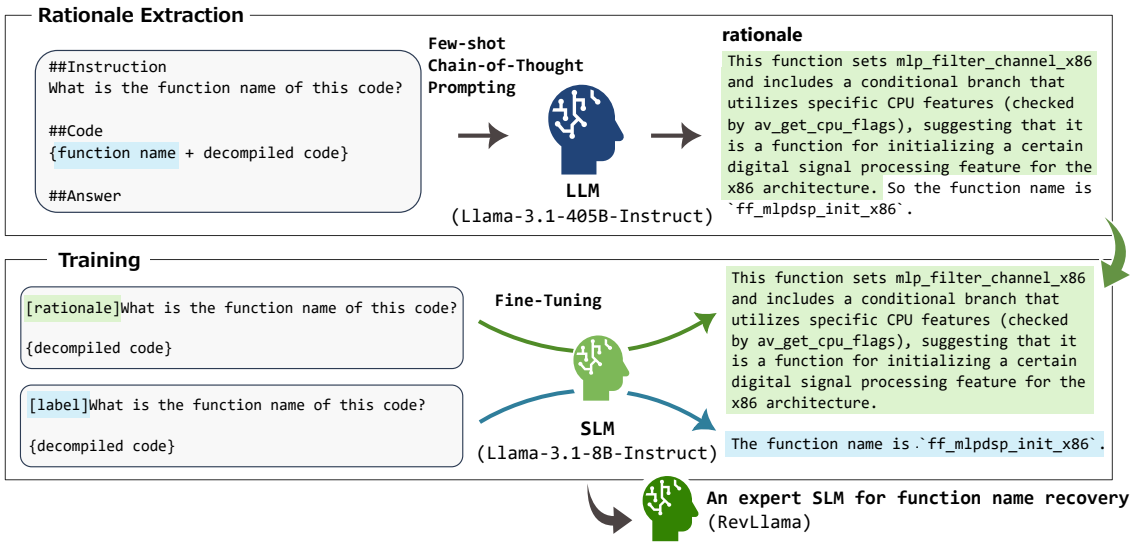

Figure 5.1: Overview of the RevLlama construction method

Our approach specializes the concept of *Distilling Step by Step* [34] for the function name prediction task. As shown in Figure 5.1, this method consists of two main steps:

1. **Rationale Extraction**: An LLM is employed to generate the thought process (i.e., reasoning rationale) during function name prediction from decompiled code. Chain-of-Thought prompting is utilized to encourage step-by-step reasoning.
2. **Knowledge Transfer**: An SLM is fine-tuned via multitask learning using the extracted reasoning rationale and function names. This process enables the efficient transfer of reasoning capabilities from the LLM to the smaller model.

This approach facilitates the development of an SLM that inherits the sophisticated reasoning capabilities of an LLM for function name prediction.

5.2.2 Rationale Extraction

This phase captures the reasoning process of the LLM during function name prediction. For each decompiled function, the model generates natural language explanations that analyze key elements such as variable usage, control flow structures, and algorithmic patterns to justify its predictions.

The prompt construction process employs a labeled dataset $(x_i, y_i) \in D$, where each prompt comprises three components: decompiled code x^{p}, reasoning rationale r^{p}, and the corresponding function name y^{p}. The framework uses few-shot prompting [35] with chain-of-thought [36] examples to guide the LLM in its reasoning. The output format follows the instruction: `"Please give the function name after the rationale"`.

During this phase, the ground truth function names y_i, derived from the source code, are provided. These source-level function names serve as reliable references to ensure that the generated reasoning aligns with the actual semantics of the functions. By using precise source-level function names as ground truth, the framework ensures high-quality reasoning and accurate function name prediction.

The LLM outputs are processed by removing final function name declarations (for example, `The function name is 'xxx'`) to isolate the pure reasoning process. These rationales serve as training signals for the knowledge transfer phase, providing detailed insights into function behavior and purpose.

5.2.3 Knowledge Transfer

Training of the student SLM is performed using multitask learning on the processed dataset $(x_i, y_i, r_i) \in D$. The model $f(x_i) \rightarrow (\hat{y}_i, \hat{r}_i)$ is trained to generate both function names $\hat{y}_i$ and reasoning rationales $\hat{r}_i$ for each input x_i.

The training objective combines the tasks of function name prediction and rationale generation. The total loss $\mathcal{L}$ is defined as:

$$\mathcal{L} = \mathcal{L}_{\text{label}} + \mathcal{L}_{\text{rationale}}$$

$$\mathcal{L}_{\text{label}} = \frac{1}{N} \sum_{i=1}^{N} \ell(f(x_i), \hat{y}_i),$$

$$\mathcal{L}_{\text{rationale}} = \frac{1}{N} \sum_{i=1}^{N} \ell(f(x_i), \hat{r}_i)$$

Task identifiers (`[label]` or `[rationale]`) are prepended to the input during training, enabling the model to learn the appropriate output format for each task.

5.2.4 Inference

During inference, the `[label]` identifier is prepended to the input decompiled code. The model then generates function names in a tuned format (`The function name is 'xxx'`), ensuring compatibility with established reverse engineering workflows.

5.3 Evaluation

We evaluate the effectiveness of RevLlama, our proposed function name prediction model, by conducting two experiments that address the following research questions (RQs):

RQ1: How effective is RevLlama's knowledge distillation approach for function name recovery?

RQ2: How does RevLlama's performance compare to existing approaches and state-of-the-art LLM?

5.3.1 Experimental Setup

This section outlines the experimental settings shared across both experiments.

(1) Implementation

We used Llama-3.1-405B-Instruct[1] as the teacher LLM, one of the largest publicly available models with permissive licensing terms. We employed Llama-3.1-8B-Instruct[2] as the student SLM. Our framework can be adapted to other model architectures.

The experiments were conducted on a system running 64-bit Ubuntu 22.04, equipped with an Intel(R) Xeon(R) CPU @ 2.20GHz and an NVIDIA A100-SXM4-40GB GPU. For the inference (evaluation) phase, we utilized vllm [37] to accelerate the generation process. To ensure reproducibility, we set the temperature parameter to 0 and used greedy decoding, minimizing probabilistic variations in model outputs.

(2) Rationale Extraction

For rationale extraction from the teacher model, we used Llama-3.1-405B-Instruct via the Fireworks AI[3] API. As shown in Figure 5.2, we guide the model's thinking process for function name prediction using few-shot prompting. Specifically, we provided pairs of decompiled code and function names of the source code, prompting the model to explain why each function name was appropriate. To facilitate accurate rationale generation, function names were left unmasked in the decompiled code, enabling the model to reference the source code function names. The reasoning rationales were extracted by removing the final sentence of the output, which contained the function name declaration.

[1] https://huggingface.co/meta-llama/Llama-3.1-405B-Instruct
[2] https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct
[3] https://fireworks.ai/

```
##Instruction
What is the function name of this code?

##Code
__int64 __fastcall ff_mlpdsp_init_x86(_QWORD *a1)
{
__int64 result; // rax

result = av_get_cpu_flags() & 1;
if ( (_DWORD)result )
{
result = (__int64)a1;
*a1 = mlp_filter_channel_x86;
}
return result;
}

##Answer
This function sets mlp_filter_channel_x86 and includes a conditional branch
that utilizes specific CPU features (checked by av_get_cpu_flags), suggesting
that it is a function for initializing a certain digital signal processing
feature for the x86 architecture. So the function name is `ff_mlpdsp_init_x86`.
```

(a) Demonstration Example 1

```
##Instruction
What is the function name of this code?

##Code
__int64 __fastcall rpl_remove_routes(__int64 a1)
{
__int64 result; // rax
__int64 i; // [rsp+18h] [rbp-8h]

result = uip_ds6_route_list_head();
for ( i = result; i; i = result )
{
if ( a1 == *(_QWORD *)(i + 56) )
{
uip_ds6_route_rm(i);
result = uip_ds6_route_list_head();
}
else
{
result = list_item_next(i);
}
}
return result;
}

##Answer
This function performs route removal operations in a routing list structure,
iterating over each route and removing those matching a specified condition
(a1 == *(_QWORD *)(i + 56)). So the function name is `rpl_remove_routes`.
```

(b) Demonstration Example 2

Figure 5.2: Demonstration examples for rationale extraction

(3) Fine-Tuning

For fine-tuning the student model, we employed LoRA [38] and organized the training data using the `[label]` and `[rationale]` templates. Based on preliminary experiments, we set the number of epochs to 1 and the learning rate to 1×10^{-4}. The LoRA hyperparameters were configured with a rank of $r = 32$ and a scaling coefficient of $\alpha = 16$. Losses were

computed solely on the output portion, and after fine-tuning, the LoRA weights were merged into the base model to facilitate efficient inference using vllm.

5.3.2 Evaluation Metrics

To comprehensively assess our method, we employ two types of evaluation metrics. Traditional word-matching metrics, commonly used in previous studies, are limited in evaluating expressions that convey similar meanings. For example, function names such as `get_value` and `fetch_data` convey similar meanings yet may receive a score of zero due to the absence of overlapping words. To address this limitation, we supplement conventional word-matching metrics with a metric based on Sentence-BERT, which captures semantic similarities between function names.

(1) Word-Level Matching

Following previous studies [17, 18], we compute word-level precision, recall, and F1-score between the predicted and ground truth function names. Function names are first segmented into words based on three common naming conventions: `CamelCase`, `snake_case`, and `kebab-case`, and then normalized to lowercase. For example, `getUserValue` is segmented into the words `get`, `user`, and `value`. The scores for each sample are computed based on these word sets, and the final metric is obtained by averaging the scores across all samples.

(2) Sentence-BERT

To evaluate semantic similarities between function names, we employ a metric based on Sentence-BERT [39]. As illustrated in Figure 5.3, function names are first segmented into words using the same method as in the word-level matching. These word sequences are then converted into 768-dimensional vectors using the all-mpnet-base-v2 model[4], which captures semantic relationships between words. Finally, semantic similarity is quantified by computing the cosine similarity between the predicted and ground truth vectors, and the scores are averaged across all samples.

[4]https://huggingface.co/sentence-transformers/all-mpnet-base-v2

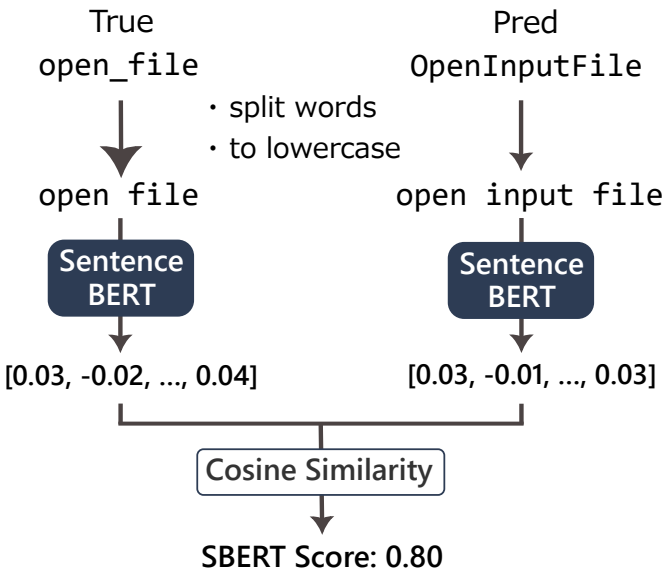

Figure 5.3: Calculation method for Sentence-BERT scores

5.3.3 Exp. 1: Effect of Learning with Reasoning Rationale

We evaluate the effectiveness of incorporating reasoning rationale by comparing the performance of RevLlama against baseline models.

(1) Dataset

We employ DIRT dataset [40], which was created by decompiling C-language GitHub repositories using IDA Pro. The dataset comprises high-quality pairs of decompiled code and their corresponding function names. In our experiments, 40,000 functions are used for training and 5,000 for testing, with the function names in the decompiled code replaced by the token `<FUNC_NAME>`.

(2) Baselines

Using Llama-3.1-8B-Instruct as the base model, we define the following three baselines:

- **Base Model**: The base model without additional training. To minimize formatting errors, the following instruction prompt was used:

```
What is the original function name of this code?
The original name is masked by <FUNC_NAME>.
Follow the output format:  "The function name is ẋxx."

{code}
```

- **FT with Label**: A model fine-tuned using only function names, representing a conventional approach without incorporating reasoning evidence.
- **FT with CoT**: A model fine-tuned directly on LLM outputs generated with Chain-of-Thought prompting, learning both reasoning evidence and function names simul-

taneously as a single task.

We evaluate the effect of reasoning evidence by comparing RevLlama with the FT with Label baseline and assess the effect of multi-task learning by comparing it with the FT with CoT baseline.

(3) Results

Table 5.1: Performance of function name prediction

	SBERT	Precision	Recall	F1
base_model	0.391	0.353	0.279	0.312
FT with label	0.474	0.390	0.365	0.377
FT with CoT	0.494	0.414	0.396	0.405
RevLlama	**0.528**	**0.454**	**0.442**	**0.448**

Top scores for each metric are shown in **bold**.

Table 5.1 shows the evaluation results for RevLlama and the baseline models. RevLlama consistently outperformed all baselines across every evaluation metric. Specifically, it achieved an SBERT score of 0.528, a 6.9% improvement over the next best model (FT with CoT at 0.494). For traditional metrics, RevLlama obtained an F1 score of 0.448, a 10.6% improvement over FT with CoT (0.405).

The progression in performance across the different training approaches highlights clear improvements at each stage. Fine-tuning with labels increased the base model's F1 score by 20.8% (from 0.312 to 0.377). Incorporating Chain-of-Thought reasoning further boosted the F1 score to 0.405. Finally, RevLlama's knowledge distillation approach yielded the most substantial improvements, with consistent gains across all metrics, demonstrating its effectiveness in capturing both semantic understanding and naming accuracy.

RevLlama's superior performance compared to FT with CoT suggests that training on separate datasets for function names and reasoning rationales is more effective than learning them together as a single task. This separation likely enables the model to better capture the unique characteristics of each aspect, leading to improved overall results.

5.3.4 Exp. 2: Comparison with Existing Methods

(1) Dataset

To ensure fair comparison with existing methods, we conducted evaluation experiments on the widely-used Nero dataset [16]. Since the dataset is provided in executable format, we decompiled it using IDA Pro and extracted decompiled functions with matching file names and function names from the training and test sets. A dataset for function name prediction was constructed by masking function names in the decompiled code with `<FUNC_NAME>`.

(2) Baselines

We compared RevLlama against three existing methods specialized for binary analysis: SymLM [17], Nero [16], and HexT5 [18]. For these methods, performance values (Precision, Recall, F1-score) were adopted from their respective papers, evaluated on the same dataset. Additionally, we measured and included the performance of GPT-4o, a state-of-the-art LLM, for comparison.

(3) Results

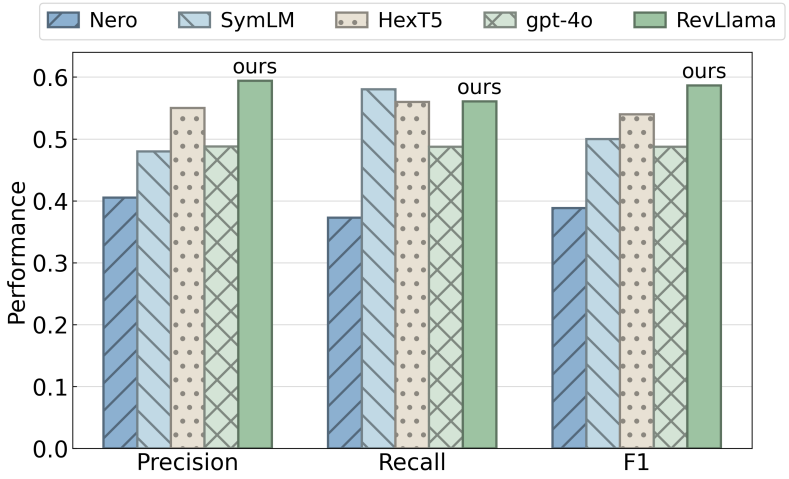

Figure 5.4: Evaluation results on the Nero dataset

As shown in Figure 5.4, RevLlama outperformed all existing methods and GPT-4o across evaluation metrics. RevLlama achieved the highest F1 score of 0.587, representing an 8.7% improvement over the previous state-of-the-art method, HexT5 (F1: 0.54). In precision, RevLlama scored 0.594, marking a 7.9% improvement over HexT5 (Precision: 0.55).

Notably, RevLlama also surpassed GPT-4o (F1: 0.488) by 20.3%. These results demonstrate that task-specific knowledge distillation from LLMs can outperform general-purpose LLMs, offering superior performance while maintaining compatibility with secure local environments.

5.4 Qualitative Analysis

To better understand the characteristics of RevLlama's predictions, we analyze concrete examples of predicted function names. Table 5.2 shows examples of function names predicted by RevLlama and GPT-4o, along with their Sentence-BERT scores. Analysis of RevLlama's predictions reveals that it more accurately captures the functionality contained in the original function names.

Table 5.2: Examples of predicted function names and their SBERT scores

True	RevLlama		GPT-4o	
	function name	SBERT	function name	SBERT
__update_last_label	set_last_label	0.872	set_label	0.662
utf8len	utf8_strlen	0.727	getStringLength	0.306
skip_past_newline_1	skip_past_line	0.811	find_next_line	0.633
do_tutorial	do_tutorial	1.000	process_script_line	0.099

For instance, for the function `utf8len`, RevLlama predicts `utf8_strlen`. This prediction retains not only the string length calculation functionality but also the contextual information about handling UTF-8 encoding. In contrast, GPT-4o predicts `getStringLength`, which captures the basic functionality but lacks information about character encoding. This difference in prediction is reflected in the SBERT scores, with RevLlama scoring 0.727 and GPT-4o scoring 0.306.

Similarly, for `skip_past_newline_1`, RevLlama predicts `skip_past_line`, accurately expressing the essential line-advancing operation. In comparison, GPT-4o's prediction of `find_next_line`, while referring to a similar operation, results in a more general and ambiguous expression. A notable example is the case of `do_tutorial`, where RevLlama achieves perfect prediction (SBERT score 1.000), while GPT-4o makes a less relevant prediction of `process_script_line`.

These examples demonstrate that by leveraging reasoning evidence, RevLlama attains a deeper understanding of a function's behavior and generates more precise function names. In particular, it effectively exploits the contextual information present in decompiled code to yield more specific and accurate predictions.

5.5 Discussion

This section discusses the effectiveness and limitations of our approach.

5.5.1 Effectiveness

Our experimental results highlight the effectiveness of knowledge distillation with reasoning rationales for function name prediction. RevLlama achieves substantial improvements over existing methods and GPT-4, with up to a 51.0% increase in F1 score compared to prior approaches. These gains can be attributed to three key aspects of our knowledge distillation methodology.

First, learning reasoning rationales enables RevLlama to develop enhanced program comprehension capabilities. The model systematically analyzes decompiled code, leveraging both structural patterns and semantic relationships. This facilitates more accurate interpretation of function behaviors, resulting in precise name predictions.

Second, our approach offers significant practical advantages in computational efficiency and deployment flexibility. By utilizing fine-tuning instead of in-context learning, we substantially reduce input length requirements, optimizing resource utilization. The model

can operate efficiently in CPU-only environments using standard quantization techniques, making it accessible for security analysts in restricted settings.

Third, RevLlama provides superior output control compared to general-purpose LLMs. While LLMs often produce inconsistent or verbose outputs, RevLlama generates standardized function names suitable for seamless integration with reverse engineering tools such as IDA Pro or Ghidra. Additionally, reasoning rationales can be displayed as code comments, enabling practical features like automatic function renaming.

5.5.2 Limitations

Our approach has several limitations that warrant future exploration. First, the model currently treats each function independently, without considering function call relationships or broader program-wide context. Real-world programs often involve complex interdependencies that could enhance name prediction accuracy if leveraged effectively.

Second, while our evaluation demonstrates effectiveness on benchmark datasets, further validation on real-world malware samples is necessary. Malware often employs code obfuscation techniques that pose significant challenges to our current method.

5.6 Conclusion

In this paper, we presented RevLlama, an approach to function name prediction in binary analysis that combines LLM reasoning capabilities with local execution requirements. Through knowledge distillation from LLMs, RevLlama achieves superior performance compared to both existing methods and GPT-4o, while maintaining the ability to operate in secure local environments. Our experimental results demonstrate a 20.3% improvement in F1 score over GPT-4o, validating the effectiveness of our approach.

Future work could focus on incorporating program-wide context and addressing challenges posed by obfuscated code to further enhance prediction accuracy. We believe this research represents a significant step forward in enabling secure and efficient binary analysis, making sophisticated capabilities accessible in restricted environments.

Chapter 6

Empowering LLM-based Malware Analysis with Synthetic Code

6.1 Introduction

Chapter 5 introduced RevLlama, a local language model trained via rationale distillation for function name recovery. While effective on general-purpose binaries, its performance declines when applied to real malware, due to a lack of domain-specific features in the training data. Malware differs significantly from benign code—it often employs obfuscation, anti-analysis techniques, and uncommon API usage. Furthermore, real malware source code is rarely available, making it difficult to train or evaluate models specialized for malware analysis.

To overcome these limitations, this chapter introduces a method for generating synthetic malware code guided by threat intelligence, enabling large and small language models to better understand malicious behavior. In addition, a new benchmark dataset, **MalFuncBench**, is introduced to evaluate function-level analysis capabilities on real malware samples.

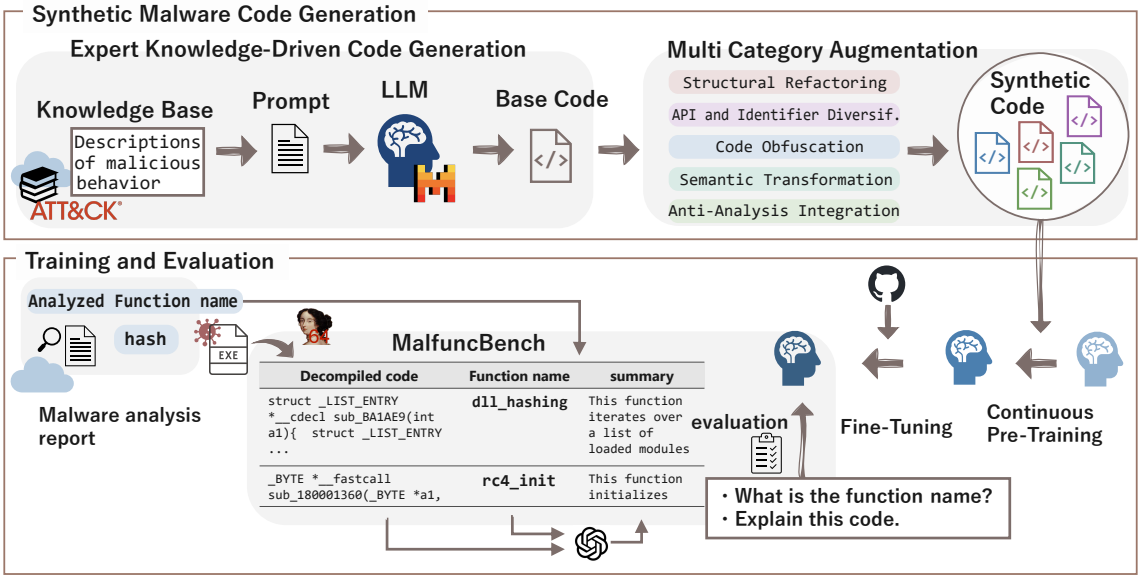

Figure 6.1: Overview of our synthetic malware generation framework

6.2 A Framework for Synthetic Malware Code Generation

This section presents our framework for generating synthetic malware code that addresses the data scarcity challenge in training LLMs for malware analysis. We first formalize the problem, then describe the framework components.

6.2.1 Problem Formulation

Let $\mathcal{T}$ denote the set of attack techniques from expert knowledge bases. Our goal is to generate a synthetic dataset $\mathcal{D}_{\text{syn}} = \{(c_i, n_i)\}_{i=1}^{N}$ where c_i is a malware function code containing its function name n_i within the code itself.

The generation process can be formalized as a two-stage transformation:

$$\mathcal{T} \xrightarrow{G_{\text{init}}} \mathcal{C}_{\text{base}} \xrightarrow{A_{\text{multi}}} \mathcal{D}_{\text{syn}} \tag{6.1}$$

where G_{init} is the initial code generation function that transforms expert knowledge into base implementations, and A_{multi} is the multi-category augmentation function that creates diverse variants.

6.2.2 Framework Overview

Figure 6.1 illustrates our two-stage framework for synthetic malware generation. The framework systematically transforms threat intelligence into diverse code samples suitable for training malware analysis models.

The framework operates through two main components:

1. **Expert Knowledge-Driven Generation** (G_{init}): This component leverages threat intelligence databases to guide initial code generation. By grounding generation in documented attack techniques, we ensure that synthetic samples reflect real-world malware behaviors rather than arbitrary malicious patterns.

2. **Multi-Category Augmentation** (A_{multi}): To create the diversity necessary for effective model training, this component applies systematic transformations across five complementary dimensions: structural refactoring, API and identifier diversification, code obfuscation, semantic transformation, and anti-analysis integration.

The design principle underlying our augmentation strategy draws from research on LLM knowledge acquisition [41], which demonstrates that presenting the same concept through multiple paraphrases improves knowledge retention. Similarly, our framework generates multiple implementations of each malicious behavior, enabling models to learn robust representations that generalize across implementation variations. Each generated function maintains its core malicious functionality while exhibiting diverse coding patterns, API usage, and obfuscation techniques commonly observed in real malware.

The framework produces training data for continuous pre-training. By pre-training on synthetic malware, models acquire domain-specific knowledge about malware behaviors, API patterns, and code structures. This domain adaptation prepares models for downstream tasks such as function naming and code summarization.

6.2.3 Expert Knowledge-Driven Code Generation

(1) Knowledge Extraction

Threat intelligence databases provide structured information about malware behaviors. Taking MITRE ATT&CK as an example, each malware family is associated with multiple techniques, where each technique includes its name and usage description. From a knowledge base $\mathcal{K}$, a set of technique descriptions $\mathcal{T} = \{t_1, t_2, ..., t_m\}$ is extracted, with each technique t_i represented as:

$$t_i = \langle \text{name}_i, \text{use}_i, \text{desc}_i \rangle \tag{6.2}$$

where name_i is the technique name (e.g., "Process Injection"), use_i describes how the technique is employed by malware, and desc_i provides implementation details.

Table 6.1 presents example techniques extracted from threat intelligence sources. While we utilize MITRE ATT&CK in our implementation, the extraction process is designed to accommodate various threat intelligence formats, enabling integration with multiple knowledge sources.

(2) Code Generation Function

The initial generation function $G_{\text{init}} : \mathcal{T} \rightarrow \mathcal{C}_{\text{base}}$ transforms technique descriptions into code using an LLM $\mathcal{L}$ with a structured prompt template P:

Table 6.1: Example attack techniques extracted from threat intelligence

Technique Name	**Technique Use**	**Description**
Obfuscated Files or Information	AppleJeus has XOR-encrypted collected system information prior to sending to a C2. AppleJeus has also used the open source ADVObfuscation library for its components.	AppleJeus is a family of downloaders initially discovered in 2018 embedded within trojanized cryptocurrency applications. AppleJeus has been used by Lazarus Group, targeting companies in the energy, finance, government, industry, technology, and telecommunications sectors, and several countries including the United States, United Kingdom, South Korea, Australia, Brazil, New Zealand, and Russia. AppleJeus has been used to distribute the FALLCHILL RAT.
Shared Modules	Bumblebee can use LoadLibrary to attempt to execute GdiPlus.dll.	Bumblebee is a custom loader written in C++ that has been used by multiple threat actors, including possible initial access brokers, to download and execute additional payloads since at least March 2022. Bumblebee has been linked to ransomware operations including Conti, Quantum, and Mountlocker and derived its name from the appearance of "bumblebee" in the user-agent.
Process Discovery	Avaddon has collected information about running processes.	Avaddon is ransomware written in C++ that has been offered as Ransomware-as-a-Service (RaaS) since at least June 2020.

$$G_{\text{init}}(t_i) = \mathcal{L}(P(t_i)) \tag{6.3}$$

The prompt template P is carefully designed to extract compilable C code that accurately implements the described technique:

```
 Write C code that implements the functionality based on the following
description.  Generate code that can be compiled on an x64 Windows environment.
Don't explain anything.

Feature to be implemented:  "{name_i}"
{use_i}
{desc_i}
```

This structured prompting ensures generated code aligns with documented attack patterns while maintaining compilation compatibility. The resulting base implementations serve as seeds for subsequent augmentation.

6.2.4 Multi-Category Augmentation

Real-world malware exhibits significant implementation diversity as authors employ various coding styles and evasion techniques. To capture this diversity, five augmentation categories are defined that transform code while preserving malicious functionality.

(1) Augmentation Categories

Five augmentation functions $\mathcal{A} = \{A_1, A_2, ..., A_5\}$ are defined:

- A_1: **Structural Refactoring** - Rearranges functions, classes, and control flow
- A_2: **API and Identifier Diversification** - Swaps API calls and renames identifiers
- A_3: **Code Obfuscation** - Applies control-flow flattening and indirect calls
- A_4: **Semantic Transformation** - Rewrites using different algorithms or constructs
- A_5: **Anti-Analysis Integration** - Embeds anti-debugging and environment checks

Each augmentation function $A_j : \mathcal{C} \to \mathcal{C}'$ transforms code while preserving functional behavior:

$$\forall c \in \mathcal{C},\ \text{behavior}(A_j(c)) = \text{behavior}(c) \tag{6.4}$$

where $\mathcal{C}$ and $\mathcal{C}'$ denote the original and transformed code spaces respectively.

(2) Augmentation Process

For each base implementation $c_i^{(0)} \in \mathcal{C}_{\text{base}}$, we generate k variants through systematic augmentation. The variant set for each base code is:

$$\mathcal{V}_i = \{c_i^{(1)}, c_i^{(2)}, ..., c_i^{(k)}\} \tag{6.5}$$

where each variant $c_i^{(j)}$ is produced by applying a subset of augmentation functions:

$$c_i^{(j)} = (A_{i_n} \circ ... \circ A_{i_2} \circ A_{i_1})(c_i^{(0)}) \tag{6.6}$$

with $\{A_{i_1}, A_{i_2}, ..., A_{i_n}\} \subseteq \mathcal{A}$ selected by function $\sigma(j)$.

In our implementation, we employ two selection strategies:

- **Random selection**: $\sigma_{\text{rand}}(j)$ randomly selects one augmentation category:

$$\sigma_{\text{rand}}(j) = \{A_i\} \text{ where } i \sim \text{Uniform}(1, 5) \tag{6.7}$$

- **Full application**: $\sigma_{\text{full}}(j)$ applies each augmentation category individually:

$$\sigma_{\text{full}}(j) = \{A_j\} \text{ where } j \in \{1, 2, 3, 4, 5\} \tag{6.8}$$

Each augmentation is implemented by prompting the LLM with category-specific instructions. For a subset $S \subseteq \mathcal{A}$ of augmentation categories, the transformation prompt is:

```
Rewrite this code diversely using these approaches:
{categories in S}

Original code:
{c_i^(0)}

Generate only the transformed code without any explanations.
```

This prompting strategy leverages the LLM's code understanding while enforcing semantic preservation through explicit constraints.

6.2.5 Dataset Construction

The complete synthetic dataset aggregates all generated variants:

$$\mathcal{D}_{\text{syn}} = \bigcup_{i=1}^{m} \{(c_i^{(j)}, n_i^{(j)}) \mid c_i^{(j)} \in \mathcal{V}_i\} \tag{6.9}$$

Function names $n_i^{(j)}$ naturally evolve during transformation. For example, API diversification might transform `inject_process` to `inject_process_nt` when substituting `VirtualAllocEx` with `NtAllocateVirtualMemory`, reflecting the implementation change.

The dataset scales as $|\mathcal{D}_{\text{syn}}| = m \times k$, where $m = |\mathcal{T}|$ represents the number of base techniques and k denotes variants per technique. This enables generation of arbitrarily large datasets limited only by computational resources.

This systematic augmentation ensures that models trained on synthetic data encounter diverse implementation patterns, improving their ability to generalize across real-world malware variations while maintaining behavioral authenticity.

6.3 MalFuncBench: A Benchmark for Malware Code Understanding

To evaluate the effectiveness of synthetic malware code for training LLMs, we introduce MalFuncBench—the first benchmark dataset specifically designed for function-level malware understanding. The dataset derives from expert malware analysis reports, providing high-quality ground truth for evaluating model performance on real-world reverse engineering tasks.

6.3.1 Benchmark Tasks

Let $\mathcal{F}$ denote the space of decompiled malware functions. We define two complementary tasks that reflect essential reverse engineering activities:

(1) Task 1: Function Name Prediction

Given a decompiled function $f \in \mathcal{F}$ with obfuscated identifiers, predict the semantically meaningful name n that captures its functionality:

$$\text{Task}_1 : f \mapsto n \tag{6.10}$$

This task evaluates a model's ability to recognize malware-specific behavioral patterns from low-level code. In compiled binaries, functions appear with auto-generated names like `sub_401000` that provide no semantic information. Expert analysts assign meaningful names based on their understanding of the code—for instance, naming a sandbox detection routine as `check_sandbox` or `detect_analysis_env`. Accurate name prediction requires understanding both code semantics and malware-specific conventions.

(2) Task 2: Function Summarization

Given a decompiled function $f \in \mathcal{F}$, generate a natural language summary s describing its behavior:

$$\text{Task}_2 : f \mapsto s \tag{6.11}$$

While function names provide concise labels, summaries offer comprehensive behavioral descriptions. A good summary identifies the function's primary purpose, documents key API calls and their parameters, describes input/output relationships, and explains the malicious intent. This task measures a model's ability to produce human-readable explanations that aid analysts in understanding complex malware behaviors.

6.3.2 Dataset Construction

(1) Data Collection from Expert Reports

We systematically collected malware analysis reports from security vendors' technical blogs, incident response documentation, and threat intelligence platforms. Reports were

selected based on three criteria: (1) detailed function-level reverse engineering analysis, (2) availability of corresponding malware samples for validation, and (3) comprehensive technical documentation with code snippets. The complete list of utilized reports is provided in Appendix 7.

From each report, we extracted functions that experts explicitly analyzed and assigned meaningful names. These functions implement core malware capabilities including anti-analysis checks, cryptographic operations, command-and-control communication, persistence mechanisms, and payload deployment. The expert-assigned names encode domain knowledge about each function's role within the malware's operation.

(2) Sample Processing and Annotation

We obtained malware binaries from VirusTotal, MalwareBazaar, and other public repositories, verifying sample integrity through SHA256 hash matching against report metadata. Using IDA Pro with the Hex-Rays decompiler, we extracted decompiled C code for each expert-analyzed function, focusing on x86/x64 Windows PE files which constitute the majority of documented samples.

A critical preprocessing step addressed the challenge of generic function names in decompiled output. Decompiled code frequently contains calls to other functions with meaningless identifiers (e.g., `sub_404520`), which disrupts code comprehension. When expert annotations provided mappings for these called functions, we systematically renamed them to preserve semantic context. This preprocessing is essential because malware functions often implement complex behaviors through interactions with custom subroutines.

For the summarization task, we employed a semi-automated approach to ensure accuracy. Initial summaries were generated using GPT-4o with both the decompiled code and expert-assigned function name as input. Security researchers then manually reviewed each summary, verifying consistency with the malware's documented capabilities and correcting any inaccuracies. This human-in-the-loop process ensures that summaries accurately reflect expert understanding while maintaining annotation efficiency.

The resulting dataset contains 47 unique functions extracted from 4 distinct malware families, representing diverse attack techniques and implementation strategies. Each entry includes the decompiled function code, expert-assigned name, and validated natural language summary.

6.3.3 Evaluation Metrics

To comprehensively assess model performance on malware understanding tasks, we employ complementary metrics that capture both exact matching accuracy and semantic equivalence. This multi-metric approach addresses the inherent flexibility in how malware behaviors can be described.

(1) Function Name Prediction Metrics

Function naming in reverse engineering exhibits significant variation—multiple names can correctly describe the same functionality. For instance, `get_config` and `fetch_settings` express identical concepts but share no common tokens. To address this challenge, we combine traditional word-matching metrics with semantic similarity measures.

Word-Level Matching Following established practices in code understanding [17, 18], we compute word-level precision, recall, and F1-score between predicted and ground truth names. Function names are tokenized according to common programming conventions:

- `CamelCase`: `getUserValue` → {`get`, `user`, `value`}
- `snake_case`: `decrypt_config` → {`decrypt`, `config`}
- `kebab-case`: `check-sandbox` → {`check`, `sandbox`}

Tokens are normalized to lowercase before comparison. Let P and G denote the token sets for predicted and ground truth names respectively. We compute:

$$\text{Precision} = \frac{|P \cap G|}{|P|}, \quad \text{Recall} = \frac{|P \cap G|}{|G|}, \tag{6.12}$$

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{6.13}$$

Semantic Similarity via Sentence-BERT To capture semantic equivalence between functionally similar names, we employ Sentence-BERT [39]. Function names are first tokenized into word sequences, then encoded into 768-dimensional dense vectors using the `all-mpnet-base-v2` model. Semantic similarity is computed as:

$$\text{Sim}_{\text{SBERT}}(p, g) = \frac{\mathbf{e}_p \cdot \mathbf{e}_g}{||\mathbf{e}_p|| \cdot ||\mathbf{e}_g||} \tag{6.14}$$

where $\mathbf{e}_p$ and $\mathbf{e}_g$ are the Sentence-BERT embeddings for predicted and ground truth names. This metric effectively captures relationships between semantically related terms prevalent in malware analysis.

(2) Function Summarization Metrics

Natural language summaries require evaluation methods that balance content accuracy with linguistic flexibility. We employ two complementary metrics:

- **BLEU-4** [42]: Measures n-gram overlap (up to 4-grams) between generated and reference summaries. While BLEU rewards exact phraseology, it may undervalue accurate paraphrases common in technical descriptions.
- **BERTScore** [43]: Computes token-level similarity using contextual embeddings from pre-trained language models. BERTScore better captures semantic equivalence between technical synonyms and alternative phrasings, making it particularly suitable for evaluating malware behavior descriptions where multiple valid explanations exist.

By combining surface-level matching with semantic understanding metrics, our evaluation framework provides nuanced assessment of model capabilities on malware code understanding tasks. This approach acknowledges that effective malware analysis requires both precision in technical terminology and flexibility in conceptual understanding.

6.4 Evaluation

6.4.1 Research Questions

Our evaluation addresses the following research questions:

- **RQ1:** How effectively does synthetic malware code improve model performance on malware understanding tasks compared to state-of-the-art baselines?
- **RQ2:** Can synthetic data simulate the distribution of real malware code?

RQ1 evaluates whether our synthetic data generation framework can enhance LLM capabilities for malware analysis tasks. We compare our approach against commercial LLMs (GPT-4o), models trained without synthetic data, and ablated versions to isolate the contribution of each component.

RQ2 investigates whether systematically generated synthetic malware can capture the essential characteristics of real malware for training purposes. We compare models trained on real malware source code, synthetic data only, and combined datasets to assess distributional similarity and training effectiveness. Due to the complexity of analyzing all statistical properties, we focus our evaluation on function naming and summarization tasks as a representative indicator of distribution matching.

6.4.2 Experimental Setup

The lower portion of Figure 5.1 illustrates our training and evaluation pipeline. We first generate synthetic malware code using our framework, perform continued pre-training for domain adaptation, then fine-tune on task-specific formats using general-purpose code to isolate the effect of synthetic malware pre-training. Finally, we evaluate on MalFuncBench to measure malware understanding capabilities.

Synthetic Code Generation. We employ Codestral-25.01 [44] for code generation due to its superior performance on low-level programming tasks. Using MITRE ATT&CK as our knowledge base, we extract 193 unique attack techniques. Each technique generates base implementation with temperature=0.9 to ensure diversity, yielding 10,815 initial codes. These codes are augmented using random selection or full application.

Model Configuration. We use two base models to validate our approach: Llama-3.1-8B-Instruct [45] and Gemma-3-12B-Instruct [46], balancing performance with computational efficiency. Training consists of two stages:

1. **Continued Pre-training**: We adapt the models to the malware domain using LoRA [38] with rank=128, alpha=32. Training runs for 1 epoch using Unsloth framework[1] with learning rate 5e-5.

2. **Task Fine-tuning**: To isolate the impact of domain adaptation, we fine-tune on 10,000 general-purpose C functions decompiled from open-source projects. These samples are extracted from the DIRT dataset [40]. We format the data for each task as follows:

   - Function naming:
     Input: `"What is the function name of this code? {decompiled_function}"`
     Output: `"{function_name}"`
   - Summarization:
     Input: `"Explain the processing of this code. {decompiled_function}"`
     Output: `"{natural_language_summary}"`

   Summaries are generated using Llama-3.1-405B-Instruct. Fine-tuning uses learning rate 1e-4 for 1 epoch.

Evaluation Protocol. Models are evaluated on MalFuncBench using metrics from Section 6.3.3. For inference, we use temperature=0.1, top_p=1.0, and top_k=50 to encourage deterministic outputs suitable for code understanding tasks. We segment function names following [17] conventions and compute Sentence-BERT embeddings using `all-mpnet-base-v2`.

6.4.3 RQ1: Synthetic Data Effectiveness

(1) Baselines

We evaluate against three categories of baselines to comprehensively assess our approach:

- **Commercial LLMs**: GPT-4o evaluated in zero-shot setting for function naming and few-shot setting for summarization (using 2 examples from our fine-tuning data to ensure consistent output format)
- **Without Synthetic Training**: Base models (Llama-3.1-8B-Instruct and Gemma-3-12B-Instruct) fine-tuned only on general-purpose code, measuring the impact of synthetic malware pre-training
- **With Base Code Only**: Our framework using only initial generated code (10,815 samples) without multi-category augmentation, isolating the contribution of augmentation

Our full approach (**With Synthetic Training**) applies the complete framework with full application strategy, generating 54,075 training samples through systematic augmentation.

[1] `https://unsloth.ai`

(2) Results

Table 6.2: Performance comparison on MalFuncBench

Model	Function Naming		Summarization	
	F1	S-BERT	BLEU-4	BERTScore
Commercial LLM				
GPT-4o	0.329	0.506	0.189	0.368
Without Synthetic Training				
Llama-3.1-8B-Instruct	0.217	0.326	0.132	0.346
gemma-3-12b-it	0.224	0.369	0.127	0.323
With Synthetic Training				
Llama-3.1 (base code only)	0.251	0.411	0.113	0.341
Llama-3.1 (aug. code only)	0.284	0.454	0.137	0.351
gemma3 (base code only)	0.218	0.374	0.134	0.357
gemma3 (aug. code only)	0.235	0.408	0.143	0.352

Table 6.2 presents the results on MalFuncBench. Our synthetic training approach demonstrates improvements for both base models, though the magnitude varies across models and tasks.

Function Naming Performance. For Llama-3.1-8B, our augmented approach improves F1 from 21.7% to 28.4%, representing a 31% relative improvement. S-BERT similarity also increases substantially from 32.6% to 45.4%. For Gemma-3-12B, the improvements are more modest, with F1 increasing from 22.4% to 23.5%. These results indicate that the effectiveness of synthetic training varies across model architectures, with Llama-3.1 showing greater responsiveness to domain adaptation.

Summarization Quality. In summarization tasks, both models show improvements with synthetic training. Llama-3.1 demonstrates small gains in BLEU-4 and BERTScore, while Gemma-3-12B shows larger improvements, with BLEU-4 increasing from 12.7% to 14.3%. The base code only variants generally perform worse than augmented versions, confirming the value of systematic augmentation.

Impact of Augmentation. Multi-category augmentation consistently improves performance over base code only training. For function naming, Llama-3.1 shows a 3.3 percentage point improvement with augmentation, while Gemma-3-12B shows a 1.7 point gain. Similar patterns emerge in summarization tasks, demonstrating the consistent value of our augmentation framework.

Performance Relative to Commercial Models. Our best performing model (Llama-3.1 with augmentation) achieves 86% of GPT-4o's F1 score for function naming. While not exceeding commercial model performance, these results demonstrate that domain-specific synthetic training can reduce the performance gap in resource-constrained settings.

(3) Answering RQ1

Our experiments demonstrate that synthetic malware code can improve model performance on malware understanding tasks, though the effectiveness varies by model architecture. The systematic augmentation framework provides consistent benefits over base code generation, with improvements ranging from modest (Gemma-3-12B) to notable (Llama-3.1-8B). While our approach does not surpass state-of-the-art commercial models, it offers a practical method for enhancing smaller models' malware analysis capabilities.

6.4.4 RQ2: Synthetic vs. Real Malware Training

(1) Experimental Settings

Real Malware Dataset. To compare synthetic and real malware training, we collected malware source code from the VX Underground repository[2], focusing on the Win32 category to maintain consistency with our synthetic data's Windows API focus. We treated each source file as a single sample and constructed a dataset of 10,815 samples to match the size of our synthetic dataset.

Training Configurations. We evaluate three training strategies:

- **Real malware only**: Continued pre-training using manually collected malware source code (10,815 samples)
- **Synthetic only**: Our approach using random augmentation strategy (10,815 samples)
- **Real + Synthetic**: Combined training on both datasets (21,630 samples total)

All models use identical hyperparameters and fine-tuning procedures to ensure fair comparison.

(2) Results

Table 6.3 compares synthetic and real malware training. For Llama-3.1-8B, real malware training achieves 27.1% F1 versus 26.2% for synthetic data—a difference of 0.9 percentage points. For Gemma-3-12B, the gap is even smaller: 26.2% versus 25.4% (0.8 percentage points). In semantic similarity, synthetic data performs comparably or slightly better in some cases (42.1% vs. 41.3% S-BERT for Llama-3.1).

[2] `https://github.com/vxunderground/MalwareSourceCode`

Table 6.3: Comparison between synthetic and real malware training on MalFuncBench

Training Data	**Function Naming**		**Summarization**	
	F1	S-BERT	BLEU-4	BERTScore
Llama-3.1-8B-Instruct				
Real malware only	0.271	0.413	0.131	0.362
Synthetic only	0.262	0.421	0.120	0.362
Real + Synthetic	0.254	0.409	0.128	0.359
gemma-3-12b-it				
Real malware only	0.262	0.409	0.120	0.348
Synthetic only	0.254	0.398	0.123	0.349
Real + Synthetic	0.258	0.401	0.118	0.345

Scalability Advantages of Synthetic Data. While real and synthetic data achieve comparable performance, synthetic generation offers significant practical advantages. Our framework can generate arbitrarily large datasets by varying parameters, augmentation strategies, and target techniques. This scalability addresses the fundamental limitation of real malware datasets, which are constrained by availability, legal restrictions, and the effort required for expert annotation. The ability to systematically expand training data enables more comprehensive coverage of malware behaviors and supports training larger models that require extensive datasets.

Quality and Consistency Benefits. Our systematic generation approach produces more uniform code quality and consistent annotation standards compared to real malware collections, which exhibit significant variation in complexity, coding styles, and documentation quality. This consistency facilitates more stable training and evaluation, while the controlled generation process ensures balanced representation across different attack techniques and implementation patterns.

(3) Answering RQ2

Our results demonstrate that synthetic data can effectively simulate real malware code for function understanding tasks. The minimal performance differences between synthetic and real malware training (less than 1 percentage point in F1 score) indicate that our systematic generation framework successfully captures the essential characteristics needed for malware analysis. Moreover, synthetic generation provides crucial scalability advantages, enabling the creation of large-scale datasets that would be impractical to obtain through real malware collection.

6.5 Qualitative Analysis

Table 6.4 compares function naming predictions between Llama-3.1-8B with and without synthetic training, with S-BERT scores providing insight into semantic understanding.

Table 6.4: Examples of predicted function names and their SBERT scores

True	With Synthetic Training		Without Synthetic Training	
	function name	S-BERT	function name	S-BERT
api_hashing_algo	hash_1	0.471	kriz	-0.038
w_rc4_init	RC4_KEY_EXPAND	0.522	aalborg	0.109
GetLocalAdminGroupName	get_user_name	0.391	sid_to_account_name_wide	0.263
CheckVirtualMachinesAndTools	check_virtual_machines	0.891	detect_virtual_machine	0.687

The most notable differences appear in cryptographic function understanding. For the API hashing algorithm, the baseline model without domain adaptation produces `kriz`—a meaningless string with a negative S-BERT score (-0.038). Similarly, for RC4 initialization, it generates `aalborg` with a low score (0.109). These nonsensical outputs indicate the model's inability to recognize cryptographic patterns specific to malware.

With synthetic training, the model demonstrates improved understanding of security-critical operations. It predicts `hash_1` for the hashing algorithm (S-BERT: 0.471) and `RC4_KEY_EXPAND` for RC4 initialization (S-BERT: 0.522). While not exact matches, these predictions capture the functional intent of the operations.

For `GetLocalAdminGroupName`, the baseline model predicts `sid_to_account_name_wide` (S-BERT: 0.263)—a Windows API name that shows some understanding but misses the administrative group context. The synthetic-trained model predicts `get_user_name` (S-BERT: 0.391), which better captures the user-related functionality.

The virtual machine detection example shows both models recognizing the core functionality, but synthetic training produces `check_virtual_machines` (S-BERT: 0.891), which more comprehensively matches `CheckVirtualMachinesAndTools` compared to the baseline's `detect_virtual_machine` (S-BERT: 0.687).

The S-BERT scores align with intuitive semantic similarity, validating it as a meaningful metric for malware function naming evaluation. These examples demonstrate that without domain-specific training, models struggle with malware-specific patterns, often producing irrelevant outputs when encountering unfamiliar cryptographic or security operations. Synthetic training enables recognition of malware-specific idioms, leading to predictions that better preserve semantic intent.

6.6 Limitations

Function-Level Focus. Our evaluation focuses on function-level understanding, while practical malware analysis often requires reasoning about inter-procedural relationships and whole-program behavior. However, function-level analysis serves as a foundation that can be extended to program-level understanding through compositional approaches, making this a natural first step toward comprehensive malware analysis systems.

Evaluation Scope. MalFuncBench contains only 47 functions from 4 malware families, limiting the generalizability of our results. This constraint stems from the labor-intensive process of extracting high-quality ground truth from expert reports. Expanding the bench-

mark could strengthen evaluation reliability and broaden coverage.

6.7 Ethical Considerations

Our research raises important ethical considerations regarding the generation of malware-related code. We emphasize that our synthetic functions are generated individually and lack the system-level integration necessary to function as actual malware. The generated code represents isolated techniques rather than complete attack tools. To balance research transparency with security concerns, we will share our dataset with qualified researchers upon request rather than releasing it publicly without restrictions.

6.8 Conclusion

This research introduced a systematic approach to malware code understanding through synthetic data generation. By leveraging LLMs to transform MITRE ATT&CK descriptions into executable malware code, we address training data scarcity in security research.

Our key contributions include: (1) A framework generating over 54,000 training samples with multi-category augmentation that improves malware function understanding performance; (2) MalFuncBench, the first benchmark for malware function understanding with 47 expert-validated functions; (3) Demonstration that synthetic malware code training achieves comparable performance to real malware training while enhancing model capabilities for malware-specific analysis tasks.

Experimental results show measurable improvements, with synthetic training improving Llama-3.1-8B function naming F1 from 21.7% to 28.4%. Synthetic data performs comparably to real malware training (within 1 percentage point) while offering superior consistency. Qualitative analysis reveals that synthetic training enables recognition of cryptographic patterns that baseline models miss entirely, with S-BERT scores validating semantic understanding.

Our approach enables specialized model training without handling real malware, providing the security research community with a scalable and sustainable method for malware analysis capability development. We anticipate that this work will motivate future developments in AI-based security applications.

Chapter 7

Conclusion

This thesis has addressed fundamental challenges in binary analysis—semantic information loss, lack of task-specific training data, and model interpretability—by proposing a unified framework that integrates interpretable machine learning and natural language processing techniques.

The first contribution, FCGAT, introduced an interpretable malware classification method that models programs as function call graphs and highlights key functions using attention mechanisms. By applying semantic feature extraction to low-level code and emphasizing functions critical for classification, FCGAT not only achieved state-of-the-art accuracy but also provided human-readable explanations that significantly reduced analysts' workload.

The second contribution, RevLlama, tackled the task of function name recovery in stripped binaries. Through rationale-based knowledge distillation from a large language model to a locally deployable smaller model, RevLlama demonstrated that high-level reasoning capabilities can be transferred to computationally efficient models suitable for sensitive environments. This approach achieved superior performance compared to existing LLMs, while also providing interpretable rationales.

To address the domain-specific data scarcity problem, the thesis further introduced a synthetic data generation framework grounded in threat intelligence. By systematically augmenting initial samples across five semantic dimensions, the framework enabled the creation of diverse, realistic malware training data. The resulting benchmark, MalFuncBench, allowed rigorous evaluation of model capabilities in function understanding. Notably, models trained solely on synthetic data achieved performance comparable to those trained on real malware, demonstrating the potential of this approach to mitigate ethical and practical barriers to data access.

Beyond their academic contributions, these studies also provide practical benefits for cybersecurity operations. FCGAT offers analysts function-level explanations that accelerate malware triage and facilitate the creation of reliable detection rules. RevLlama enables the deployment of advanced reasoning models in secure, air-gapped environments where cloud-based services are infeasible, improving the efficiency of reverse engineering under strict security policies. The synthetic data generation framework reduces dependence on scarce and sensitive malware samples, supporting safer training and evaluation pipelines

that can be openly shared across the research community. When combined, these approaches form a coherent workflow: FCGAT highlights suspicious functions, RevLlama recovers informative names for those functions, and the synthetic data framework ensures that both models can be continually trained and evaluated without relying on limited real-world malware. Together, they provide a foundation for practical, scalable, and explainable malware analysis systems that integrate seamlessly into operational security environments.

Collectively, these contributions represent a significant step toward bridging the gap between powerful AI techniques and the practical demands of binary analysis. By prioritizing interpretability, efficiency, and domain alignment, this thesis lays the groundwork for next-generation reverse engineering tools that are not only accurate, but also trustworthy and deployable in real-world cybersecurity operations.

Acknowledgments

I would like to express my deepest gratitude to my supervisor, Prof. Otsuka, for his dedicated guidance and support throughout this research. Since my master's program, he has consistently taught me the joy and depth of research, and has always guided me in the right direction. The excellent research environment he provided allowed me to devote myself fully to my studies, and I am convinced that without his mentorship, I would never have chosen to pursue a doctoral degree.

I am sincerely grateful to Dr. Otsubo for his collaboration as a co-author in writing papers. Despite my repeated requests for reviews under tight deadlines, he always provided prompt and insightful feedback that greatly enhanced the quality of my work.

I also wish to express my appreciation to Prof. Arita, Prof. Inaba, and Prof. Yoshioka for serving as members of my dissertation committee. Their valuable comments and suggestions during the examination significantly improved the completeness of this dissertation.

My gratitude extends to all the researchers and students in the Otsuka Laboratory for their helpful advice and warm encouragement. The atmosphere of active discussion and mutual inspiration has been a tremendous source of support in advancing my research.

I would also like to thank all faculty and staff members at the Institute of Information Security for their support and guidance, not only in research activities but also in many other aspects.

Finally, I dedicate my heartfelt thanks to my family, who have always stood by me and supported my life. My husband has always listened to me with patience and surrounded me with kindness. My parents, grandmother, and parents-in-law have always cared deeply about my well-being.

This journey would not have been possible without the support of all these wonderful people who believed in me and my work.

List of Publications

Peer-Reviewed Journal Articles

- Minami Someya, Yuhei Otsubo, Akira Otsuka, “Graph neural network based function call graph embedding for malware classification,” Journal of Surveillance, Security and Safety. 4, no.2, pp. 47-61, 2023.

Peer-Reviewed Paper in Proceedings of International Canference

- Minami Someya, Yuhei Otsubo and Akira Otsuka, “FCGAT: Interpretable Malware Classification Method using Function Call Graph and Attention Mechanism,” Proceedings of NDSS Workshop on Binary Analysis Research (BAR2023).
- Minami Someya and Akira Otsuka, “RevLlama: Recovering Function Names via Rationale Distillation from Large to Small Language Models,” AAAI Workshops 2025 (AICS).

Book

- 大塚玲, 大坪雄平, 萬谷暢崇, 羽田大樹, 染谷実奈美, ”ゼロからマスター！ Colab Python でバ イナリファイル解析実践ガイド”科学情報出版株式会社，2024【8 章担当】

Bibliography

[1] Edward Raff, Jon Barker, J Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K Nicholas. Malware detection by eating a whole EXE. *AAAI Workshops*, 2018.

[2] Xiao-Wang Wu, Yan Wang, Yong Fang, and Peng Jia. Embedding vector generation based on function call graph for effective malware detection and classification. *Neural Comput. Appl.*, February 2022.

[3] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

[4] L Nataraj, S Karthikeyan, G Jacob, and B S Manjunath. Malware images. In *Proceedings of the 8th International Symposium on Visualization for Cyber Security - VizSec '11*, New York, New York, USA, 2011. ACM Press.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, June 2016.

[6] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for Large-Scale image recognition. September 2014.

[7] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2818–2826, June 2016.

[8] Danish Vasan, Mamoun Alazab, Sobia Wassan, Hamad Naeem, Babak Safaei, and Qin Zheng. IMCFN: Image-based malware classification using fine-tuned convolutional neural network architecture. *Comput. netw.*, Vol. 171, No. 107138, p. 107138, April 2020.

[9] Yanchen Qiao, Bin Zhang, and Weizhe Zhang. Malware classification method based on word vector of bytes and multilayer perception. In *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*. IEEE, June 2020.

[10] Tomas Mikolov, Kai Chen, G Corrado, and J Dean. Efficient estimation of word representations in vector space. *ICLR*, 2013.

[11] Jiaqi Yan, Guanhua Yan, and Dong Jin. Classifying malware represented as control flow graphs using deep graph convolutional neural network. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 52–63, June 2019.

[12] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An End-to-End deep learning architecture for graph classification. *AAAI*, 2018.

[13] Royi Ronen, Marian Radu, Corina Feuerstein, E Yom-Tov, and Mansour Ahmadi. Microsoft malware classification challenge. *ArXiv*, 2018.

[14] Hiromu Yakura, Shinnosuke Shinozaki, Reon Nishimura, Yoshihiro Oyama, and Jun Sakuma. Neural Malware Analysis with Attention Mechanism. *Comput. Secur.*, Vol. 87, , 2019.

[15] Jerome Dinal Herath, Priti Prabhakar Wakodikar, Ping Yang, and Guanhua Yan. CFGExplainer: Explaining Graph Neural Network-Based Malware Classification from Control Flow Graphs. In *52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2022*, pp. 172–184. IEEE, 2022.

[16] Yaniv David, Uri Alon, and Eran Yahav. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proc. ACM Program. Lang.*, Vol. 4, No. OOPSLA, pp. 225:1–225:28, 2020.

[17] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pp. 1631–1645. ACM, 2022.

[18] Jiaqi Xiong, Guoqiang Chen, Kejiang Chen, Han Gao, Shaoyin Cheng, and Weiming Zhang. Hext5: Unified pre-training for stripped binary code information inference. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pp. 774–786. IEEE, 2023.

[19] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pp. 8696–8708. Association for Computational Linguistics, 2021.

[20] Xiuwei Shang, Shaoyin Cheng, Guoqiang Chen, Yanming Zhang, Li Hu, Xiao Yu, Gangyang Li, Weiming Zhang, and Nenghai Yu. How far have we gone in binary code understanding using large language models, 2024.

[21] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Trans. Neural Netw.*, Vol. 20, No. 1, pp. 61–80, January 2009.

[22] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural networks. *IEEE Trans Neural Netw Learn Syst*, Vol. 32, No. 1, pp. 4–24, January 2021.

[23] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. October 2017.

[24] Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. Order matters: Sequence to sequence for sets. November 2015.

[25] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *NIPS*, 2014.

[26] Nitish Srivastava, Geoffrey E Hinton, A Krizhevsky, Ilya Sutskever, and R Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 2014.

[27] I Loshchilov and F Hutter. Decoupled weight decay regularization. *ICLR*, 2017.

[28] Yixuan Ma, Shuang Liu, Jiajun Jiang, Guanhong Chen, and Keqiu Li. A comprehensive study on learning-based PE malware family classification methods. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, pp. 1314–1325, New York, NY, USA, August 2021. Association for Computing Machinery.

[29] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. AVclass: A tool for massive malware labeling. In *Research in Attacks, Intrusions, and Defenses*, Lecture notes in computer science, pp. 230–253. Springer International Publishing, Cham, 2016.

[30] Yara Awad, Mohamed Nassar, and Haidar Safa. Modeling malware as a language. In *2018 IEEE International Conference on Communications (ICC)*. IEEE, May 2018.

[31] Sang Ni, Quan Qian, and Rui Zhang. Malware identification using visualization images and deep learning. *Comput. Secur.*, Vol. 77, pp. 871–885, August 2018.

[32] Shamik Bose, Timothy Barao, and Xiuwen Liu. Explaining AI for malware detection: Analysis of mechanisms of MalConv. In *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, July 2020.

[33] Limin Yang, Arridhana Ciptadi, Ihar Laziuk, Ali Ahmadzadeh, and Gang Wang. BODMAS: An Open Dataset for Learning based Temporal Analysis of PE Malware. In *IEEE Security and Privacy Workshops, SP Workshops 2021*, pp. 78–84. IEEE, 2021.

[34] Cheng-Yu Hsieh, Chun-Liang Li, Chih-kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alex Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes. In *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 8003–8017, Toronto, Canada, July 2023. Association for Computational Linguistics.

[35] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[36] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[37] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pp. 611–626. ACM, 2023.

[38] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.

[39] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, pp. 3980–3990. Association for Computational Linguistics, 2019.

[40] Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Augmenting decompiler output with learned variable names and types. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pp. 4327–4343. USENIX Association, 2022.

[41] Zeyuan Allen-Zhu and Yuanzhi Li. Physics of language models: part 3.1, knowledge storage and extraction. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[42] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*, pp. 311–318. ACL, 2002.

[43] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with bert. *ArXiv*, Vol. abs/1904.09675, , 2019.

[44] MistralAI. Codestral. https://mistral.ai/news/codestral/, 2024. Accessed: 2025-06-20.

[45] Meta AI. Introducing llama 3.1: Our most capable models to date. https://ai.meta.com/blog/meta-llama-3-1/, 2024. Accessed: 2025-06-23.

[46] Gemma Team. Gemma 3 technical report, 2025.

Appendix. Malware Analysis Reports Used in MalFuncBench

This appendix lists the malware analysis reports from which we extracted expert-annotated functions for MalFuncBench. All reports were accessed June 2025.

- Alexandre Borges. *Malware Analysis Series (MAS) Part 3: Amadey v1.22.* ExploitReversing, May 2022. Available at: `https://exploitreversing.com/wp-content/uploads/2022/05/mas_3.pdf`
- Alexandre Borges. *Malware Analysis Series (MAS) Part 5: Amadey Bot v3.21.* ExploitReversing, September 2022. Available at: `https://exploitreversing.wordpress.com/wp-content/uploads/2022/09/mas_5.pdf`
- Alexandre Borges. *Malware Analysis Series (MAS) Part 6: Amadey Bot v3.23.* ExploitReversing, November 2022. Available at: `https://exploitreversing.com/wp-content/uploads/2022/11/mas_6-1.pdf`
- Hossein Jazi. *A Deep Dive into Saint Bot Downloader.* Malwarebytes ThreatDown Blog, 2022. Available at: `https://www.threatdown.com/blog/a-deep-dive-into-saint-bot-downloader/`