Windows Malware Classification using Vector Databases and Hierarchical Navigable Small World Graph Algorithm

by

Nilesh Jakamputi

A Project Report Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
Master of Science
Supervised by

Dr.Tae Oh

School of Information

B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

January 2024

The project report “Windows Malware Classification Using Vector Databases and Hierarchical Navigable Small World Graph Algorithm” by Nilesh Jakamputi has been examined and approved by the following Examination Committee:

Dr.Tae Oh
Professor
Project Committee Chair

Dean Ganskop
Lecturer

Matthew Wright
Department Chair

Abstract

Windows Malware Classification using Vector Databases and Hierarchical Navigable Small World Graph Algorithm

Nilesh Jakamputi

Supervising Professor: Dr.Tae Oh

Malware classification is a significant problem in the ever-changing realm of cybersecurity, this project proposes to tackle the issue with an end-to-end pipeline for malware classification. By leveraging the use of vector representations of text from Large Language Models(LLMs) on API call sequences extracted from Windows Portable Executable (PE) files, and applying the Hierarchical Navigable Small World Graph Algorithm (HNSW) on the embeddings, one can classify Malware API call sequences with accuracy.

API calls and their arguments are extracted from PE files using an emulator known as Speakeasy, which is a Windows kernel and user mode emulator. It is widely used in the industry, and one can extract PE runtime logs using the emulator. The resulting calls are stored as JSON logs, to facilitate our use case, the logs are parsed, normalized, and transformed into a CSV file with Label and API Call Sequence columns, so we can structure the problem as a sequence classification task.

Natural Language sequences are significantly different from API Call sequences, one must adapt their approach to tackle this issue as traditional approaches will not work. This project aims to solve this issue by garnering embeddings from Large Language Models and using vector databases and vector indexing techniques to effectively query the embeddings for the most similar label matches for each chunked API call and attempt to classify a whole sequence based on retrieved matches.

The goal is to get a comprehensive representation of the runtime behavior of the malware via the API call sequence and classify the sequence by using the HNSW algorithm on each chunk of the sequence.
The project's deliverables include implementing this pipeline, conducting extensive experimentation to tune hyperparameters such as chunk size, and embedding representations, and thoroughly evaluating the system's performance in real-world malware detection scenarios.

Contents

Abstract **1**

1 Introduction **5**

2 Background **7**
2.1 Windows Operating System Internals for API Calls 7
2.2 Windows Defenses 9
2.3 Malware Evasion and Obfuscation Techniques 11
2.4 SpeakEasy Emulation Framework 13

3 Objectives **15**

4 Exploratory Data Analysis **17**
4.1 Dataset Overview 17
4.2 Malware Categories and Distribution 17
4.3 API Call Analysis 18

5 Related Work **20**

6 Methodology **24**
6.1 API Call Logs 24
6.2 Data Preprocessing 24
6.2.1 Normalization 24
6.2.2 Conversion to CSV 25
6.3 Embedding and Vector Database Storage 25
6.4 Search with Hierarchical Navigable Small World 25
6.5 Classification 26
6.6 Evaluation 27

7 Results **27**
7.1 Dynamic Analysis and Classification Performance 27
7.2 Analysis and Discussion 30
7.3 Future Work and Alternatives Tried 31

8 Conclusion **33**
8.1 Limitations 34

References **34**

List of Figures

1 Sample Data of a post-processed Backdoor 6
2 Frequency Chart of Samples 18
3 Average Number of API Calls per Label 19
4 Overlaps of API calls between Malware and Clean Samples 20
5 Confusion Matrix 28
6 Learning Rate 32
7 Training Time 32
8 Training Loss 32
9 Train Samples per second 32

1 Introduction

Malware, short for malicious software, refers to computer programs that exploit and harm computer systems. Bayer et al. (2006) emphasize that the rapid evolution of malware has made it increasingly difficult to detect and classify these threats using traditional methods. The prevalence of malware has grown significantly over the past decade, with malware developers employing increasingly sophisticated tools and techniques to evade detection (Ye et al., 2017).

Malware authors are often financially motivated, and their end goals determine the techniques used to construct the malware. The most prevalent types of malware include ransomware, worms, trojan horses, and spyware. Ye et al. (2017) highlight that the increasing number and complexity of malware pose a significant threat to computer systems and users worldwide, necessitating the development of more effective detection and classification methods.

According to an AV-TEST report from 2019 to 2024, approximately 114 million new malware samples with varying exploitation, obfuscation, and evasion techniques are developed every year ("AV-TEST ATLAS 2008 - 2024," 2024). The report also reveals that over 90% of malware targets Windows systems, as enterprises predominantly use the Microsoft ecosystem. This emphasizes the need for robust malware detection and classification techniques tailored to Windows-based malware.

Malware analysis can be categorized into two approaches: static analysis and dynamic analysis. Static analysis involves examining the program's code and structure without executing it, using tools such as IDA Pro, Ghidra, and dnSpy. However, Damodaran et al. (2017) point out that static analysis methods struggle to cope with sophisticated obfuscation techniques employed by malware authors when applying machine learning techniques to malware detection.

Obfuscation poses a specific challenge for machine learning-based malware detection. Damodaran et al. (2017) explain that obfuscation techniques, such as code encryption, polymorphism, and metamorphism, can significantly alter the static features of malware samples without changing their underlying functionality. This leads to a discrepancy between the training data and the obfuscated malware samples encountered in real-world scenarios, making it difficult for machine learning models to generalize and accurately detect obfuscated malware.

To address this challenge, Igor Santos (2013) propose a dynamic analysis approach that captures the behavior of malware at runtime by executing it in a controlled environment and documenting its interactions with the operating system through API calls. By focusing on the API calls made by the malware, this approach can effectively

identify and classify malicious behavior, even in the presence of obfuscation.

Traditional malware detection techniques rely on signatures and heuristics used by endpoint detection and response software. Aslan and Samet (2020) explain that these methods compare malware to known malicious patterns or artifacts, such as file paths, registry keys, or IP addresses. However, they also highlight that these approaches often struggle to generalize to unknown attacks or variants and can be easily circumvented by obfuscation techniques.

In this report, we detail an end-to-end pipeline that leverages dynamic analysis, natural language processing, embedding models, vector databases, and cosine similarity to improve malware identification and classification. Our primary objectives are to overcome the limitations of traditional signature-based approaches by focusing on the runtime behavior of malware through API call analysis, enhance the generalization capabilities of malware detection models to accurately identify and classify unknown malware variants and obfuscated samples, and ultimately contribute to the field of malware analysis by providing a more robust and effective approach to detecting and classifying malware.

Figure 1. Sample Data of a post-processed Backdoor

The pipeline, inspired by the work of Trizna et al. (2023), begins by running the malicious Windows PE file in the Speakeasy emulator, which generates a JSON log containing the API calls and parameters used by the malware. We then apply feature engineering techniques to normalize the logs and reduce the vocabulary, ensuring more accurate representations across the dataset. The normalized logs are transformed into a CSV format, with columns for the label and the API call sequence.

We use frameworks such as LangChain Langchain-Ai (n.d.) to chunk the API call sequence using regular expressions and employ a vector embedding model to generate

embeddings for each chunk. These embeddings, along with their corresponding labels, are stored and indexed in a vector database such as Qdrant Open (n.d.). To classify unknown malware samples, we transform the input logs using the same normalization and chunking process. We then use the Hierarchical Navigable Small World Graph algorithm, as described by Malkov and Yashunin (2018), to search the embedding space and retrieve the most relevant labels for each chunk. The classification is based on the retrieved label frequencies, potentially utilizing the Cohere rerank algorithm for improved accuracy.

By developing this end-to-end pipeline, we aim to contribute to the field of malware analysis by providing a more robust and effective approach to detecting and classifying malware, particularly in the face of increasingly sophisticated obfuscation techniques employed by malware authors. Our focus on dynamic analysis using emulation, NLP techniques, embedding models, and vector databases represents a novel approach to tackling the challenges posed by modern malware, and we believe that our work will have significant implications for the development of more effective malware detection and classification systems.

2 Background

2.1 Windows Operating System Internals for API Calls

The Windows Operating System, developed by Microsoft, is a closed-source, graphical operating system that abstracts many of the low-level details of hardware interaction. This abstraction allows users and developers to interact with the system in a more straightforward manner. For developers, access to the kernel is only possible through a specific system of API calls, which is crucial to understand when analyzing the behavior of Windows applications, including malware.

When a Windows executable (PE file) is launched, the Windows loader is responsible for loading the program into memory. The loader reads the PE headers, maps the necessary sections into memory, and resolves the import table, which contains references to the required DLLs and their exported functions. The loader then maps the DLLs into the process's memory space and updates the import address table (IAT) with the actual addresses of the imported functions. Finally, the loader transfers control to the program's entry point, typically the WinMain or DllMain function.

In user mode (ring 3), the primary way to request resources and execute instructions is by calling functions exported by dynamic link libraries (DLLs). These DLLs, such as Kernel32.dll, User32.dll, and Gdi32.dll, are part of the Windows subsystem and provide an interface between user-mode applications and the kernel. When a binary is statically linked, the required functions are included directly in the executable and called at runtime. In the case of dynamic linking, a copy of the DLL is mapped to the

process, and its location is stored in the Process Environment Block (PEB) structure, specifically in the_PEB_LDR_DATA structure, which contains information about the loaded modules.

DLL functions are the most common method for user-mode applications to request resources from the kernel, as access to various system resources and services is implemented through these exported functions. This means that all Windows applications, including malware, typically use these API calls to perform any meaningful operations.

However, some malware authors may employ techniques to bypass the standard API calling mechanism and directly invoke system calls (syscalls) to evade detection or analysis. One such technique is known as ”direct syscalls” or ”syscall stubbing,” which involves manually crafting the necessary data structures and invoking the syscall instruction directly, bypassing the need for API calls through DLLs. This technique can make it more difficult for security tools and researchers to monitor and analyze the malware's behavior, as it circumvents the traditional API hooking and interception methods.

To illustrate the standard API calling process, consider the example of calling TerminateProcess in a C program to terminate a process. The application calls the TerminateProcess function exported by Kernel32.dll, which in turn calls the NtTerminateProcess function from ntdll.dll. NtTerminateProcess is a wrapper for the underlying system call, which sets up the necessary arguments and invokes the syscall instruction to switch context to the kernel with elevated permissions and terminate the specified process.

The typical flow for a Win32 API call is as follows: Application layer API (e.g., Kernel32.dll) -> Native API (e.g., ntdll.dll) -> System call -> Kernel mode -> Kernel API executes requested operation -> Returns result to user mode

This layered architecture of API calls allows for a stable and consistent interface for user-mode applications while providing flexibility for the operating system to change the underlying implementation of system calls and kernel-mode services across different versions of Windows.

While the native API functions in ntdll.dll are not officially documented, developers can find documentation for the Windows API functions exported by the higher-level DLLs (e.g., Kernel32.dll, User32.dll) in the official Microsoft documentation (Microsoft, 2024). Additional information on undocumented Windows internals can be found in the Vergilius project (Svitlana Storchak, 2023) and Geoff Chappell's website (GeoffChappel, 2023).

To ensure compatibility across Windows versions, even when underlying system call implementations change, developers are encouraged to use the documented Windows API functions provided in the official Microsoft documentation. This advice applies to both legitimate applications and malware. Malware authors can either directly invoke system calls after enumerating the necessary information, use the documented Windows API functions, or employ techniques like indirect syscalls to evade detection.

Understanding the Windows API calling mechanism, the internals of the Windows

operating system, and the techniques used by malware authors is essential for analyzing the behavior of Windows applications, particularly malware. By gaining a deep understanding of how user-mode applications interact with the kernel through API calls and direct syscalls, researchers and security professionals can develop more effective tools and techniques for analyzing and detecting malicious behavior. This knowledge also forms the foundation for creating emulation frameworks, such as Speakeasy, which aim to provide a high-resolution view of the Windows operating system for dynamic malware analysis.

2.2 Windows Defenses

Windows employs a multi-layered approach to defend against malware, exploits, and unauthorized access attempts. This defense strategy includes a combination of exploit mitigations, antivirus solutions, and advanced security features that work together to provide comprehensive protection.

Exploit Mitigations: Windows implements several exploit mitigations to prevent or hinder the successful execution of exploits targeting specific vulnerabilities or bug classes. These mitigations operate at different levels of the system, from hardware-enforced features to compiler-level protections.

Data Execution Prevention (DEP) is a hardware-enforced feature that marks memory pages as non-executable, preventing the execution of code from data regions. This mitigation effectively combats buffer overflow exploits that attempt to execute malicious code injected into data areas.

Address Space Layout Randomization (ASLR) randomizes the memory layout of processes, making it difficult for attackers to predict the location of code and data. By randomly arranging the base addresses of the executable, stack, heap, and libraries, ASLR thwarts exploits that rely on hardcoded memory addresses.

Control Flow Guard (CFG) and Extended Flow Guard (XFG) ensure that indirect function calls align with the program's intended control flow. These mitigations protect against control flow hijacking attempts by verifying the target addresses of indirect calls against a pre-computed list of valid targets.

Arbitrary Code Guard (ACG) prevents the introduction of dynamically generated code by enforcing code integrity checks. It ensures that only code signed by Microsoft or the application developer can be executed, thus reducing the attack surface for code injection exploits.

Export Address Filtering (EAF) filters access to exported functions from specific modules, preventing techniques like DLL hijacking. EAF checks if the calling module is authorized to access the exported function, based on a predefined allow list.

Stack protection mechanisms, such as Stack Canaries and Shadow Stacks, detect and prevent buffer overflow attempts on the stack. Stack Canaries are random values placed between the local variables and the return address, which are checked for

integrity before function returns. Shadow Stacks provide an additional layer of protection by maintaining a separate, protected stack for storing return addresses.

Structured Exception Handling Protection (SEHOP) and SafeSEH protect against the abuse of exception handling mechanisms for exploitation. SEHOP validates the integrity of the exception handler chain, while SafeSEH ensures that only registered exception handlers can be called during exception handling.

Supervisor Mode Execution and Access Prevention (SMEP/SMAP) are CPU-level features that prevent the execution of user-mode code with kernel privileges (SMEP) and unauthorized access to user-mode pages from kernel mode (SMAP). These mitigations hinder privilege escalation attempts and the abuse of kernel vulnerabilities.

Windows Defender Antivirus: Windows Defender is the built-in antivirus solution in Windows that employs various techniques to detect and prevent malware infections. It operates at different levels of the system, using a combination of signature-based detection, heuristic analysis, and behavioral monitoring.

The Antimalware Scan Interface (AMSI) is a critical component of Windows Defender that hooks into multiple layers of the OS, including scripting engines and the Windows API. When a script or code snippet is about to be executed, AMSI intercepts the execution and scans the code for potentially malicious behavior. It uses a combination of signature-based detection and heuristic analysis to determine if the code is malicious.

AMSI also provides an interface for third-party antivirus solutions to integrate their scanning engines, allowing for a more comprehensive and collaborative approach to malware detection.

Windows Defender's file scanning capabilities are handled by components like MpEngine.dll and MpSvc.dll, which are part of the Microsoft Defender Scan Engine. These components scan files on disk and in memory for known malware signatures and suspicious patterns. They utilize static analysis techniques, such as hash-based lookups and pattern matching, and dynamic analysis techniques, like emulation and behavioral analysis.

When a file is accessed or executed, Windows Defender scans it against a database of known malware signatures. If a match is found, the file is flagged as malicious, and appropriate actions are taken, such as blocking the execution or quarantining the file.

In addition to signature-based detection, Windows Defender employs heuristic analysis and machine learning models to identify potentially malicious files and activities. Heuristic analysis involves examining the characteristics and behavior of a file to determine if it exhibits suspicious or malicious traits, even if it doesn't match any known malware signatures. This approach allows Windows Defender to detect new and previously unseen malware variants.

Behavioral monitoring is another technique used by Windows Defender to detect malicious activities. It involves monitoring the actions of running processes and detecting suspicious patterns, such as unauthorized modifications to system files, registry changes, or network communication with known malicious domains.

Windows Defender also leverages cloud-based protection through the Microsoft Defender Antivirus cloud service. This service collects telemetry data from Windows systems worldwide, including information about detected threats and suspicious files. The collected data is analyzed using machine learning algorithms and other advanced analytics techniques to identify emerging threats and provide real-time protection updates to Windows Defender clients.

When a suspicious file is detected, Windows Defender can take various actions based on the configured policies and the severity of the threat. These actions include:

Blocking the execution of the file to prevent it from causing harm. Quarantining the file by moving it to a secure location and preventing it from being accessed or executed. Removing the file from the system to eliminate the threat. Alerting the user and providing recommendations for further action. Windows Defender also integrates with other Windows security features, such as the Microsoft Defender Firewall and the Microsoft Defender for Endpoint (formerly Windows Defender Advanced Threat Protection). The firewall component helps to block unauthorized network traffic and prevent malware from communicating with command-and-control servers. Microsoft Defender for Endpoint provides advanced threat detection, investigation, and response capabilities, leveraging behavioral analytics and machine learning to identify and respond to sophisticated threats.

To ensure the effectiveness of Windows Defender, Microsoft regularly updates its malware definitions and detection algorithms based on the latest threat intelligence and research. These updates are automatically downloaded and applied to Windows systems, providing continuous protection against evolving threats.

Windows Defender combines signature-based detection, heuristic analysis, behavioral monitoring, and cloud-based protection to provide a comprehensive defense against malware. Its integration with other Windows security features and the ability to leverage the collective intelligence of the Microsoft Defender Antivirus cloud service makes it a robust and adaptable antivirus solution.

It's important to note that while Windows Defender provides a strong baseline of protection, it is not foolproof. Malware authors continually develop new techniques to evade detection, and no single antivirus solution can guarantee 100% protection. Yosifovich (2017)

2.3 Malware Evasion and Obfuscation Techniques

Malware authors employ a wide range of techniques to evade detection and hinder analysis, making it challenging for security software and malware analysts to decipher the malware's true intentions and functionality. These techniques encompass obfuscation, anti-debugging, and anti-analysis measures, which are designed to conceal the malware's code and behavior, both on disk and during runtime.

When analyzing malware on disk, researchers often rely on disassemblers and decompilers to reconstruct the malware's code from raw bytes into human-readable assembly code or a high-level language representation. Tools such as IDA Pro, Ghidra, Binary Ninja, dotPeek, and dnSpy are commonly used for this purpose, depending on the programming language and compilation target (e.g., C/C++, C#, Python, or Java).

Legitimate programs are typically developed following software engineering best practices, emphasizing code readability, maintainability, and efficiency. In contrast, malware often employs convoluted code paths, obfuscated strings, and seemingly random API calls to conceal its true functionality and evade detection. The malicious code is usually nested within layers of obfuscation and misdirection, often called a code cave, making it difficult to identify and understand the malware's behavior.

According to a study by Galloro et al. (2022), malware evasion techniques can be broadly categorized into two main groups: anti-debugging and anti-sandbox/virtualization. Anti-debugging techniques aim to detect and hinder the use of debuggers, which are essential tools for malware analysts to step through the code and understand its behavior. Anti-sandbox and anti-virtualization techniques, on the other hand, focus on detecting and evading analysis environments, such as virtual machines or sandboxes, which are commonly used for safe and isolated malware execution.

One specific anti-debugging technique is the use of time-based checks. Malware can measure the time taken to execute specific parts of its code and compare it against expected values. If the execution time is significantly longer than expected, the malware assumes that it is being run under a debugger, alters its behavior, and terminates itself to avoid analysis. This technique exploits the fact that debugging typically slows down the execution of the code due to the additional overhead of the debugger.

Another anti-debugging technique involves the use of anti-disassembly tricks. Malware authors can employ techniques such as inserting junk code or overlapping instructions to confuse disassemblers and make the code harder to understand. For example, the malware may include conditional jumps that are never taken, leading to the disassembler incorrectly interpreting the following bytes as valid instructions. This can result in a disassembled code that is misleading or difficult to follow, hindering the analysis process.

Anti-sandbox and anti-virtualization techniques used by malware aim to detect the presence of analysis environments and alter the malware's behavior accordingly. One common approach is to check for the presence of specific artifacts or indicators associated with virtual machines or sandboxes. These indicators can include certain registry keys, file paths, or system services that are typically present in analysis environments but not on real machines. If the malware detects such indicators, it may choose to terminate itself or exhibit benign behavior to avoid revealing its malicious intent.

Malware authors also employ various obfuscation techniques to make their code more difficult to understand and analyze. One common obfuscation method is packing, which involves compressing and encrypting the malware's code and data. The packed malware includes a small unpacking routine that decompresses and decrypts

the original code at runtime, making it harder to analyze statically. Packers can also employ anti-debugging and anti-disassembly techniques to further complicate the analysis process.

Another advanced obfuscation technique is virtualization-based obfuscation, where the malware includes a custom virtual machine with its own instruction set. The malicious code is translated into this custom instruction set and executed by the virtual machine at runtime. This approach significantly increases the complexity of the malware and makes it extremely challenging to analyze using traditional reverse engineering techniques.

Malware may also leverage less common features of the operating system to hide its presence. For example, alternate data streams (ADS) in the Windows NTFS file system can be used to store malicious components or data without being easily detectable. Malware can also abuse legitimate system mechanisms, such as thread-local storage (TLS) callbacks or fiber-local storage (FLS), to execute malicious code during the early stages of process initialization, before many security tools are fully loaded.

Obfuscation techniques can be applied at various levels, from the individual instructions and data structures to the overall flow and structure of the program. High-level languages like C# and JavaScript are particularly susceptible to source code obfuscation, where the original code is transformed into a convoluted and hard-to-read form while preserving its functionality. This can involve renaming variables and functions to meaningless names, inserting dummy code, or using complex control flow structures to obscure the program's logic.

The combination of evasion techniques, obfuscation methods, and the constant evolution of malware makes it increasingly challenging to detect and analyze malicious code. Malware analysts and security researchers must continuously adapt their techniques and tools to keep pace with the ever-changing landscape of malware evasion and obfuscation. This often involves leveraging advanced analysis methods, such as dynamic analysis, machine learning, and data mining, to uncover the true behavior and intent of evasive malware samples.

2.4 SpeakEasy Emulation Framework

SpeakEasy Mandiant (2022) is an advanced emulation framework developed by Mandiant that aims to provide a high-fidelity representation of the Windows operating system for dynamic malware analysis. The primary goal of Speakeasy is to emulate Windows user mode and kernel mode binaries, targeting malware designed for x86 and amd64 platforms. By offering comprehensive API support and emulating various Windows components, Speakeasy enables researchers to observe and analyze malware behavior in a controlled and isolated environment.

At the heart of Speakeasy lies the Unicorn engine, a powerful emulation tool based on the QEMU emulator. Unicorn is responsible for emulating CPU instructions for both amd64 and x86 architectures, providing a foundation for Speakeasy's emulation

capabilities. When a malware sample, in the form of a Portable Executable (PE) file or shellcode, is loaded into Speakeasy, it is mapped into the emulated memory space at a specified location or the default address of 0x40000.

To create a realistic emulation environment, Speakeasy constructs the necessary Windows data structures and components that malware typically interacts with. These include the Process Environment Block (PEB), Thread Environment Block (TEB), I/O Controls (IOCTLs), system files, DLLs, registry hives, file systems (NTFS/FAT32), drive paths, and even fake DNS responses. By meticulously emulating these structures and subsystems, Speakeasy allows the malware to execute as if it were running on a real Windows system, enabling researchers to observe its behavior and interactions with the emulated environment.

Speakeasy's Windows API emulation is handled through Python API handlers. These handlers are responsible for providing the expected outputs and return values to the emulated functions, ensuring that the malware's code paths are properly executed. In cases where a specific function is not implemented in Speakeasy, the framework gracefully handles the situation by reporting the missing function, skipping it, and proceeding to the next function's entry point. This approach ensures that the emulation can continue even if certain API calls are not fully supported, allowing for a more comprehensive analysis of the malware's behavior.

One of the key advantages of using emulation frameworks like Speakeasy for malware analysis is their ability to circumvent many common evasion and obfuscation techniques employed by malware authors. Malware often incorporates anti-debugging, anti-sandbox, and anti-virtualization mechanisms to hinder analysis efforts. These techniques are designed to detect and evade traditional analysis environments, such as debuggers and sandboxes, which are known to be widely used by researchers.

However, emulation frameworks present a different environment that may not trigger the same evasion mechanisms. Malware authors typically focus their evasion efforts on popular analysis tools and environments, such as Cuckoo Sandbox, as they are aware that researchers commonly use these tools to analyze malware and collect behavioral logs. In contrast, emulation frameworks like Speakeasy provide a unique environment that malware may not be specifically designed to detect or evade.

Moreover, emulation offers several practical advantages over sandbox-based analysis. Sandbox analysis often requires setting up a clean environment, taking snapshots, and performing resets for each analyzed sample to prevent contamination and ensure consistent results. There is also a risk of malware escaping the sandbox and compromising the host system, potentially leading to data leaks or system instability.

Emulation frameworks, on the other hand, provide a highly controlled and isolated environment for malware analysis. While an initial setup is required to configure the emulation framework, there is no need to perform resets or restore snapshots between analyzing different malware samples. This is because the malware code is not executed but emulated, eliminating the risk of contamination or unintended consequences.

This makes emulation a more cost-effective and efficient approach for analyzing a

large number of malware samples. Researchers can quickly iterate through multiple samples without the overhead of resetting the analysis environment each time. Additionally, emulation frameworks often provide detailed logging and tracing capabilities, allowing researchers to capture and analyze the malware's behavior at a granular level.

However, it is important to acknowledge that emulation is not a perfect solution for all malware analysis scenarios. As emulation frameworks gain popularity and adoption, malware authors may start developing techniques specifically designed to detect and evade emulation environments. Moreover, emulation may not perfectly replicate every aspect of a real system, and some malware may exhibit different behavior or fail to execute properly in an emulated environment compared to a physical machine.

Despite these limitations, emulation frameworks like Speakeasy provide a powerful tool for malware analysis and research. By offering a high-fidelity emulation of the Windows operating system, Speakeasy enables researchers to observe and analyze malware behavior in a controlled and isolated environment. The framework's ability to emulate various Windows components and APIs allows for a more comprehensive understanding of the malware's functionality, persistence mechanisms, and potential impact on infected systems.

As malware continues to evolve and employ increasingly sophisticated evasion and obfuscation techniques, the cybersecurity community must adapt and develop advanced analysis tools and techniques. Emulation frameworks like Speakeasy, when used in conjunction with other analysis methods such as static analysis, dynamic analysis, and machine learning, can provide valuable insights into the inner workings of malware and aid in the development of effective defense mechanisms.

Speakeasy's modular architecture and extensibility also make it a valuable platform for future research and development in the field of malware analysis. As new malware techniques emerge, researchers can extend and adapt Speakeasy to incorporate additional emulation capabilities, API support, and analysis features. This flexibility ensures that the framework can keep pace with the ever-evolving landscape of malware threats.

In conclusion, the Speakeasy emulation framework represents a significant advancement in the field of malware analysis. By providing a high-fidelity emulation of the Windows operating system and enabling researchers to observe malware behavior in a controlled and isolated environment, Speakeasy offers a powerful tool for understanding and combating malware threats. As the cybersecurity community continues to face the challenges posed by sophisticated malware, emulation frameworks like Speakeasy will play an increasingly crucial role in the fight against malicious software.

3 Objectives

The objectives are as follows:

1. To develop a robust and efficient pipeline for processing and transforming raw

malware analysis logs into a structured format suitable for machine learning tasks, specifically focusing on sequential API call data and their corresponding labels.

2. To investigate and implement effective techniques for chunking and embedding API call sequences, leveraging state-of-the-art tools and models such as regular expressions, vector embedding models, and vector databases, to capture and represent the behavioral characteristics of malware samples.

3. To design and create a comprehensive vector database that stores the embedded chunks along with their associated metadata, enabling efficient retrieval and comparison of malware behavioral profiles.

4. To explore and evaluate advanced similarity search algorithms, such as the Hierarchical Navigable Small World Graph algorithm, for querying the vector database and retrieving relevant labels for test input chunks, facilitating accurate and efficient malware classification.

5. To develop a robust classification mechanism that utilizes the retrieved labels from the vector database to accurately classify test inputs, demonstrating the effectiveness of the proposed approach in identifying and categorizing malware based on their behavioral characteristics.

6. To conduct thorough testing and evaluation of the developed pipeline, assessing its performance using relevant metrics such as accuracy, precision, recall, and F1-score, to validate its efficacy in real-world malware detection and classification scenarios.

7. To contribute to the field of malware analysis by presenting a novel and innovative approach that combines dynamic analysis, vector embeddings, and advanced similarity search techniques, pushing the boundaries of current malware detection and classification methods.

8. To provide a solid foundation for future research and development in the area of behavioral-based malware analysis, encouraging further exploration and refinement of the proposed techniques and methodologies.

By achieving these objectives, this capstone project aims to advance the state-of-the-art in malware detection and classification, offering a powerful and flexible framework that can adapt to the ever-evolving landscape of malware threats and contribute to the development of more effective and efficient security solutions.

4 Exploratory Data Analysis

4.1 Dataset Overview

The dataset used in this study was generated using Speakeasy v1.5.9 (Mandiant, 2022), a powerful Windows kernel emulator capable of capturing detailed behavioral reports from both legitimate and malicious samples. The dataset comprises approximately 80,000 samples, with malicious samples belonging to seven distinct malware types. This diverse composition makes the dataset suitable for both malware detection and classification tasks.

One notable aspect of this dataset is the temporal separation between the training and test sets. The training set consists of samples collected in January 2022, while the test set was collected in April 2022. This separation allows for the examination of concept drift in malware behavior, as malware techniques and patterns may evolve. By evaluating the model's performance on a temporally distinct test set, we can assess its ability to generalize and adapt to changing malware landscapes.

4.2 Malware Categories and Distribution

The dataset encompasses seven distinct malware types: backdoor, ransomware, trojan, dropper, coinminer, keylogger, and rat. Each malware type exhibits unique characteristics and behaviors, reflecting their specific purposes and attack mechanisms. Additionally, the dataset includes a 'clean' category, which represents legitimate Windows executable files, including the 'windows' samples.

Figure 2 presents the distribution of samples across different malware types and the clean category. The chart reveals an imbalance in the dataset, with the clean samples (including the 'windows' samples) significantly outnumbering the malicious ones. Specifically, the dataset contains 24,668 clean samples (24,433 'clean' + 235 'windows'), while the malware samples range from 1,697 for the 'rat' category to 11,061 for the 'backdoor' category.

This imbalance is a common challenge in malware analysis datasets, as legitimate samples are often more readily available than malicious ones. It highlights the need for careful consideration during model training and evaluation to ensure that the models can accurately identify and classify malware samples, even when their prevalence is relatively low compared to clean samples.

Backdoor and ransomware have the highest representation among the malware types, with 11,061 and 9,627 samples, respectively. This distribution aligns with the prevalence and significance of these malware types in real-world scenarios. Backdoors provide unauthorized remote access to infected systems, while ransomware encrypts files and demands ransom payments.

Trojans and droppers also have a significant presence in the dataset, with 8,733 and 8,243 samples, respectively. These malware types are known for their ability to deceive

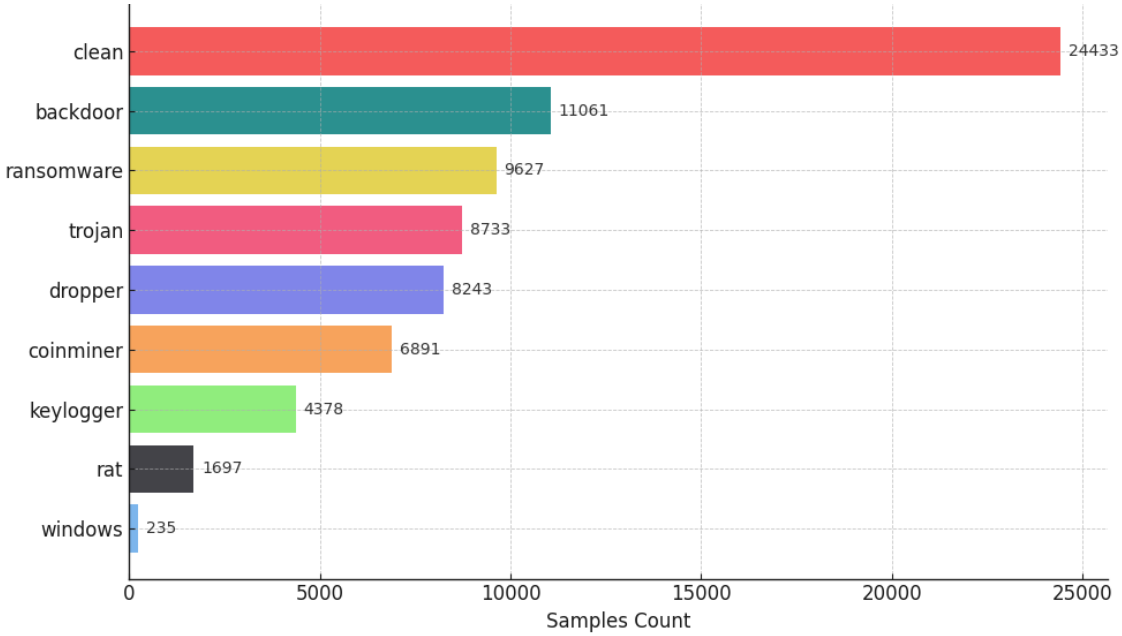

Figure 2. Frequency Chart of Samples

users and deliver additional malware payloads.

The presence of coinminers (6,891 samples), keyloggers (4,378 samples), and remote access trojans (RATs) (1,697 samples) adds further diversity to the malware landscape. Coinminers focus on unauthorized cryptocurrency mining, keyloggers capture sensitive user inputs, and RATs provide extensive remote control capabilities to attackers.

Understanding the distribution of malware types is crucial for developing effective detection and classification models. The imbalanced nature of the dataset requires careful consideration during model training and evaluation to ensure that the models can accurately identify and classify malware samples, even when their prevalence is relatively low compared to clean samples.

4.3 API Call Analysis

The Speakeasy emulator captures detailed logs of API calls made by the executed samples, providing valuable insights into their behavioral characteristics. Figure 3 presents the average number of API calls per malware type and the clean category.

The chart reveals a significant difference in the average number of API calls between malicious and clean samples. Malware samples generally exhibit a higher number of API calls compared to clean samples. On average, backdoor samples make 418.117892 API calls, while clean samples make 168.142389 calls, and Windows samples make

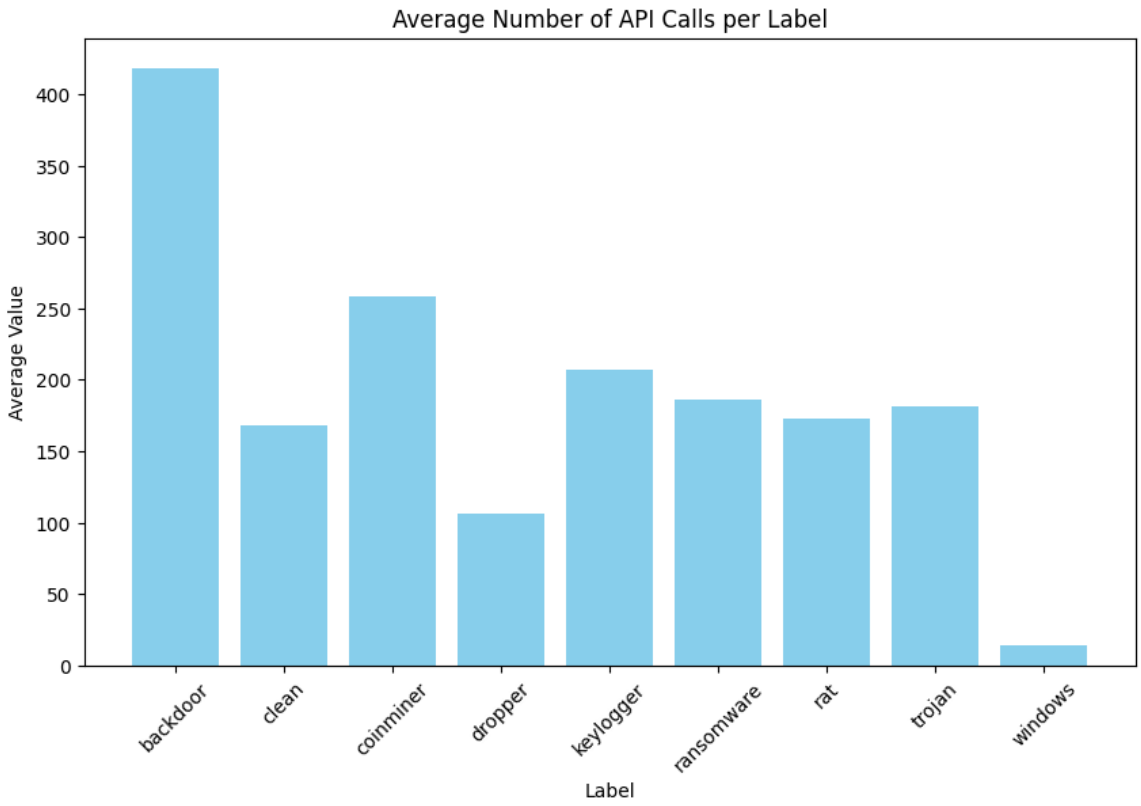

Figure 3. Average Number of API Calls per Label

14.055319 API calls.

This observation aligns with the understanding that malware often performs more extensive system interactions and manipulations to carry out its malicious activities. The higher number of API calls in malware samples indicates their complex behavior and the need to interact with various system components to achieve their goals.

Among the malware types, backdoors stand out with the highest average number of API calls (418.117892). This higher complexity can be attributed to the wide range of capabilities typically associated with backdoors, such as establishing persistence, communicating with command-and-control servers, and performing data exfiltration.

Ransomware samples also exhibit a relatively high average number of API calls (186.488002) compared to the clean samples, which can be attributed to their file encryption activities and network communication for delivering ransom notes and communicating with command-and-control servers.

The variations in the average number of API calls across different malware types reflect their specific behaviors and attack mechanisms. For example, trojans (181.196153 average API calls) and droppers (105.883052 average API calls) may have more process- and registry-related API calls for payload execution and persistence.

Coinminers (258.112901 average API calls), keyloggers (207.054134 average API calls), and RATs (173.041249 average API calls) have high average API calls compared to other malware types such as droppers and rats but are still significantly higher than clean samples. This indicates that even these specialized malware types exhibit more complex behavior than benign software.

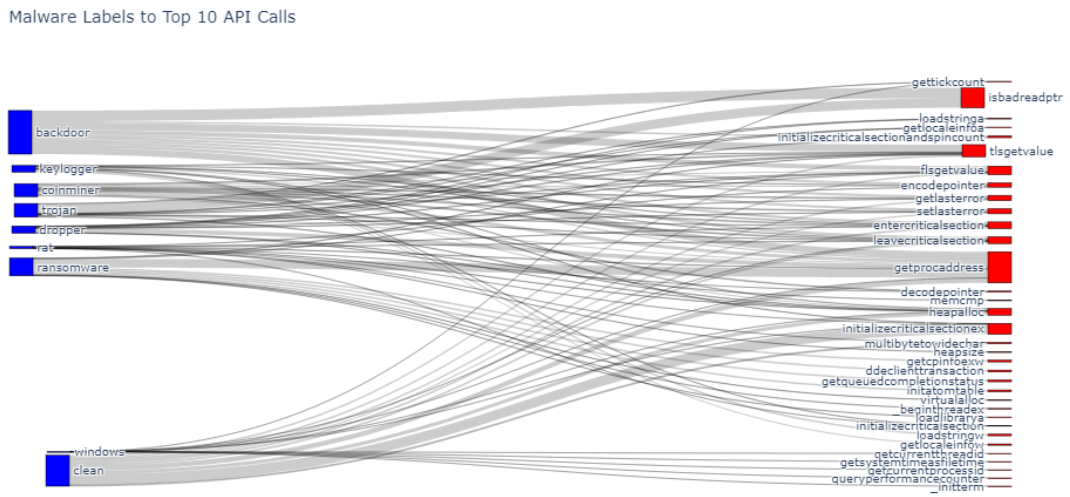

Figure 4. Overlaps of API calls between Malware and Clean Samples

Figure 4 visually represents the overlap in the top 10 API calls between malware and clean samples. The Sankey diagram reveals that certain API calls, such as 'gettickcount', 'loadlibrarya', and 'getmodulehandlea', are commonly used by both malicious and benign samples.

This overlap showcases the challenge of distinguishing malware from benign software based solely on individual API calls. Malware often leverages legitimate system functions to carry out its malicious activities, making it difficult to identify malicious behavior through the presence or absence of specific API calls alone.

5 Related Work

Malware detection has been a critical research focus due to its paramount importance in maintaining the security of cyberspace. Over the years, various methodologies have been proposed, evolving from traditional signature-based techniques to advanced machine learning (ML) and deep learning (DL) approaches (Aslan & Samet, 2020). This

literature review traces the journey of malware detection research, with a specific focus on the recent surge in graph-based learning methodologies and the use of vector embeddings for the detection of Windows-based malware.

The early days of malware detection relied heavily on signature-based methods, which involved identifying unique patterns or signatures in malware samples and using them for detection. While effective against known malware, these methods struggled to cope with the rapid evolution and obfuscation techniques employed by malware authors. Behavior-based and heuristic techniques emerged as a response, aiming to identify malicious behaviors rather than specific signatures. (Aslan & Samet, 2020) provides a comprehensive review of these traditional methods, highlighting their strengths and limitations. They discuss the challenges posed by obfuscation techniques and the need for more advanced detection approaches.

As the volume and complexity of malware grew, researchers turned to machine learning to develop more robust and adaptive detection systems. (Usman et al., 2019) explore the application of representation learning in the cybersecurity domain, laying the foundation for the use of advanced ML techniques. They highlight the potential of deep learning models, such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs), for malware detection and classification. Deep learning, in particular, has shown great promise in malware detection, with (Gopinath & Sethuraman, 2023) providing an extensive review of DL-based techniques. They discuss the effectiveness of various deep learning architectures, including autoencoders, generative adversarial networks (GANs), and graph neural networks (GNNs), for malware analysis and detection.

The rise of graph-based learning methodologies has marked a significant milestone in malware detection research. Graphs provide a natural way to represent the complex relationships and interactions within malware samples, enabling the capture of structural and sequential information. Zhou et al., 2020, Zhang et al., 2020, Wu et al., 2020, and Chen et al., 2020 have presented comprehensive surveys on graph neural networks, highlighting their potential in various domains, including malware detection. These surveys discuss the different types of GNNs, such as graph convolutional networks (GCNs), graph attention networks (GATs), and graph isomorphism networks (GINs), and their applications in modeling complex graph-structured data.

Several notable studies have applied graph-based learning to Windows malware detection, each with its unique approach and contributions. MAGIC (Yan et al., 2019) and HawkEye (Xu et al., 2021) utilize Control Flow Graphs (CFGs) to represent the flow of control within malware samples, employing Deep Graph Convolutional Neural Networks (DGCNNs) and custom Graph Neural Networks (GNNs) respectively for classification. MAGIC focuses on malware detection using CFGs extracted from disassembled binaries, while HawkEye extends this approach to cross-platform malware detection by analyzing both static and dynamic CFGs. While effective, these approaches rely on static analysis, which may struggle with heavily obfuscated or dynamically generated malware.

To address this limitation, researchers have explored the use of dynamic analysis to construct behavioral graphs. DLGraph (Jiang et al., 2018) constructs a Function Call Graph (FCG) from disassembled binaries, using Node2vec and a Stacked Denoising Autoencoder (SDA) for graph embedding and classification. Node2vec is a graph embedding algorithm that learns low-dimensional representations of nodes in a graph, capturing their structural and neighborhood information. The SDA is used to learn a compressed representation of the graph embeddings, which is then used for malware classification. Oliveira et al. (Oliveira & Sassi, 2019) take a similar approach, using API call sequences from sandboxed executions to build behavioral graphs and employing a DGCNN for classification. They demonstrate the effectiveness of graph-based learning in capturing the temporal and structural patterns of API calls for malware detection.

While these graph-based approaches have shown promise, they often rely on complex custom graph neural network architectures, which can be challenging to design and train. Moreover, the interpretability of these models is often limited, making it difficult to understand the reasoning behind their predictions.

Recent advancements in vector embeddings and similarity search have opened up new possibilities for malware detection. Vector embeddings allow for the representation of complex entities, such as malware samples or API call sequences, as dense vectors in a high-dimensional space. By leveraging the geometric properties of this space, similarity search algorithms can efficiently identify similar entities, enabling fast and accurate malware classification.

Our proposed approach seeks to harness the power of vector embeddings and similarity search for malware detection, utilizing the Qdrant vector database and the Hierarchical Navigable Small World (HNSW) graph-based search algorithm (Malkov & Yashunin, 2018). Qdrant is a high-performance vector similarity search engine that enables efficient storage and retrieval of vector embeddings. It supports various indexing and search algorithms, including HNSW, which is a state-of-the-art algorithm for approximate nearest neighbor search in high-dimensional spaces. By representing malware samples and their behavioral characteristics as vector embeddings, we aim to develop a scalable and efficient malware detection system that can adapt to the ever-evolving landscape of malware threats.

Our work is distinct from existing approaches in several ways. First, we utilize a combination of static and dynamic analysis to extract comprehensive behavioral features from malware samples, including API call sequences, function interactions, and system entity communications. We then apply a custom chunking pattern using regular expressions to split the API call sequences into meaningful chunks, with each chunk associated with a sequence tag. This chunking approach allows us to capture fine-grained behavioral patterns and preserve the temporal information of the API calls.

Second, we leverage state-of-the-art vector embedding models, such as BAAI/bge-small-en-v1.5, to generate dense vector representations of these API call chunks. The BAAI/bge-small-en-v1.5 model is a pre-trained language model that has been specifically designed for generating high-quality embeddings for text data. By applying this

model to the API call chunks, we can obtain rich and informative vector representations that capture the semantic and contextual information of the malware behavior. The vector embeddings, along with their associated metadata (including the sequence tags), are stored in the Qdrant vector database, enabling efficient similarity search and retrieval.

Third, we employ the Maximum Marginal Relevance (MMR) algorithm to rerank the retrieved results from Qdrant. MMR is a diversity-based reranking algorithm that aims to select a subset of items that are both relevant and diverse (Carbonell & Stewart, 1999). In our context, MMR allows us to optimize the relevance and diversity of the retrieved API call chunks, ensuring that the most informative and representative chunks are selected for classification. The reranked chunks are then divided into three separate arrays based on their MMR scores, with each array corresponding to a different rank level. This reranking step helps to prioritize the most discriminative behavioral patterns and reduce the impact of noise and redundancy in the retrieved results.

Finally, we perform classification based on frequency analysis of the metadata associated with the chunks in each rank-level array. By analyzing the frequency distribution of the metadata, particularly the sequence tags, we can identify patterns and characteristics that are indicative of malicious or benign behavior. This frequency-based classification approach allows us to make accurate predictions about the nature of the malware samples while providing interpretable insights into the key behavioral factors contributing to the classification decision. The sequence tags play a crucial role in this process, as they capture the temporal order and relationships between the API calls, enabling us to identify malware-specific behavioral patterns.

Our approach offers several advantages over existing methods. By leveraging vector embeddings and similarity search, we can efficiently process and classify large-scale malware datasets, overcoming the limitations of traditional graph-based approaches that rely on complex custom architectures. The use of MMR for reranking ensures that the most relevant and diverse behavioral patterns are considered during classification, enhancing the robustness and generalizability of our system. The Qdrant vector database provides a scalable and efficient solution for storing and retrieving vector embeddings, enabling real-time malware detection and classification.

Moreover, the frequency-based classification approach provides interpretability, allowing analysts to understand behavioral factors contributing to the classification. By leveraging sequence tags and frequency analysis, our approach can provide actionable insights for malware analysis.

In conclusion, our proposed approach represents a significant advancement in the field of malware detection, leveraging the power of vector embeddings, similarity search, and the Qdrant vector database. By combining comprehensive behavioral analysis, vector embedding models, and efficient similarity search algorithms, along with MMR-based reranking and frequency-based classification, we aim to develop a scalable, efficient, and interpretable malware detection system that can effectively combat the ever-evolving landscape of malware threats. This review has set the stage for our study,

which will delve into the details of applying these cutting-edge techniques for malware detection, analyzing the entire pipeline from behavioral feature extraction to classification using the Qdrant vector database, the HNSW algorithm, and MMR-based reranking.

6 Methodology

Our methodology consists of several stages, starting from the raw API call logs, transforming them into a suitable representation for machine learning models, and ultimately performing malware classification.

6.1 API Call Logs

The initial data comprises API call logs in JSON format, which are obtained by dynamically analyzing PE files using the Speakeasy emulation framework (Mandiant, 2022). Each log contains a sequential record of the API calls made by the program during execution, along with their corresponding arguments. These logs offer a detailed view of the program's runtime behavior and its interactions with the operating system and external libraries.

6.2 Data Preprocessing

To prepare the raw API call logs for analysis, we employ a data preprocessing pipeline that involves extracting, cleaning, and transforming each API call and its arguments into an appropriate sequence. This feature extraction process consists of the following key steps:

6.2.1 Normalization

Normalization plays a crucial role in our approach, we apply normalization techniques to file paths, file hashes, domain strings, and IP strings. The idea behind normalization is to generalize the data while preserving its essential structure and patterns.

Our normalization functions handle various data types:

- File paths: Drive letters, network hosts, usernames, and environment variables are substituted with placeholders such as `<drive>`, `<net>`, `<user>`, and their corresponding values.
- File hashes: SHA-256, SHA-1, and MD5 hashes are replaced with placeholders like `<sha256>`, `<sha1>`, and `<md5>`, respectively.

- Domain strings: Domain names are substituted with a placeholder (`<domain>`) using regular expressions.
- IP addresses: Local, private, and public IP addresses are replaced with placeholders such as `<lopIP>`, `<prvIP>`, and `<pubIP>`, respectively, using regular expressions.

By applying these strategies, we mitigate the variability between different logs.

6.2.2 Conversion to CSV

Following normalization, we convert the entire collection of JSON documents into a comma-separated value (CSV) file. We iterate through the JSON dictionary and employ regular expressions to transform the API calls and arguments into a sequential string representation. The resulting CSV file contains rows representing the sequential representation of all API calls and arguments for each malware file, along with their corresponding labels.

6.3 Embedding and Vector Database Storage

To generate vector representations of our data chunks, we utilize a vector embedding model. We also utilize Langchain's text-splitters Python framework (Langchain-Ai, n.d.) to partition the data into chunks. Each chunk comprises a Module Name, API Name, Arguments, and Return Value, extracted using a custom regular expression pattern.

The chunked documents are subsequently passed to the BAAI/bge-small-en-v1.5 embedding model from Hugging Face (Wolf et al., 2020) to generate embeddings. This model produces a 384-dimensional vector representation for each input chunk. The vector representations are indexed and stored in Qdrant (Open, n.d.), a vector database, facilitating efficient similarity search. Due to computational limitations, we downsample our dataset from 80,000 samples to 8,000 samples while preserving the original distribution.

By breaking down the API call sequences into smaller chunks, we can capture behavioral patterns and enable similarity search. Our custom chunking technique, based on the regular expression pattern $< module > (. + ?) < api\_name > (. + ?) < args > (. + ?) < ret\_val > (. + ?)$, allows us to extract pertinent information such as the Module Name, API Name, Arguments, and Return Value, providing a comprehensive representation of each API call.

6.4 Search with Hierarchical Navigable Small World

To efficiently search for similar API call chunks in the high-dimensional vector space, we employ the Hierarchical Navigable Small World (HNSW) algorithm (Malkov & Yashunin,

2018). HNSW is designed for fast Approximate Nearest Neighbor (ANN) searches and is utilized in Qdrant as a vector indexing method.

HNSW constructs a multi-layer graph, with each layer representing a subset of the data. The search process employs a greedy algorithm, navigating from sparse upper layers to denser lower layers, effectively identifying the nearest neighbors. This hierarchical structure enables rapid and accurate similarity searches among large collections of vector embeddings.

During the search process, we employ Maximum Marginal Relevance (MMR) (Carbonell & Stewart, 1999) to retrieve the most similar chunks along with their associated labels. MMR is a technique that aims to select a subset of items that are both relevant to the query and diverse from each other. In our methodology, MMR is applied to the retrieved chunks to ensure that the selected results are not only similar to the query but also provide a diverse representation of the relevant information.

MMR is calculated based on the similarity of each chunk to the query and the dissimilarity to the previously selected chunks. The dissimilarity term in MMR helps to promote diversity in the retrieved results, reducing redundancy and potentially capturing a wider range of relevant behavioral patterns.

The use of MMR in our methodology is inspired by prior research in information retrieval and text summarization (Carbonell & Stewart, 1999). However, to our knowledge, the application of MMR in the context of malware analysis and API call chunk retrieval is a novel contribution of our work.

6.5 Classification

Once the database has returned the labels of the chunks considered most relevant to our query, we focus on the top three results across the API call sequence. The rationale behind selecting the top three results is to balance considering the most relevant information and maintaining a manageable set of results for classification. By limiting the number of results, we aim to capture the most significant behavioral patterns while avoiding potential noise and computational overhead associated with processing a larger set of results.

To arrive at a classification decision, we employ a frequency analysis approach. We analyze the frequency distribution of the retrieved labels and assign the final classification based on the most frequently occurring label among the top three results. This approach assumes that the most recurring label across the top results is likely to be indicative of the overall behavior and characteristics of the malware sample.

In addition to the frequency analysis, we also explore an alternative classification approach using fine-tuned language models (LLMs). We fine-tune an LLM using a supervised prompt-based approach, where the model is trained to make a classification decision based on the patterns and characteristics observed in the metadata of the

retrieved chunks. The fine-tuned LLM can potentially capture more complex relationships and dependencies among the API calls and their arguments, providing a complementary perspective to the frequency-based classification.

By combining the frequency analysis and the fine-tuned LLM approach, we aim to enhance the robustness and accuracy of our malware classification methodology. The frequency analysis provides a straightforward and interpretable classification based on the most prevalent labels, while the LLM-based approach allows for a more nuanced understanding of the behavioral patterns and characteristics present in the API call sequences.

6.6 Evaluation

To assess the effectiveness of our methodology, we utilize standard metrics such as accuracy, precision, recall, and F1-score. We compare the predicted labels obtained from our classification approach with the ground truth labels associated with each malware sample. The evaluation metrics provide insights into the performance of our methodology in correctly identifying and classifying different types of malware based on their API call behavior.

Furthermore, we generate confusion matrices to visualize the performance of our classification approach across different malware categories. The confusion matrices allow us to identify potential strengths and weaknesses of our methodology in distinguishing between different types of malware and help us gain insights into the specific behavioral patterns that may be more challenging to classify accurately.

7 Results

In this section, we present and analyze the results of our framework developed to classify the behavioral dynamics of Portable Executable (PE) files. Our approach combines dynamic analysis, high-dimensional embeddings using the fastembedd library from Qdrant, with the BAAI-BGE-Small - EN V.1.5 vector embedding model, and the precision of vector space modeling. These components are integrated through the utilization of the Hierarchical Navigable Small World (HNSW) search algorithm. The primary goal is to differentiate entire API call sequences and classify them into one of 9 categories.

7.1 Dynamic Analysis and Classification Performance

Dynamic analysis serves as the foundation of our approach, allowing us to capture the executable actions within a controlled environment and establish a baseline of activity. By running the PE files in a simulated environment and monitoring their behavior, we obtain valuable insights into the API call sequences that characterize different types of malware and benign software.

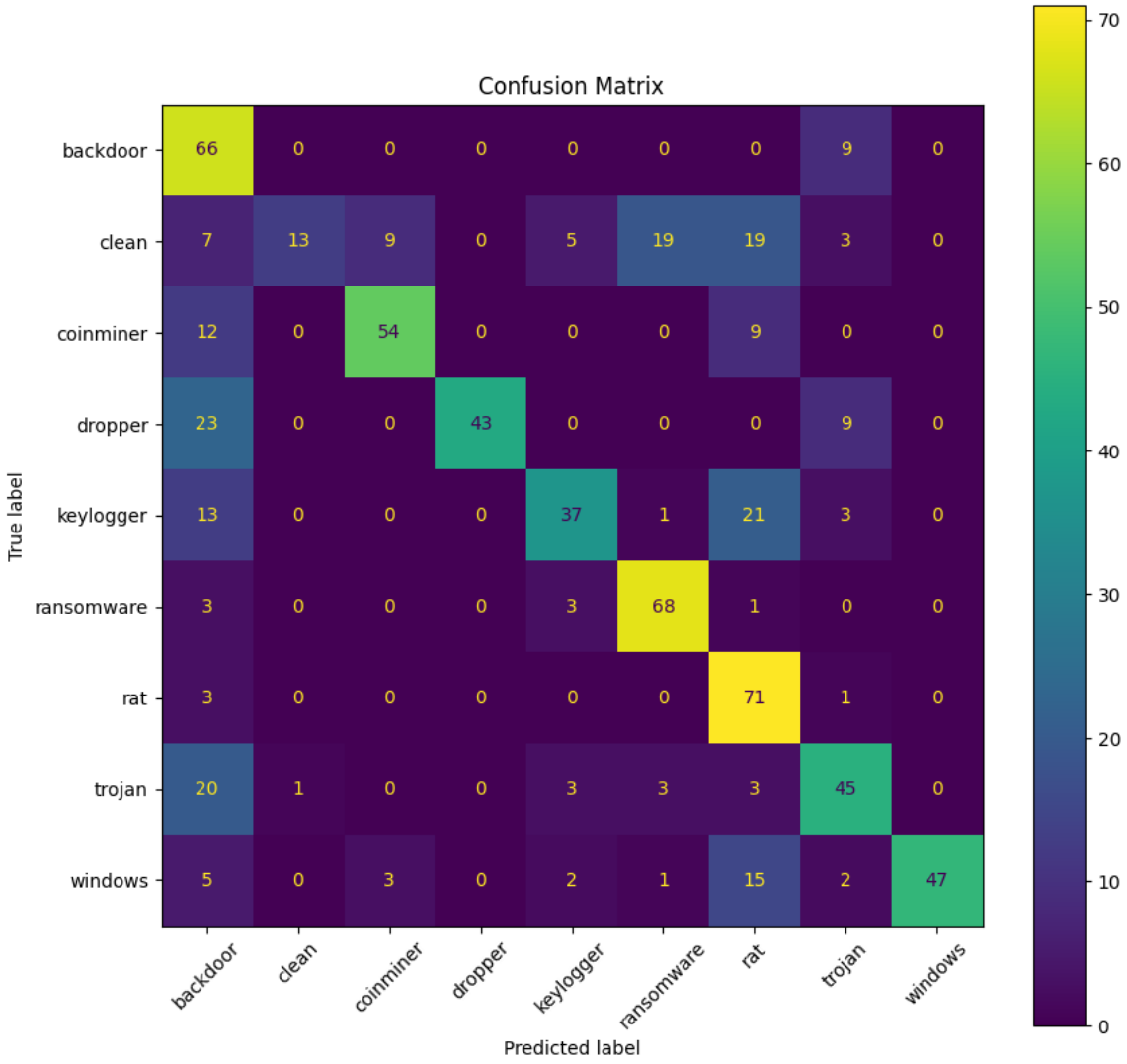

Figure 5. Confusion Matrix

The confusion matrix in Figure 5 provides a detailed breakdown of the classification outcomes. The prominent values along the diagonal of the matrix indicate the instances where the frequency analysis successfully identified the true nature of the API call sequences. For example, the ”backdoor” category, with a high frequency of 66, demonstrates that the classification often aligns with the actual label. Similar trends can be observed for the ”ransomware” and ”rat” categories, where the frequency analysis correctly identified the majority of true positives, as evidenced by the counts of 68 and 71, respectively.

However, the confusion matrix also reveals instances of wrong classification, particularly where ”clean” sequences are incorrectly flagged as other categories such as ”coinminer”, ”keylogger”, ”rat”, and ”ransomware”. This pattern highlights the challenges in distinguishing between benign and malicious behaviors based solely on the frequency of certain API calls. Furthermore, there are notable misclassifications between different malware categories, such as ”trojan” being misclassified as ”backdoor” and ”keylogger” being misclassified as ”rat”. These misclassifications suggest that there are similar patterns among different malware types, making accurate classification more difficult.

Table 1. Classification Report

Category	**Precision**	**Recall**	**F1-score**	**Support**
backdoor	0.43	0.88	0.58	75
clean	0.93	0.17	0.29	75
coinminer	0.82	0.72	0.77	75
dropper	1.00	0.57	0.73	75
keylogger	0.74	0.49	0.59	75
ransomware	0.74	0.91	0.81	75
rat	0.51	0.95	0.66	75
trojan	0.62	0.60	0.61	75
windows	1.00	0.63	0.77	75
accuracy			0.66	675
macro avg	0.76	0.66	0.65	675
weighted avg	0.76	0.66	0.65	675

The classification report in Table 1 provides a quantitative assessment of the frequency analysis results. Categories such as ’clean’, ’dropper’, and ’windows’ exhibit strong precision scores, indicating that when these labels are predicted, they are usually accurate. The ”clean” category, in particular, achieves a precision of 0.93, suggesting that the classifier is highly effective in identifying benign sequences. Similarly, the ”dropper” and ”windows” categories also show high precision scores of 1.00, indicating that when these labels are predicted, they are almost always correct.

However, the recall scores for these categories are comparatively lower, suggesting that some instances of these categories are being missed by the classifier. For example, the recall score for the ”clean” category is 0.17, implying that a significant portion of clean sequences are being misclassified as other categories. This pattern is also evident in the ”windows” category, which has a recall score of 0.63.

Conversely, categories like ”backdoor”, ”coinminer”, and ”ransomware” show higher recall scores, implying that the classifier is capable of correctly identifying a larger proportion of these malware types. The ”ransomware” category, in particular, achieves a recall score of 0.91, indicating that the classifier is highly effective in identifying ransomware sequences. The ”backdoor” and ”coinminer” categories also exhibit relatively high recall scores of 0.88 and 0.72, respectively.

The F1-scores, which provide a balanced measure of precision and recall, are highest for ”coinminer” (0.77), ”dropper” (0.73), and ”windows” (0.77). These scores suggest that the classifier performs well in terms of both precision and recall for these categories.

The overall accuracy of our approach, as reported in the classification report, is 66%. This means that approximately two-thirds of the API call sequences were classified correctly. While this accuracy is promising, it also indicates that there is room for improvement in the classification performance.

The macro average and weighted average scores offer a summarized view of the classifier's performance across all categories. The macro average precision, recall and F1-score are 0.76, 0.66, and 0.65, respectively, suggesting a reasonable overall performance considering the class imbalance present in the dataset. The weighted average scores take into account the support (number of instances) for each category, resulting in a precision of 0.76, recall of 0.66, and F1-score of 0.65.

7.2 Analysis and Discussion

The results obtained from our frequency-based classification approach demonstrate the potential of using API call sequences to identify and classify different types of malware. The confusion matrix and classification report provide valuable insights into the strengths and weaknesses of our approach.

One notable observation is the high precision scores achieved for the ”clean”, ”dropper”, and ”windows” categories. This indicates that when the classifier predicts these labels, it is highly likely to be correct. However, the lower recall scores for these categories suggest that the classifier is missing a significant portion of instances belonging to these categories. This could be attributed to the similarities in API call sequences between benign and malicious software, making it challenging to distinguish between them based solely on frequency analysis.

On the other hand, the higher recall scores for categories like ”backdoor”, ”coinminer”, and ”ransomware” indicate that the classifier is effective in identifying a larger proportion of these malware types. This suggests that the behavioral patterns exhibited by these malware categories are more distinct and easily distinguishable from other categories.

The misclassifications between different malware categories, such as ”trojan” being misclassified as ”backdoor” and ”keylogger” being misclassified as ”ransomware”, highlight the overlapping and similar behavioral patterns among different malware types. This suggests that there may be shared API call sequences or common behaviors exhibited by these categories, making accurate classification more challenging.

To address these challenges and improve classification performance, several approaches can be explored. One potential area for enhancement is the weighting scheme applied to different API calls. By carefully examining the importance and discriminative power of specific API calls, we can adjust the weights to better differentiate between

malware types and reduce misclassifications. Additionally, incorporating contextual information and considering the sequence and dependencies of API calls may provide valuable insights and improve the classifier's ability to distinguish between benign and malicious behaviors.

Another avenue for improvement is the exploration of more advanced machine learning techniques. While the frequency-based approach serves as a solid foundation, employing deep learning models or ensemble methods may help capture more intricate patterns and relationships within the API call sequences. These techniques have the potential to uncover hidden features and dependencies that are not easily discernible through frequency analysis alone.

Furthermore, expanding the dataset to include a larger and more diverse collection of PE files could enhance the robustness and generalization capability of the classifier. By training on a wider range of malware samples and benign software, the classifier can learn to recognize a broader spectrum of behavioral patterns and improve its accuracy in real-world scenarios.

It is important to acknowledge that malware classification is an ongoing challenge, as malware authors continuously evolve their techniques to evade detection and obfuscate their code. Therefore, regular updates and retraining of the classification model are necessary to keep pace with the ever-changing malware landscape.

Despite the limitations and areas for improvement, our frequency-based classification approach demonstrates the feasibility of using API call sequences as a means to identify and classify malware. The combination of dynamic analysis, vector embeddings, and efficient search algorithms provides a promising foundation for further research and development in this field.

7.3 Future Work and Alternatives Tried

While our frequency-based classification approach using dynamic analysis, vector embeddings, and efficient search algorithms has shown promising results, there are limitations that need to be addressed. These limitations open up avenues for future work and improvements.

One potential direction for future work is to leverage Large Language Models (LLMs) to directly classify the API call sequences. This approach relies on the context windows of LLMs, which determine the number of tokens they can process at a given time. As new technologies and architectures emerge, such as Mamba, which offers linear-time sequence modeling with selective state spaces, the limitations associated with token lengths may be overcome, enabling more effective classification using LLMs.

As an alternative approach, we have explored using prompt fine-tuning on a dataset generated from the metadata retrieved from our database of 8,000 samples. We selected a balanced distribution of 75 samples for each class and prepended the metadata with a prompt. The generated dataset was then used for prompt fine-tuning using Unsloth, a library written in Triton that allows for running quantized LLMs on lower-end GPUs.

While this approach sacrifices some accuracy for memory efficiency, it is suitable for tasks like sentiment classification or database tagging.

A sample of the data used for prompt fine-tuning is shown below:

[”input_text”: ”Classify the following into one of the following classes [clean, backdoor, ransomware, trojan, dropper, coinminer, keylogger, rat, windows], text: windows 1; rat 135; windows 2; clean 4; rat 5; clean 6; coinminer 14; clean 8; clean 30; clean 10; ransomware 32; backdoor 11; clean 13; clean 6; coinminer 14; clean 8; clean 8; clean 8; clean 8; clean 165; clean 8; clean 8; clean 8; clean 165; clean 165; clean 165; clean 8; clean 8; clean 52; clean 30; clean 10; ransomware 32; clean 152; clean 34; .... (truncated for brevity), ”output_text”: ”clean”]

The fine-tuning process was monitored using various metrics, including learning rate, training time, training loss, and train samples per second. These metrics, as shown in Figures 13-16, provide insights into the progress and performance of the model during training.

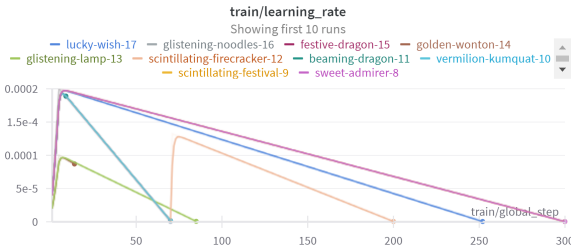

Figure 6. Learning Rate

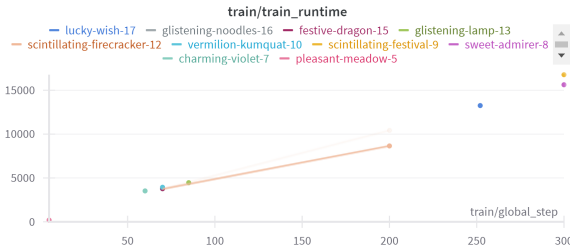

Figure 7. Training Time

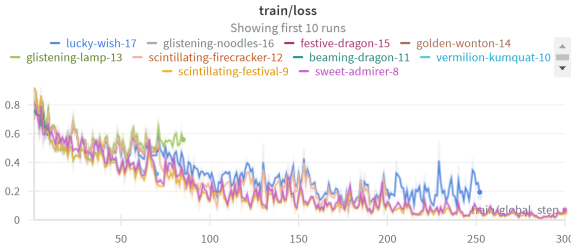

Figure 8. Training Loss

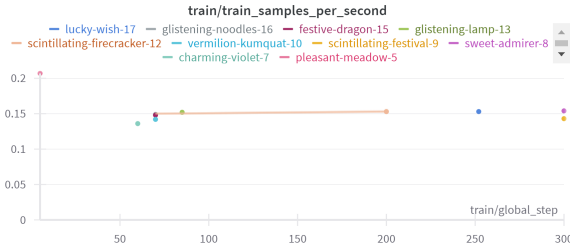

Figure 9. Train Samples per second

Upon testing the trained model, we observed that while it accurately classified data seen during training, it struggled to generalize well to unseen data. The model tended to output either ”clean” or ”backdoor” for most unseen test inputs. This issue highlights the need for further improvements in the training process and data variability.

To address the generalization issue, we propose increasing the variability and amount of training data by expanding the pool in the Qdrant database to include all 80,000 samples from our dataset. By incorporating more diverse data during training, the model’s ability to generalize to unseen samples can be enhanced.

Another approach to improve the model's performance is to use a non-quantized LLM. In our test run, we used a 4-bit quantized LLM, which resulted in a loss of parameter size and quality compared to the original brain float 16 models. Using a higher-precision LLM could potentially yield better classification results, albeit at the cost of increased computational requirements.

Generating better embeddings is another area for future exploration. One approach is to pre-train a SentencePiece embedder specifically on API call sequence data, allowing for more domain-specific representations. Alternatively, using an LLM with a context set by prompting could also lead to improved embeddings. However, these approaches are computationally intensive and were limited by the hardware available in our current setup.

The complete Weights and Biases report for our experiments can be found at https://api.wandb.ai/links/nj9814/aephkc13. This report provides detailed insights into the training process and model performance.

In conclusion, while our current approach has shown promise, there are several avenues for future work and improvement. Exploring the use of LLMs for direct classification, increasing data variability, using non-quantized models, and generating better embeddings are all potential directions to enhance the performance and generalization ability of our malware classification system. As we continue to refine our techniques and address the identified limitations, we aim to develop a more robust and accurate approach to behavioral-based malware classification.

8 Conclusion

This project has explored the use of frequency analysis on API call sequence metadata for the classification of malware types. By leveraging the intricate patterns that distinguish malware from benign software, our approach offers a unique perspective on malware classification and threat detection.

The confusion matrix highlights the method's substantial capability in accurately identifying certain malware categories, such as 'ransomware' and 'rat'. However, instances of misclassification, particularly between 'clean' and 'backdoor' sequences, underscore the challenges in differentiating between deceptively similar behaviors.

The classification report demonstrates the overall efficacy of our approach, with an accuracy of 66%. The precision of certain malware types indicates a strong alignment between predicted and actual labels, while the F1 scores provide a balanced measure of performance. Disparities in recall, especially for the 'clean' label, suggest the need for further refinements, such as adjusting metadata frequency weightings or expanding the considered attributes.

In conclusion, the application of frequency analysis to API call sequences has shown considerable promise as an innovative approach to malware detection. The insights gained contribute to a broader understanding of malware mechanics and provide a

foundation for future improvements.

8.1 Limitations

While our approach has demonstrated potential, it is important to acknowledge its limitations:

High-Dimensional Data Complexity: The performance of HNSW can be affected by the curse of dimensionality, leading to longer search times and less accurate results, especially with extremely high-dimensional data. Resource Intensive: Generating embeddings for each API call and storing them in a vector database can be computationally and storage-intensive, particularly with large datasets. Accuracy vs. Speed Trade-off (HNSW): As an approximate nearest neighbor search algorithm, HNSW may not always return the exact nearest neighbor. Achieving better accuracy may require more computational resources, such as using higher-parameter LLMs for generating embeddings and providing context for API calls. Despite these limitations, our approach offers a valuable contribution to the field of malware classification and opens up avenues for further research and development

References

Aslan, Ö. A., & Samet, R. (2020). A comprehensive review on malware detection approaches. *IEEE Access*.

Av-test atlas 2008 - 2024. (2024). https://portal.av-atlas.org/malware/statistics

Bayer, U., Moser, A., Kruegel, C., & Kirda, E. (2006). Dynamic analysis of malicious code. *Journal in Computer Virology*.

Carbonell, J., & Stewart, J. (1999). The use of mmr, diversity-based reranking for reordering documents and producing summaries. *SIGIR Forum (ACM Special Interest Group on Information Retrieval)*. https://doi.org/10.1145/290941.291025

Chen, F., Wang, Y.-C., Wang, B., & Kuo, C.-C. J. (2020). Graph representation learning: A survey. *APSIPA Transactions on Signal and Information Processing, Cambridge University Press*.

Damodaran, A., Troia, F., Visaggio, C., et al. (2017). A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, *13*, 1–12. https://doi.org/https://doi.org/10.1007/s11416-015-0261-z

Galloro, N., Polino, M., Carminati, M., Continella, A., & Zanero, S. (2022). A systematical and longitudinal study of evasive behaviors in windows malware. *Computers and Security*, 102550.

GeoffChappel. (2023). *Windowsinternals*. Retrieved 2023, from https://www.geoffchappell.com/

Gopinath, M., & Sethuraman, S. C. (2023). A comprehensive survey on deep learning based malware detection techniques. *Computer Science Review, Elsevier*.

Igor Santos, J. D. (2013). Opem: A static-dynamic approach for machine-learning-based malware detection. *International Conference CISIS12-ICEUTE12*, 271–280.

Jiang, H., Turki, T., & Wang, J. T. (2018). Dlgraph: Malware detection using deep learning and graph embedding. *2018 17th IEEE international conference on machine learning and applications (ICMLA)*.

Langchain-Ai. (n.d.). Langchain-ai/langchain:build context-aware reasoning applications. https://github.com/langchain-ai/langchain

Malkov, Y. A., & Yashunin, D. A. (2018). Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs.

Mandiant. (2022). Speakeasy emulation framework (1.59). https://github.com/mandiant/speakeasy

Microsoft. (2024). *Msdn*. Retrieved 2024, from https://learn.microsoft.com/en-us/windows/win32/api/

Oliveira, A., & Sassi, R. (2019). Behavioral malware detection using deep graph convolutional neural networks. *Techlixiv.[link]*.

Open, S. (n.d.). Qdrant vector database. https://qdrant.tech/

Svitlana Storchak, S. P. (2023). *Windowsinternals*. Retrieved 2023, from https://www.vergiliusproject.com/

Trizna, D., Demetrio, L., Biggio, B., & Roli, F. (2023). Nebula: Self-attention for dynamic malware analysis.

Usman, M., Jan, M. A., He, X., & Chen, J. (2019). A survey on representation learning efforts in cybersecurity domain. *ACM Computing Surveys (CSUR), ACM New York, NY, USA*.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T. L., Gugger, S., … Rush, A. M. (2020). Huggingface's transformers: State-of-the-art natural language processing.

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., & Philip, S. Y. (2020). A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems, IEEE*.

Xu, P., Zhang, Y., Eckert, C., & Zarras, A. (2021). Hawkeye: Cross-platform malware detection with representation learning on graphs. *Artificial Neural Networks and Machine Learning–ICANN 2021: 30th International Conference on Artificial Neural Networks, Bratislava, Slovakia, September 14–17, 2021, Proceedings, Part III 30*.

Yan, J., Yan, G., & Jin, D. (2019). Classifying malware represented as control flow graphs using deep graph convolutional neural network. *2019 49th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*.

Ye, Y., Li, T., Adjeroh, D., & Iyengar, S. S. (2017). A survey on malware detection using data mining techniques. *ACM Computing Surveys (CSUR)*.

Yosifovich, P. (2017). *Windows internals part 1 7th edition*. Pearson. https://www.microsoftpressstore.com/store/windows-internals-part-1-system-architecture-processes-9780735684188

Zhang, Z., Cui, P., & Zhu, W. (2020). Deep learning on graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering, IEEE*.

Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., & Sun, M. (2020). Graph neural networks: A review of methods and applications. *AI Open, Elsevier*.