Master's thesis
Master's Programme in Data Science

Adversarial Robustness of Hybrid Machine Learning Architecture for Malware Classification

Dmitrijs Trizna

May 12, 2022

Supervisor(s): Assoc. Prof. Antti Honkela

Examiner(s): Assoc. Prof. Antti Honkela
Assoc. Prof. Nikolaj Tatti

University of Helsinki
Faculty of Science

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Degree programme
Faculty of Science	Master's Programme in Data Science

Tekijä — Författare — Author

Dmitrijs Trizna

Työn nimi — Arbetets titel — Title

Adversarial Robustness of Hybrid Machine Learning Architecture for Malware Classification

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidantal — Number of pages
Master's thesis	May 12, 2022	65

Tiivistelmä — Referat — Abstract

The detection heuristic in contemporary machine learning Windows malware classifiers is typically based on the static properties of the sample. In contrast, simultaneous utilization of static and behavioral telemetry is vaguely explored. We propose a hybrid model that employs dynamic malware analysis techniques, contextual information as an executable filesystem path on the system, and static representations used in modern state-of-the-art detectors. It does not require an operating system virtualization platform. Instead, it relies on kernel emulation for dynamic analysis. Our model reports enhanced detection heuristic and identify malicious samples, even if none of the separate models express high confidence in categorizing the file as malevolent. For instance, given the 0.05% false positive rate, individual static, dynamic, and contextual model detection rates are 18.04%, 37.20%, and 15.66%. However, we show that composite processing of all three achieves a detection rate of 96.54%, above the cumulative performance of individual components. Moreover, simultaneous use of distinct malware analysis techniques address independent unit weaknesses, minimizing false positives and increasing adversarial robustness. Our experiments show a decrease in contemporary adversarial attack evasion rates from 26.06% to 0.35% when behavioral and contextual representations of sample are employed in detection heuristic.

ACM Computing Classification System (CCS):
Security and privacy → Intrusion/anomaly detection and malware mitigation → Malware and its mitigation
Computing methodologies → Machine learning → Machine learning algorithms

Avainsanat — Nyckelord — Keywords

adversarial machine learning, malicious software, windows, deep learning

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1. Introduction

Machine learning (ML) algorithms have become an essential component of malicious software (malware) detection in conventional cybersecurity intrusion prevention systems. Such systems can learn common patterns across a vast dataset of known malware, obtaining a predictive power to classify previously unseen malicious samples.

Several publicly available solutions are considered State-of-The-Art (SoTA) models. The convolutional deep learning model introduced by Raff et al. [40] and the feed-forward neural network (FFNN) introduced by Rudd et al. [42] are example of "featureless" representation learning from a Portable Executable on a byte level. On the contrary, the gradient boosted decision tree (GBDT) model shown by Anderson and Roth [5] employs sophisticated feature engineering based on PE structure. These models achieve remarkable results in the identification of malicious samples.

At the same time, during the last decade, the machine learning community realized that model behavior could be affected by the presence of a potential adversary. Producing perturbations on input data threat actors capable of creating an adversarial example, which leads to unexpected performance of ML models like misclassification [48].

While originally adversarial attacks were harnessed in the domain of computer vision, the concept spread to a malware classification problem, with several high-efficiency adversarial algorithms known today [14, 30, 41, 45]. Attacks based on these algorithms severely decrease the performance of the SoTA solutions mentioned above, while preserving malicious functionality. This is achieved by altering static properties of Portable Executable files (with few exceptions that produce behavioral perturbations [10, 41]). The effectiveness of such attacks happens since SoTA solutions employ representations from static properties of executable files. As a consequence, attacks have a direct effect on the model's input vector.

Moreover, there is evidence that existing SoTA models lack epistemic capacity due to limited contextual awareness, and adding this information boosts the performance of malware classification task [31]. Incorporating system logging as input of ML models for malware classification is a remarkably complex task. Therefore, we consider that it might require a hybrid solution with multiple parallel processing tasks performing

analysis on different aspects of an executable file. While the academy favors end-to-end solutions, we argue that security problem with various telemetry sources is too diverse to propose a single, homogeneous model. Additionally, it is shown by Yang et al. [51] that composite neural networks with a high probability surpass the performance of any pre-trained components. We assume that this property holds in the direction of malware classification applied models.

Therefore, we hypothesize that *malware classification based on a hybrid machine learning architecture, which incorporates static, contextual, and behavioral patterns of software, can (a) improve classification performance and (b) yield better robustness against adversarial attacks.*

To evaluate this hypothesis, in addition to static properties of binary files, we incorporate data produced by threat intelligence with in-the-wild filesystem paths of samples. Moreover, we narrow the epistemic gap further by adding behavioral properties of Portable Executables obtained from dynamic analysis with a Windows kernel emulator. Attempts to utilize emulated telemetry are still vaguely explored in applied machine learning research. As a consequence, we construct a modular machine learning architecture that relies on four modules:

- file-path 1D convolutional neural network
- emulated API call 1D convolutional neural network
- pre-trained Raff et al. [40] neural network
- pre-trained Anderson and Roth [5] GBDT model

The lack of publication artifacts is a notorious drawback in any research and contributes to science's reproducibility crisis. Hence we release pre-trained PyTorch [37] models, as well as provide widely adopted `scikit-learn`-like [39] API interface to our composite model in public code repository.*

We consider main contributions of our work are as follows:

- to our knowledge, we are the first to build a composite machine learning model that incorporates static, dynamic, and contextual malware analysis telemetry without the need of any virtualization platform;
- we perform adversarial robustness evaluation of a our model, with identification of drawbacks in contemporary adversarial malware generation algorithms;
- collection and release of an anonymized dataset representing emulation reports of **108204** executable files, significantly larger than in related research on behavioral malware analysis;

*https://github.com/dtrizna/quo.vadis

- implementation of anti-anti-debugging techniques and contribution to open-source Windows kernel emulator [33] based on the evasion methods observed in the malware corpus;
- definition and release of a noteworthy file path augmentation logic inherited from our file path dataset.

This work is structured as follows. Chapter 2 covers the necessary background knowledge essential for concepts discussed in later chapters. Chapter 3 discusses the most influential techniques used in contemporary malware classification research based on machine learning algorithms. Chapter 4 enlightens the structure and details of dataset we are using. Chapter 5 does a methodical description of our hybrid machine learning architecture and its performance. Chapter 6 covers current achievements in the direction of adversarial sample generation. Chapter 7 evaluates these techniques against our model and report results of robustness experiments. Finally Chapter 8 discuss further development ideas and Chapter 9 concludes this work.

2. Background

This chapter describes the underlying concepts behind machine learning powered malware classification. Section 2.1 enlightens the structure of Portable Executable format, Section 2.2 covers taxonomy and methods of conventional malware analysis, Section 2.3 explains the idea behind Windows kernel emulation, with challenges and limitations of this approach.

2.1 Portable Executable Format

The Windows Portable Executable (PE)[†] describes the structure of launchable image files under the Windows family of operating systems. It describes both executables (commonly having `exe` extension) and libraries (extension `dll`). Terms "executable", "binary", and "image" might be used interchangeably to represent PE files. We will not provide a comprehensive description of all structures with PE files but cover a high-level view of the main components and a deeper explanation of fields relevant for malware analysts, which are mostly influenced by adversarial algorithms.

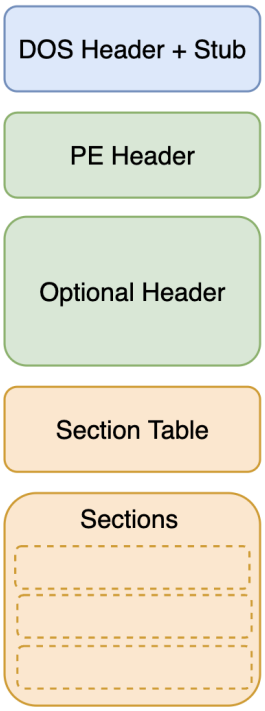

Figure 2.1: Portable Executable structure [15].

DOS Header and Stub. To remain compatible with previous versions of MS-DOS and Windows, the PE file format retains the old `MZ` header from MS-DOS [53]. It acts as the ubiquitous distinctive property of PE files, with the first two bytes being `0x4d5a` in hexadecimal notation or `MZ` as a string.

PE Header. Acts as the "real" header of the file, containing meta-data about an application, including target processor architecture, compilation timestamp, and other general characteristics [53].

Optional Header. Despite the presence of term "op-

[†] https://docs.microsoft.com/en-us/windows/win32/debug/pe-format

tional" within the title, this is not an optional entry in PE executable files. The optional header contains the essential information about the executable image, such as initial stack size, program entry point location, preferred base address, operating system version, section alignment information, and counting [53].

Sections Table. This structure contains entries about every Section represented within the binary, their names, size, real addresses (location within the image), and virtual addresses (where section content resides in memory during the execution) [53].

Sections are byte blobs that contain execution flow instructions and embedded data. There are multiple common sections: the code of the program represented by *.text*, initialized data in *.data*, read-only structures within *.rdata*, and so on [53]. Even though such section structure is represented by a supreme majority of PE files and compilers, it is important to emphasize that most section names are not explicitly predefined or mandatory present. Contemporary adversarial malware algorithms exploit this property [14, 45, 30]. Another example is packed binaries, which have altered section names, to give an instance files packed with UPX* have only three sections: `NPX0`, `UPX1`, and `.rsrc`.

Packed sample IAT

name (10)	library (5)
RegFlushKey	advapi32.dll
InternetOpenA	wininet.dll
57 (gethostvalue)	ws2_32.dll
VirtualProtect	kernel32.dll
VirtualAlloc	kernel32.dll
VirtualFree	kernel32.dll
ExitProcess	kernel32.dll
LoadLibraryA	kernel32.dll
GetProcAddress	kernel32.dll
GetDC	user32.dll

Un-packed sample IAT

name (28)	library (5)
RegSetValueExA	advapi32.dll
RegOpenKeyExA	advapi32.dll
RegDeleteValueA	advapi32.dll
RegFlushKey	advapi32.dll
RegCloseKey	advapi32.dll
HttpSendRequestA	wininet.dll
InternetQueryDataAvailable	wininet.dll
InternetReadFile	wininet.dll
InternetCloseHandle	wininet.dll
HttpQueryInfoA	wininet.dll
InternetConnectA	wininet.dll
InternetOpenA	wininet.dll
HttpOpenRequestA	wininet.dll
InternetSetOptionA	wininet.dll
52 (gethostbyvalue)	ws2_32.dll
116 (WSACleanup)	ws2_32.dll

Figure 2.2: Import Address Table (IAT) comparison of the same malware sample (BRB Bot) before and after (cropped) unpacking.

Practically all PE files import functionality from complementary libraries, most commonly to interact with the operating system via kernel API calls (in various resources referred to as "system calls"). Information about what methods are imported is represented within **Import Address Table (IAT)** which is stored as *.idata* section

*https://upx.github.io/

[53]. This is an important structure and is often used as a part of decision heuristic by both machine learning [5, 31] and conventional malware classification [9].

It indeed has a positive correlation with actual functionality of binary, however, given an intent to mislead analysts, it may not represent the actual capabilities of binary [46]. This is because it is possible to add futile entries in IAT, that will never be used within an execution flow. On the other hand, it is possible to hide actual libraries and kernel API calls, for example, using a direct *syscalls** or through the utilization of packers, as shown in Figure 2.2.

Similarly to IAT, **Export Address Table (EAT)** is a data structure stored within *.edata* section, and describes methods available for execution by other applications like `rundll32.exe`. It is common for DLL files to export multiple entry points, allowing other PEs to utilize exported methods, whereas `exe` files have only a single, *main* entry point. Otherwise, from a structural point of view, executables and library file types are the same. Given the source code of the application, it is possible to recompile the executable as a library with only a few modifications.

2.2 Malware Analysis Methodologies

Conventional malware inspection consists of two major strategies - static and dynamic analysis. Static investigations involve collecting known properties of executable files and analyzing disassembled code procedures. The dynamic examination goes a step further, with actual "detonation" of malware files in a controlled environment, thus allowing behavior record of the infected system or in-memory debugging of malware functionality [46].

Theoretically, most of the executable's embedded functionality can be inferred based on static reverse engineering only, except for additional content fetched from a remote location during the runtime. Disassembled instructions can provide an exhaustive graph with exact execution instructions and decisions the application will take [46].

Practically, however, and especially in the malware analysis domain, this is not reasonable, given limited time and human resources. Malware authors *a priori* are willing to conceal the specifics and true nature of their work. In the first place, this is needed to evade detection. However, even after identification, misleading analysts help avoid disclosing Indicators of Compromise (IoC), like addresses and specifics of Command and Control (C&C, C2) infrastructure [46].

Therefore, it is common to hide most malicious functionality from static analysis

*https://www.mdsec.co.uk/2020/12/bypassing-user-mode-hooks-and-direct-invocation-of-system-calls-for-red-teams/

through content obfuscation, for example, string encoding through an XOR operation or utilization of packers. Therefore, dynamic analysis often yields much faster and more precise results. The typical approach used by malware analysts is to execute a malicious sample in a controlled environment, allow a specimen to perform deobfuscation with expected system infection, and acquire a comprehensive record of its behavior [46].

Debugging allows to accomplish the same, but in a more controlled manner, by single-stepping through the instructions of operating software, pausing an execution (also known as *breakpoint*) of application at specific stages, or even *patching* malicious specimen to express the desired behavior [46]. High-level taxonomy of malware analysis methods and tools representative for each technique is visualized in Figure 2.3.

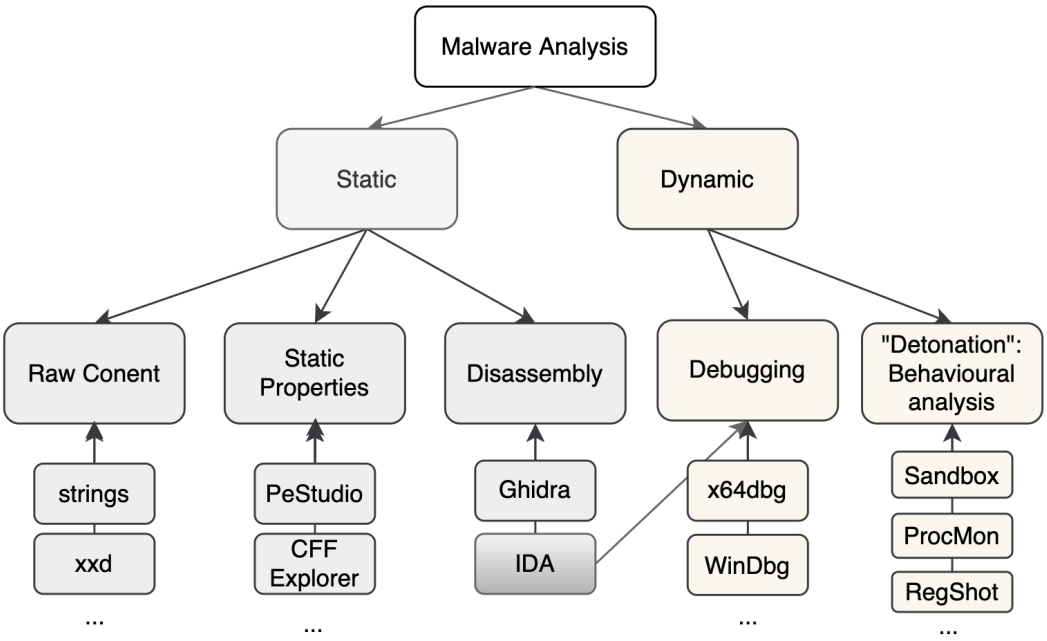

Figure 2.3: Taxonomy of Malware Analysis with the most common tools mentioned (list is not exhaustive). Note IDA [17] stands out as a Malware Analytics "swiss-knife", capable of both decompiling and debugging samples.

Since security is a never-ending cat and mouse game, malware authors and researchers are trying to circumvent analysis techniques with offensive countermeasures. It is common to see anti-debugging functionality and sandbox evasion in modern malware [38]. Therefore, we cover such techniques seen in our dataset and manipulations to minimize their impact on the emulation model in more detail within Section 4.1.1. A notable example of ML algorithm utilization for offensive needs is shown by Pearce et al. [38], who presented that it is possible to evade dynamic malware analysis by embedding a pre-trained sandbox classifier within the malware.

2.3 Windows Kernel Emulation

A noticeable operational challenge of dynamic malware analysis is the necessity to interact with malware manually or maintain sandboxes. Observing malware in a sandbox is costly from computational resource and execution time points of view. Therefore, it is hard to scale to quantities beneficial for deep learning.

A notable compromise might bring an analysis of malware specimens with kernel emulation techniques. While sandboxes utilize the merits of virtualization technology, with actual execution of binaries on real operating system kernel, emulators "pretend" to be an operating system, spoofing memory, network, and other hardware or software component interfaces [33, 7].

Since emulators do not require to mobilize full-fledged operating system operations, they allow getting vast amounts of telemetry reasonably fast. The emulation-produced report allows to bypass static analysis limitations, obtaining a sequence of kernel API calls made by the executable, identifying manipulated or saved files, registry entries, and attempted network communications [33].

We use Speakeasy [33], a Python-based emulator released and actively maintained by Mandiant under the MIT license. The Speakeasy version used in our tests is 1.5.9. It relies on QEMU [7] CPU emulation framework.

2.3.1 Emulation Limitations

While emulators have apparent benefits for extending static analysis, some crucial drawbacks establish boundaries where emulators prove to be fruitless. Emulation is an abstraction on top of the operating system (OS) where the emulator is running, and no direct interaction with hardware happens. Theoretically, the perfect emulator could spoof the logic behind any system call. Nonetheless, kernels like Windows NT incorporate a massive amount of functionality, yielding the implausible achievement of one-to-one replicas. Hence real-world emulators implement only a subset of all possible kernel manipulations [33].

While Speakeasy focuses on malware emulation and primarily emphasizes the replica of kernel logic most often used by malware, we commonly encounter API calls that the emulator does not support during the tests. By default, if the specimen refers to an unknown API call, emulation stops with an error message, documenting all previous calls. Therefore, we modify the default functionality of the emulator and add a dummy rule to return Windows failure code if an API call is missing, rather than halting the emulation process right away. Usually, this leads to execution errors at some point, like invalid memory read or write. However, such behavior is still more

beneficial for our needs if compared to default, since in some cases reveals additional functionality of specimen, valuable for ML model. In addition to that, we enrich Speakeasy functionality with anti-anti-debugging techniques to succeed in emulation of self-defending malware, with in-depth details discussed in Section 4.1.1.

3. Machine Learning in Malware Classification

Traditional anti-virus products relied on signature-based detections, which explicitly filtered out known malware samples based on various heuristics like file hash, representative string or byte patterns, and counting [9]. The concept of an anti-virus solution nowadays is extended to the endpoint detection and response (EDR) agent, which incorporates both preventive and *post factum* signature and behavioral-based analysis happening on a host and simultaneously at the security vendor's backend in a cloud. Karantzas and Patsakis [24] perform an exceptional review of the strengths and limitations of modern EDR solutions.

EDR solutions utilize sophisticated heuristics to achieve efficient defenses against previously unseen malware or even previously unknown 0-day[†] attacks. One way to solve this problem is to introduce an ML algorithm that learns common patterns across a vast dataset of known malware, obtaining a predictive power to classify malicious samples. Since the idea of malware detection using ML techniques was introduced by Schultz et al. [44], academia has produced a vast amount of research in this direction. We discuss adopted ML techniques for static analysis in Section 3.1, respective approaches for dynamic evaluation in Section 3.2, with limitations of modern solutions in Section 3.3.

3.1 Static Classifiers

Research in the utilization of ML techniques for malware classification follows conventional malware analysis taxonomy covered in Section 2.2. The prevailing part of ML-influenced malware classification is done in a domain of static analysis - ML heuristics are applied to representations acquired from static properties of malware files.

[†] https://en.wikipedia.org/wiki/Zero-day_(computing)

3.1.1 Gradient-Boosted Decision Tree Models

Notable success in malware classification through machine learning methods is achieved by a Gradient Boosted Decision Tree (GBDT) algorithm, specifically the approach introduced by Anderson and Roth [5]. Their work originally is meant to provide a benchmark dataset based on specific, pre-extracted properties from malware files, thus the title: *EMBER* (Endgame Malware BEnchmark for Research). At the same time, the paper includes an evaluation of LightGBM [25] model performance, describing an approach that employs a clever feature engineering phase. EMBER representations incorporate domain knowledge into many effective static characteristics of PE files, therefore, becoming a *de facto* standard for feature extraction from PE files in modern malware classification research.

3.1.2 Deep Neural Network Models

One of the first neural network applications for malware classification was presented by Raff et al. [40] who presented a *MalConv* model - a "featureless" Deep Neural Network (DNN) that reads raw bytes of an executable and proceeds with embeddings and 1-D convolution. MalConv can successfully learn patterns of both malicious and benign file structures, reaching Area-Under-the-Curve (AUC) values as high as 98.5% on test from the original paper [40].

An interesting hybrid approach is presented by Rudd et al. [42]. It utilizes EMBER representation vectors to train a feed-forward neural network (FFNN). They achieve considerable success by enlightening malware classification with explainability by SMART tagging [16], discussion of which is, however, out of the scope of this work.

3.2 Dynamic Classifiers

Static analysis techniques are often published with pre-trained models [5, 40, 42]. Hence solutions like Ember GBDT or MalConv are often referred to as benchmark models in the follow-up studies. Conversely, we are not aware of publicly shared dynamic analysis models available for a direct comparison. Moreover, a noticeable challenge in research is that practically all security vendors use dynamic analysis ML-powered malware classifiers in a proprietary manner without the ability to evaluate their methods.

Generally, dynamic PE analysis methods use API call telemetry to represent PE activity. Rosenberg et al. [41] construct a one-hot encoded vector out of encountered API calls. This approach is the simplest possible and ignores API sequences. Kolosnjaji et al. [28] showed how it is possible to perform API call processing to preserve

sequential information. Another example is described by Yen et al. [52], who construct a behavioral representation based on API call frequency.

Encoded API vectors are processed either by a one dimensional convolution operation [28, 41] or passed directly to a recurrent neural architecture [28, 52]. Convolutional layers act as feature extraction steps that correspond to learning n-grams essential for classification through a back-propagation. Recurrent neural network cells can learn sequential structure either by observing API calls directly or by working with already abstracted n-grams. Based on reported results, such models are well capable of learning representations of training data distribution, achieving accuracies close to 90% in multinomial classification [28, 41].

3.2.1 Classifiers Based on Emulated Malware

Utilization of emulators as a telemetry source of ML model for dynamic malware analysis research is not commonly adopted. Nevertheless, we encounter work done in this direction by a group of researchers from the University of California and Microsoft Research.

To our knowledge, the first occurrence of emulator utilization for system call collection was reported by Athiwaratkun and Stokes in 2017 [6]. Their model is influenced by Pascanu et al. [36] and resembles recurrent schemes used in Natural Language Processing (NLP). This work is further developed by Agrawal et al. [2] who present similar architecture but adopted for arbitrary long API call sequences acquired with the help of an emulator. The same group of researchers further discuss applying emulation as a platform for malware classification in a more narrow domain with a focus on emulated ransomware by adopting an enhanced neural cell [3].

3.3 Limitations of Modern Solutions

Perform mostly static property analysis. Since existing state-of-the-art models focus only on static properties of binary files, contemporary adversarial attacks are successful with only basic manipulations that exploit this property. Section manipulations like inserting new or renaming existing ones directly affect Ember feature engineering*. Adding overlay bytes alters entropy, byte histogram, and strings within the executable, affecting Ember GBDT and MalConv input vector characteristics.

As a result, attacks like GAMMA [14], or MAB-framework [45] covered in Section 6.3, are already effective just with limited action space, without the need for further sophistication of manipulations. Therefore, to develop potential techniques

*https://github.com/elastic/ember/blob/master/ember/features.py#L123

that real-world adversaries might use to extend existing adversarial attacks for artificial intelligence (AI) powered malware generation, we should strive to improve existing classifiers.

Packed software. Packed images indeed have distinct static characteristics like high entropy and uniform byte histogram. Therefore, it is logical to assume that the static classifiers can identify packed binaries. However, benign software is often packed too, for instance, to protect the intellectual property of its authors.

Packed sample	MalConv score	Ember GBDT score
calc.exe	0.9988	0.9607
brbbot.exe	0.9989	0.9999

Table 3.1: Scores of state-of-the-art ML classifiers on two binaries packed by UPX - legitimate Microsoft Calculator application and specimen of BRB Bot malware.

Albeit for static classifiers, all packed software results in a similar representation vector, since only essential kernel API calls and instructions are available during static analysis. All the application logic and data structures like embedded strings and resources are decoded during runtime. Custom packers used by malware hide malicious logic using a simple XOR operation or cryptographic functions supported by the Windows kernel.

Ignore contextual information. None of the models above consider any complementary information of compromise life-cycle. Indeed, both static properties of PE files and API call sequence are powerful information sources yet still are focused on the PE sample as a separate entity, without its environmental impact. Thus said, dynamic analysis telemetry is more heterogeneous, with behavioral inspection tools allowing information on system processes, filepath, registry, and network connections.

For example, trojan binaries patch legitimate software with a minor portion of malicious logic, resulting in almost the same static property vector and API call sequence. Consider malevolent code hiding by patching commonly used activation software `7z.exe`. Enrichment of decision heuristic with additional data structures like location on the filesystem is highly valuable to reveal the true nature of software. For example, the path would allow inferring whether it was executed from directories only accessible by the Administrator or placed in a world-writable location where such an executable file should never be present:

```
C:\Program Files(x86)\7zip\7zip.exe
C:\Users\myuser\AppData\Temp\7zip.exe
```

Security specialists raise suspicion if observing an application with a latter filepath without any analysis of the actual file. Therefore, considering only PE as an information source, we ignore a significant portion of information about the ecosystem.

Moreover, security operation centers (SOC) often do not collect raw PE files and API call sequences during infrastructure monitoring due to their low-level nature and high volume. Instead, more abstracted data from tools like Sysmon* is collected. For instance, process creation chains are reported by Sysmon's Event ID 1, all information about network connections by Event ID 3, filesystem and registry manipulations by Event IDs 11-14, and so on. By incorporating contextual information, dynamic malware analysis solutions thus can be beneficial not only for security vendors but for security operations too.

*https://docs.microsoft.com/en-us/sysinternals/downloads/sysmon

4. Dataset

The functionality of a hybrid solution presented in this work is based on input data consisting of both (a) armed PE files suitable for dynamic analysis **and** (b) contextual filepath information. The necessity to acquire contextual data expresses challenges in the utilization of public datasets since for *every* data sample we need to possess both raw PE bytes *and* filepath data on an in-the-wild system.

If we focus only on raw PE collections - the existing research landscape has multiple benchmark datasets, yet none of them is suitable for dynamic analysis. As already mentioned in Section 3.1.1, Ember is a vector dataset published by Anderson and Roth [5]. Vectors represent engineered representations based on the properties of PE files. Hence, we cannot utilize the Ember dataset for our experiments since we rely on PE file emulation, which requires functional binaries.

SOREL-20M [21] dataset represent a 8TB of compressed, but disarmed PE samples. Deactivation is done to prevent the accidental execution of malicious files. However, due to the nature of file modifications, namely `OptionalHeader.Subsystem` flag and the `FileHeader.Machine` header values were both set to 0. It is impossible to "re-arm" samples provided within the dataset, hence this collection is not suitable for dynamic analysis purposes. According to SOREL-20M authors, there is a need to fetch functional samples from other sources like VirusTotal[†] or ReversingLabs [‡] based on the hash value of the samples.

Additionally, to the best of our knowledge, none of the public datasets provide any contextual information about PE samples, like process chain information or filepath values at the moment of execution. For example, Kyadige et al. [31] use proprietary Sophos' threat intelligence feed without the public release of their dataset. The private nature of contextual data is understandable since such telemetry would contain sensitive components, like directory structure on personal computers.

Moreover, the filepath input cannot be anonymized without the effect on evaluation of model's real world utility. It has to be collected from a specific in-the-wild system, representing the real-world state of a particular PE execution. Therefore we

[†] https://www.virustotal.com/
[‡] https://www.reversinglabs.com/

map malicious file hashes to the real-world paths based on the threat intelligence of an undisclosed security partner, and, similarly to Sophos' research, because of reasons described in the partner's privacy policy, we cannot release the filepath dataset publicly.

4.1 Portable Executable Data

We collect the dataset in two sessions. The first session forms the foundation of our analysis, consisting of **98 966** samples, `329 GB` of raw PE bytes. 80% of this corpus is used as a fixed training set, and 20% form an in-sample validation set. We pretrain models and investigate our hybrid solution configuration using this data, with a detailed description of the experimental setup in Section 5.1.

The second dataset acquisition session occured *three months later*, forming an out-of-sample test set from **27 500** samples, about `100 GB` of data. This corpus is used to evaluate the real-world utility of the hybrid model and investigate model behavior on the evolved threat landscape.

The PE files in the dataset are tagged by a professional threat intelligence team, utilizing manual and automated reverse engineering tools with the help of malware analysts. The dataset spans seven malware families and benignware, with detailed distribution parameters described in Table 4.1. All labels except *Clean* represent malicious files. Therefore, we collected relatively more clean samples to balance malicious and clean labels in the dataset.

	Training & validation sets		Test set	
File label	**Size (Gb)**	**Counts**	**Size (Gb)**	**Counts**
Backdoor	30	11089	7.4	2500
Clean	127	26061	47	10000
Coinminer	46	10044	11	2500
Dropper	36	11275	9	2500
Keylogger	34	7817	9.8	2500
Ransomware	14	10014	4.6	2500
RAT[a]	5.5	9537	2.5	2500
Trojan	40	13128	7.1	2500
Total	329	98966	98	27.5k

Table 4.1: Dataset structure and size.

[a]Remote Access Tool

Since most malware is compiled as x86 binaries, we focus on 32-bit (x86) images

and deliberately skip collection of 64-bit (x64) images to maintain homogeneity and label balance of the dataset. Furthermore, malware authors prefer x86 binaries because of Microsoft backward compatibility, which allows to execution of 32-bit binaries on a 64-bit system, but not vice versa.

The dataset is formed out of executables (`exe`), and we intentionally omit library PE files (`dll`). As discussed in Section 2.1, both types of PE are practically the same, so our analysis results should highly correlate with DLL files. Given the desire to generalize this approach for a production environment, a fraction of DLLs should be included in the training process to spot distinct properties of both benign and malicious library files, like names of exported methods. Nonetheless, the involvement of DLL files is not needed for our hypothesis evaluation.

The dataset consists only of native binaries (compiled from `C, C++, Go`), but not .NET assemblies since the .NET framework was not supported by the emulator (Speakeasy v1.5.9) at the time of our experiments. Moreover, it is possible to produce lossless decompilation of .NET assemblies (using tools like `ILSpy`*) due to the nature of the .NET framework's Common Language Runtime (CLR)†. Consequently, more precise techniques can be used instead of emulation since all code level functionality and potential IoCs are accessible in decompiled source code.

4.1.1 Sample emulation

Emulated dataset. All the samples represented in Table 4.1 were processed with Speakeasy Windows kernel emulator [33]. Unfortunately, some of the sample emulations were erroneous, primarily due to an invalid memory read of write assembly instructions. However, another common reason for emulation errors is a call of unsupported API function or anti-debugging techniques. The error rate across different malware families is visualized in Figure 4.1.

We were able to successfully obtain behavioral patterns of **90857** samples from training and validation sets and **17347** samples in the test set. This corpus forms a significantly larger dataset than in related work on dynamic analysis. For instance, Kolosnjaji et al. [28] use a dataset of 4753 samples, Yen et al. [52] use 4519 samples. Relatively large size is a direct consequence of the emulation computational efficiency in comparison to the detonation of malicious samples in a virtualized environment. Emulation allows acquiring dynamic analysis telemetry fast, in a matter of a dozen seconds per sample, whereas virtualization requires hypervisor and operating system bootup before execution and cleanup after.

*https://github.com/icsharpcode/ILSpy
†https://docs.microsoft.com/en-us/dotnet/standard/clr

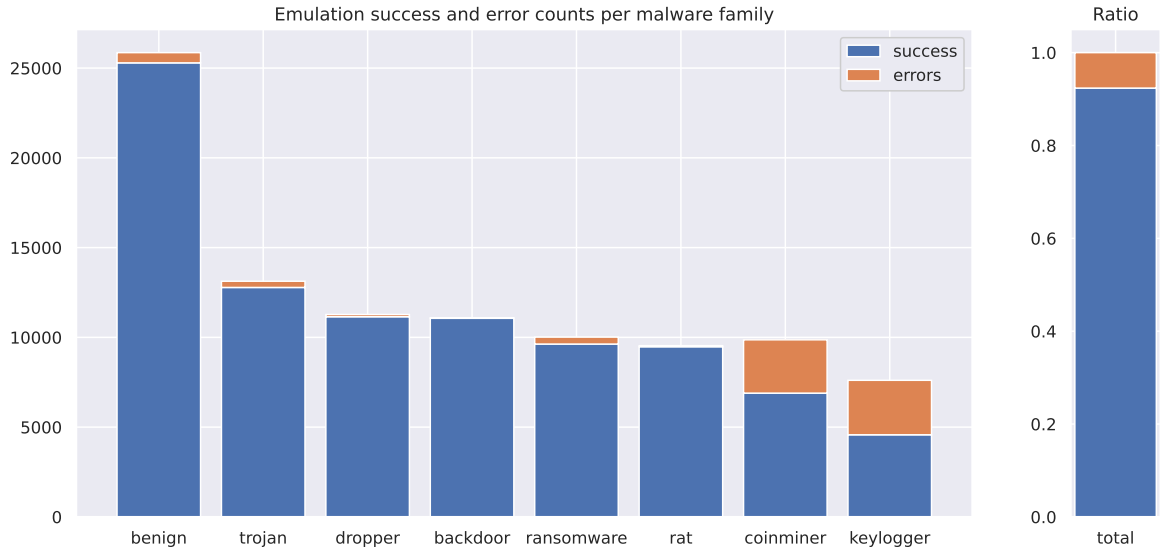

Figure 4.1: PE emulation error distribution across malware families in training and validation sets.

Speculatively, one of the potential drawbacks of the emulated dataset might be its relative sparsity if compared to the live execution of samples in a sandbox. However, empirical evidence shows that our dataset is more diverse than reported by other groups performing similar data acquisition using full Windows system virtualization. For instance, we acquire **2822** unique API calls within training and validation set reports, with the frequency graph represented in Figure 4.2. This behavior is significantly more heterogeneous than in related work datasets: Athiwaratkun and Stokes [6] have a total of 114 unique API calls, Kolosnjaji et al. [28] report 60 unique API calls, Yen et al. [52] have 286 different API calls, Rosenberg et al. [41] work with 314 unique API calls.

Partially such observation can be described by the increased volume of our dataset since the number of unique calls has a positive correlation with the number of samples. Although we emphasize this as evidence of the efficiency of the emulation technique as equivalent to sandboxing for dynamic analysis purposes - emulation reports produce rich and diverse telemetry. To contribute towards reproducibility and improvement of this and similar research, we release emulation reports publicly, which are available in JSON format within this work's repository*.

Anti-Debugging Techniques. As mentioned earlier in Section 2.2, malware authors include functionality that tends to confuse the analysis process. Obfuscation allows to slow down the identification of successful exploitation and indicators of compromise disclosure. Techniques like packers and code obfuscation are widely used to avoid static analysis. However, anti-debugging and emulation bypass functionality are often

*https://github.com/dtrizna/quo.vadis/blob/main/data/emulation.dataset/emulation.dataset.7z

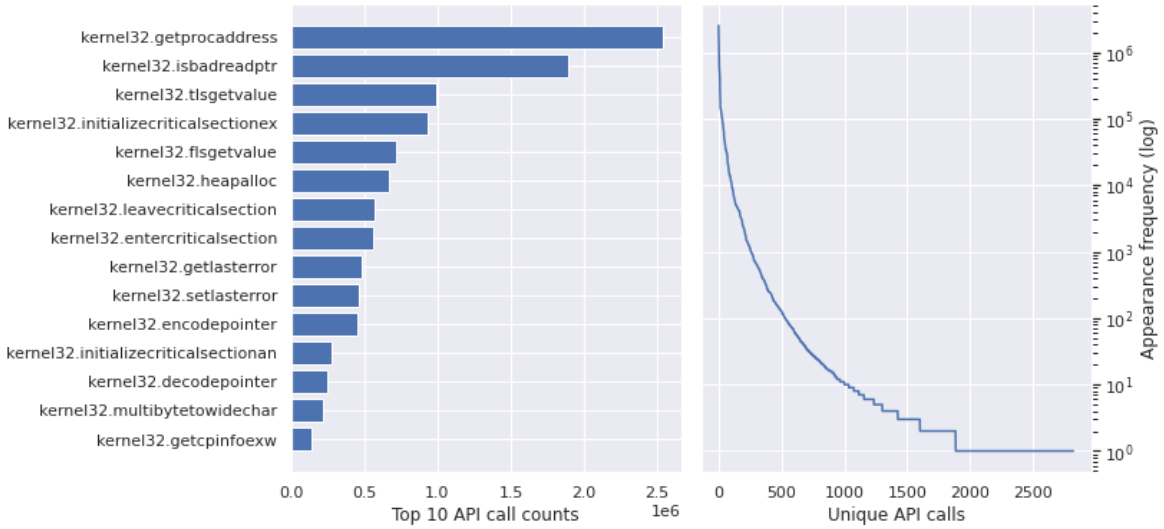

Figure 4.2: API call frequency statistics in training and validation sets.

inserted to suppress dynamic analysis attempts. For instance, rare kernel API calls are included to disrupt the process of emulation. As mentioned in Section 2.3.1, no emulation perfectly resembles the actual live system. Therefore, emulators fail to provide functionality for rare kernel methods.

In our analysis we faced unusual API calls like `GetCaretBlinkTime`, `GetClipboardViewer`, `GetMessageTime` called at the very beginning of PE execution. None of these are essential to malware functionality, yet break emulation or reveal debugging environment, preventing further execution of malicious logic at the very start of the program without proceeding to more representative API calls.

During manual analysis, reverse engineers can spot and combat anti-debugging techniques either manually or with tools like ScyllaHide*. We have analyzed frequent anti-debugging techniques in our dataset and implemented multiple hooks to overcome common emulation and sandbox enumeration attempts. Since the same anti-emulation techniques might be a challenge for other emulator users and researchers, we contributed changes to the emulator code†, and they are now merged to the main branch. Example of added API hook in Speakeasy [33] emulator:

```
@apihook("GetCaretBlinkTime", argc=0)
def GetCaretBlinkTime(self, emu, argv, ctx={}):
    return 550 # ms
```

Because of the large corpus, it is impossible to implement workarounds on all anti-emulation techniques. However, a short sequence of API calls used during emulation

*https://github.com/x64dbg/ScyllaHide
†https://github.com/mandiant/speakeasy/pull/194

analysis still can be considered representative of malware and valuable to the ML model. For example, observing a sequence of two API calls like `ShowConsoleCursor, GetUserDefaultUILanguage` with consequent termination of the program is enough to spot a self-defending malware that enumerates the sandbox environment. Even if an emulator cannot fool the malware, this information provides enough evidence for the classifier to assign a high predictive score.

4.2 Filepath Data

The filepath model was trained on extended dataset to yield better generalization against unknown threats and potentially better robustness against out-of-sample test set. The full filepath dataset consists of the following data sources:

- Threat intelligence telemetry based on in-the-wild paths of collected PE samples mentioned in Table 4.1. Paths represent a file location at the moment of PE execution. After unique filepath aggregation, this feed contributes 41934 malicious and 49380 benign paths. PE in-sample validation set and out-of-sample test set paths are excluded from training routine.
- Corporate environment network share listing representing paths of common documents and instrumental files of running a business - 422444 benign paths.
- Augmented dataset representing known techniques to store and execute malware samples, discussed in detail below within Section 4.2.1. Augmentation contributed to 224280 malicious paths.
- Regular Windows 10 filesystem with all predefined Microsoft paths and multiple legitimate applications installed - 122409 benign paths.

Total size of path dataset is about `450 MB` of textual data, consisting of **1126764** entries, with 34.74% representing a malicious file locations, and 65.26% legitimate paths.

Worth noting that the collection of clean Windows filesystem was done with built-in tools. In particular, a recursive listing of all files on a `C:` drive of a Windows system can be obtained using the following PowerShell command:

```
Get-ChildItem C:\ -Recurse | Out-File fs.raw
```

4.2.1 Data Augmentation

Our data augmentation logic is released publicly and can be observed in the accompanied repository*. Given the need to replicate experiments, it is possible to utilize or even extend this approach.

We considered multiple path generation heuristics known to be representative of malicious samples, for instance, misplaced DLL files. Such files can lead to DLL search order hijacking, providing either persistence or privilege escalation. This technique is highly efficient and is poorly detected by modern EDR solutions [24]. It is known to be used during malicious operation *Groundbait* observed in Ukraine for strategic surveillance purposes [12]. To mimic this behavior, we replicated DLL files from `System32` directory in other common DLL search locations†: Windows directory, application's current directory, directories usually listed in `PATH` environment variable.

Similar method, but in relation to `exe` files is often used for persistence given the administrative rights - executable files from `System32` directory are commonly misplaced in other folders, therefore, we applied the same augmentation technique for `exe` files as well. To give an instance, UBoatRAT‡ malware used `c:\programdata\svchost.exe` path for persistent BITS job.

One more pattern typical for malicious files and frequently observed as initial execution vector utilized by Advanced Persistence Threat (APT) groups is Windows built-in scripting engine usage, with file extensions like `.hta, .bat, .vbs, .vbe, .js` under commonly used user directories, for instance, `Downloads` folder. According to MITRE, such pattern is representative to more than a few dozen APT groups§. Another examples of augmented filepaths are: `exe` and `dll` files under hidden and temporary user directories, Microsoft Office files with macros, and so on.

*https://github.com/dtrizna/quo.vadis/blob/main/data/path.dataset/augment/augmentation.ipynb

†https://docs.microsoft.com/en-us/windows/win32/dlls/dynamic-link-library-search-order

‡https://attack.mitre.org/software/S0333/

§https://attack.mitre.org/techniques/T1059/

5. Hybrid Model for Malware Classification

As discussed in Section 3.3, limitations of current solutions are a subject of restricted visibility due to the nature of the static analysis. Therefore, extending existing models with techniques influenced by dynamic malware analysis would provide additional epistemic value to overcome the shortcomings of solutions based only on static analysis methods.

A common way to inspect the results of dynamic malware analysis is a collection of verbose system telemetry that represents software manipulations with an operating system. Such telemetry comes in form of operating system and application logs, covering system behavior without the need to collect actual executables in any central repository. Such telemetry from live systems acts as the principal visibility source in security operation centers (SOC), often without any option to perform enterprise-wide static property analysis of executable files itself.

The behavior of applications is represented in various data formats, including process creation chains, filesystem or registry read/write operations, and other information distinctive for dynamic malware analysis. Using such data as input of ML algorithm for classification is overlooked in malware detection research, and we hypothesize that it requires multiple parallel models. We focus on filepath data to incorporate system logging as part of our decision heuristic, with the subject of potentially expanding such approach with complementary modules that utilize other logging sources.

We acknowledge that filepaths alone do not provide a necessary epistemic capacity to explicitly classify a file as malicious or benign. Paths, however, often give insights on the "fitness" of a file to a usual operation of the operating system. As stated above, with a specific example in Section 3.3, experienced malware analysts can spot malicious files based on their path with high probabilities. Moreover, existing research shows that filepath data can improve static PE detection analysis [31].

We extend hybrid model's visibility by utilization of Windows kernel emulation. We consider a sequence of Windows kernel API calls provided by the emulator as a primary source of PE behavior on the system. Therefore, the input of our hybrid

solution is based on (a) file path data acquired from in-the-wild systems and (b) raw PE bytes for analysis by state-of-the-art models, where a latter component is passed through an emulator for additional pre-processing. A general overview of the hybrid model architecture is visualized in Figure 5.1.

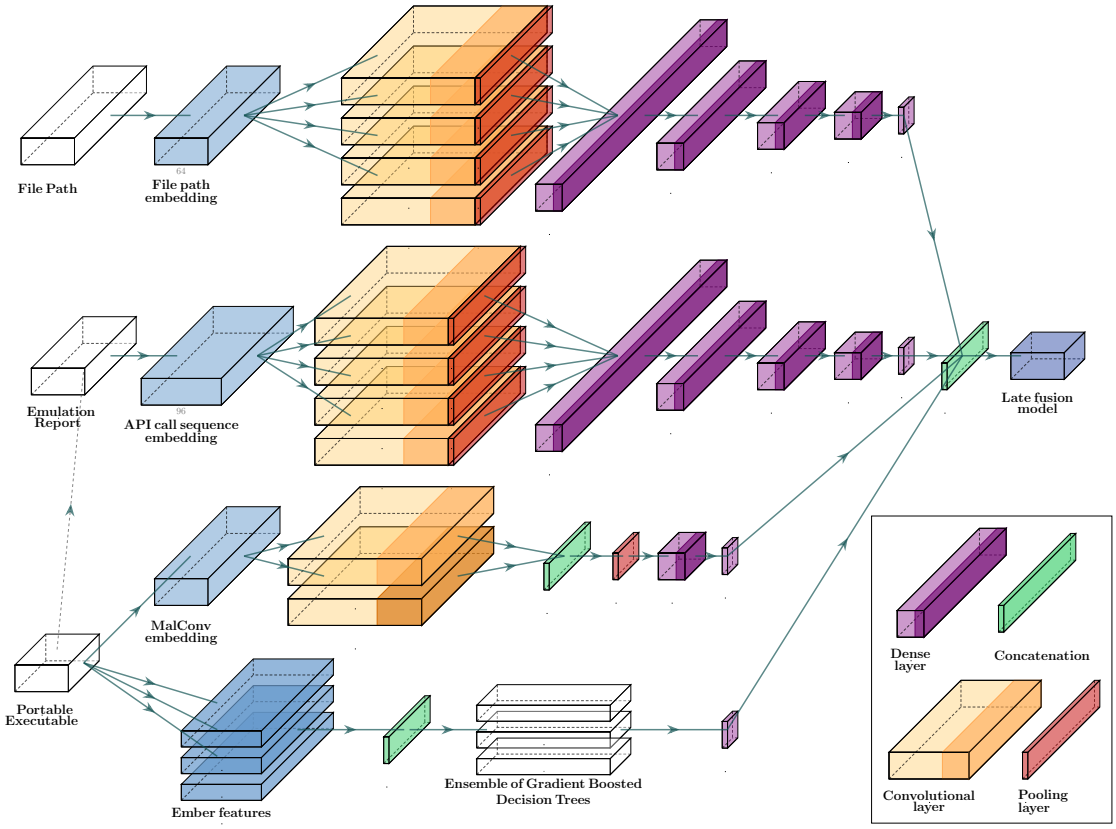

Figure 5.1: General view of hybrid model architecture representing four separate early fusion networks: (1) filepath 1D convolutional network, (2) emulated API call 1D convolutional network, (3) MalConv [40] model, and (4) Ember GBDT [5] model, which are processed by late fusion model for final decision.

Input is passed through an **early fusion models** ϕ, which are represented as four separate models:

- file-path 1D convolutional neural network, ϕ_{fp}
- emulated API call 1D convolutional neural network, ϕ_{api}
- MalConv neural network [40], ϕ_{mc}
- Ember GBDT model [5], ϕ_{emb}

Each early fusion module produces a scalar with an estimate of its input maliciousness. Every module's output is concatenated to a 4-dimensional vector. Therefore,

given an input sample x, consisting of raw PE as bytes and its filepath as string, early fusion pass collectively is denoted as:

$$\phi(x) = [\phi_{fp}, \phi_{api}, \phi_{mc}, \phi_{emb}] \in [0,1]^4.$$

Intermediate vector $\phi(x)$ is passed to a **late fusion model** ψ, which produces final prediction:

$$\hat{y} = \psi(\phi(x)) \in [0,1]. \tag{5.1}$$

Trainable early fusion models (file path and emulated API call sequence networks) are fitted independently of hybrid solution using binary cross-entropy loss function:

$$L(x, y; \theta) = -y \log(\phi(x; \theta)) + (1 - y) \log(1 - \phi(x; \theta)),$$

where $\phi(x; \theta)$ denotes function approximated by deep learning model given parameters θ, and $y \in \{0, 1\}$ are a ground-truth labels. We use pre-trained MalConv and Ember GBDT parameters on Ember dataset [5] and released in 2019 by Endgame*. We deliberately avoid retraining of MalConv and Ember GBDT to perform a direct comparison of attack evasion rates [14, 45] and originally reported performance [5].

It is worth noting that using original MalConv and Ember GBDT weights allows us to draw several important conclusions discussed in Chapter 9. We are conscious of the bias in the file path and emulation module performance compared directly with Ember GBDT since file path and emulation models were trained on data from the same distribution as the validation set and test sets. We emphasize that our goal is not a direct comparison of each module's performance but (1) an analysis of whether static and dynamic techniques complement each other and (2) the potential for improving original model.

5.1 Experimental Setup

As outlined in Section 4, filepath and emulated API call model pre-training is done on the fixed 80% of PE samples collected in the first session. Optimization is performed using Adam optimizer [26] with 0.001 learning rate and fixed batch size of 1024 samples. We constructed both networks and training routines using PyTorch [37] deep learning library.

Configuration analysis of these models is a subject of hyperparameter grid search process. All settings are fixed except for a single testable hyperparameter. Performance assessment is done on the in-sample validation set, formed from fixed 20% of data. Based on empirical observation of model convergence in our experiments, it was

*https://github.com/endgameinc/malware_evasion_competition

enough to train models for 20 epochs to evaluate *relative* hyperparameter efficiency. Furthermore, once the configuration of separate modules was identified, they were trained for 50 epochs and, collectively with pre-trained MalConv and Ember GBDT models, formed an early fusion pass of the hybrid solution.

The output of early fusion modules $\phi(x)$ is used to train the late fusion model ψ. Three simple architectures were evaluated, specifically: Logistic Regression (LR) classifier and a one layer FFNN with 15, 50, and 100 hidden neurons, implemented through `scikit-learn` library [39], as well as gradient boosted decision tree classifier based on `xgboost` [11] implementation, all models using default library configuration and training parameters.

The final utility evaluation of composite detection heuristic is done on a test set gathered three months later. Individual module and composite solution performance were estimated based on the F1-score and receiver operating characteristic (ROC) curve values. In a security domain, tolerance against False-Negatives is small. Hence analysis of a metric like an accuracy does not represent the real-world utility of the model.

Additionally, even though single value metrics like F1-score or area under the curve (AUC) take false positive and false negative rates into account, we argue that these are too vague to represent a real-world utility of the security solution. Therefore, we provide a detailed analysis of detection rates with fixed false positive rates (FPR), which represent the ability to identify malicious actions with a predefined manual load passed to human analysis who work with model results.

5.2 Early Fusion Network Architecture

Analysis of file path and API call sequences can be formulated as a related optimization problem, namely classification of 1-dimensional sequences. We utilize similar neural architecture, yet with distinct preprocessing and network configurations. The design of both deep learning classifiers is influenced by the model described by Kyadige et al. [31]. Encoded input vector x with fixed length N is provided to embedding layer with dimensions H and vocabulary size V. Selection of appropriate values is reported in Section 5.3 for file path model, and Section 5.4 for emulated API call sequence model.

The optimal values for file path model obtained by hyperparameter optimization on validation set are: input vector x_{fp} length $N = 100$, embedding dimension $H = 64$, vocabulary size $V = 150$. Respective values for emulated API call sequence model are: input vector x_{em} length $N = 150$, embedding dimension $H = 96$, and vocabulary size $V = 600$. The vocabulary of the file path model is formed out of the most common UTF-8 bytes and for API call sequences model the most common system calls are selected. Both vocabularies are enriched with two labels used for padding and rare

characters.

The output of the embedding layer is passed to four separate 1D convolution layers with kernel sizes of $2, 3, 4$, and 5 characters and the number of output channels $C = 128$. With lower C values model underperforms. For instance, having $C = 64$ file path's module validation set F1-score is as low as 0.962, while with $C \in \{100, 128, 160\}$ scores plateau around 0.966.

The output of all four convolution layers is concatenated to a vector of size $4 \times C$ and passed to a feed-forward neural network (FFNN) with four hidden layers holding 1024, 512, 256, and 128 neurons. Hidden layers of FFNN are activated using rectified linear unit (ReLU) [1]. The final layer uses a sigmoid activation. Batch normalization [23] is applied to hidden layers of FFNN before the ReLU activation. Additionally, to prevent overfitting, dropout [47] with a $P = 0.5$ rate is applied. The architecture is visualized in Figure 5.2.

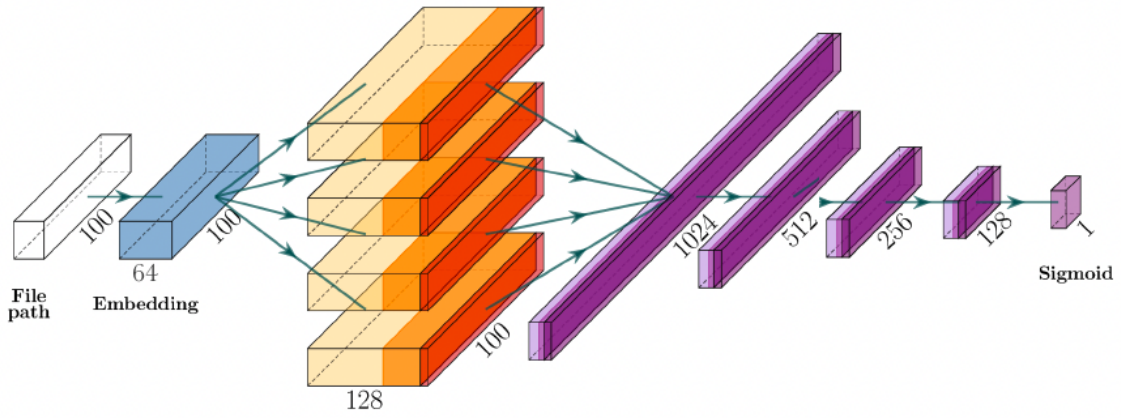

Figure 5.2: Early fusion network's architecture (representing file path module configuration). Network embeds UTF-8 encoded input vector in 64-dimensional vectorspace, then uses 4 parallel 1D convolution layers with different kernel sizes (2,3,4,5), which are flattened and passed to FFNN.

Our tests around FFNN architecture revealed that four layers with $1024, 512, 256$, and 128 hidden neurons are the optimal choice. Deeper architectures have noticeable overfitting with occasional decreases in validation set metrics. On the contrary, single, two, or three-layer FFNN slightly underperforms, having a worse validation F1 score and higher training and validation losses.

5.3 Filepath Module

This section covers the filepath classification module. We first discuss manipulations done during the path preprocessing phase, like normalization and encoding. Then

we review the experimental setup including model architecture and training routine. Finally, performance with different hyperparameter choices is reported, including suggestions on preferred network configuration.

5.3.1 Path Preprocessing

Path normalization. The first part of data preprocessing included path normalization since some parts of filepath semantics have variability that is not relevant for security analysis through the deep learning model. These include specific drive letter or network location if a universal naming convention (UNC) format is used, as well as individual usernames, so during normalization, we introduced universal placeholders for those path components as seen below:

```
[drive]\users\[user]\desktop\04-ca\8853.vbs
[drive]\users\[user]\appdata\local\file.tmp
[net]\company\priv\timesheets\april2021.xlsm
```

Additionally, we needed to parse Windows environment variables to resemble actual filepath rather than environment alias used as variable name. Therefore, we built a variable map consisting of about 30 environment variables that represent specific paths on a system and are used across contemporary and legacy Windows systems, with few examples below:

```
r"%systemdrive%": r"[drive]",
r"%systemroot%": r"[drive]\windows",
r"%userprofile%": r"[drive]\users\[user]"
...
```

As discussed later in Section 5.2, our model performs character-level convolutions, hence we do not apply any tokenization techniques. On the contrary, we perform encoding of unique letters against the UTF-8 character set. A similar approach was used by Saxe et al. [43] when evaluating URL maliciousness, and by Kyadige et al. [31] on a filepath data, using 100 and 150 most frequent UTF-8 bytes respectively. Rare characters below a frequency threshold are discarded and replaced by a single dedicated label. The frequency distribution of UTF-8 characters in our dataset case be observed in Figure 5.3, and the obvious choice is to preserve 150 most common bytes based on frequency data.

Input vector length. The next steps include path truncation or padding of paths to a specific length, where padding is done with a dedicated 'pad' label. Having longer path vectors result in slower training and higher memory consumption by dataset, but allows for preserving information from the longest paths. For observability purposes, the distribution of path lengths in our dataset is described in Table 5.1. The choice of appropriate path vector length is performed during hyperparameter selection.

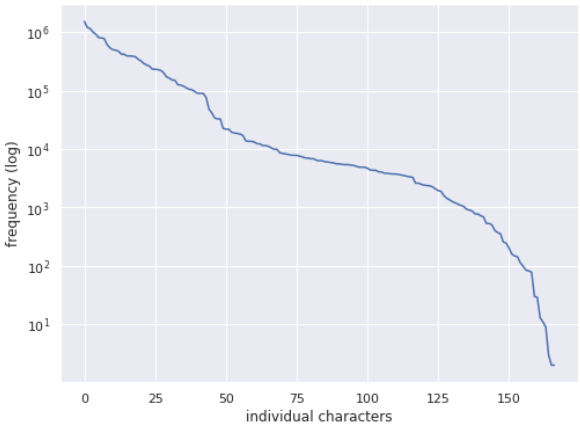

Figure 5.3: Frequency distribution of unique UTF-8 characters in path dataset. Note: y-scale is logarithmic.

We implement two approaches to perform a filepaths truncation. It is possible to discard the beginning of the path and keep only the last N characters. Alternatively, the more sophisticated technique allows abandoning the middle part, with a rationale to preserve the initial filesystem folder structure, and the filename with its extension. Both filename parts are important for security evaluation since the folder holds an epistemic value of the directory is a system or user-related, and filename and extension define file type and origin.

Path Length	150	100	75	50	25
Percentage	98.04	84.53	67.60	31.64	7.43

Table 5.1: Proportion of entries in dataset below defined path length.

Surprisingly, we did not identify any noticeable difference between the truncation types. As a borderline case, we held an experiment on input vector X_{FP} length of 50 symbols. Based on Table 5.1, only 31.64% of paths are shorter, therefore, about 70% of entries were truncated. However, whether we perform truncation in the middle of the string, or cut the beginning, had not affected the results, having validation set F1-scores 0.9665 and 0.9666 respectively.

On the contrary, our tests showed that the length of the input vector X_{FP} has important implications on the model's performance and results. Figure 5.4 displays that cutting vector lengths as short as 25 symbols severely reduces model efficiency, but vector lengths of 75 and 100 characters produce scores comparable with experiments when using longer sequences. The increase of input vectors has linear dependency on training time, therefore, we see a tradeoff during a choice of X_{FP} length.

Vocabulary size. Training iterations with vocabulary sizes $V \in \{50, 75\}$ yielded similar efficiency to the model proposed by Kyadige et al. [31] where $V = 150$. Worth noting, however, that $V \in \{100, 150\}$ requires similar training time, with a noticeable jump up happening only if more than 150 UTF-8 characters are used. Therefore, while the model does not lose much epistemic value out of data with $V \in [50, 100]$, we do not overload the model or observe overfitting using $V \in [100, 150]$, therefore we suggest using $V \approx 100 \pm 50$ depending on byte distribution in the training set.

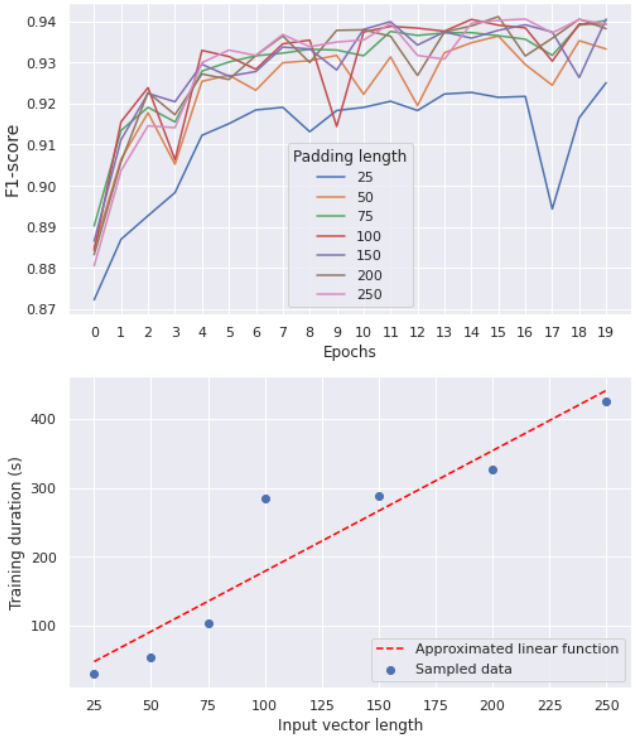

Figure 5.4: Validation set F1-score and training duration dependency on input vector length.

Embedding dimension. Results of experiments with embedding dimension H are summarized in Table 5.2. We see that the output dimension of embeddings has an important effect on model performance, with $H \in \{16, 24\}$ reporting reduced training and validation F1 scores. At the same time, we see evidence that using dimensions higher than 32, originally proposed by Kyadige et al. [31], might be beneficial. We notice that H as high as 48 or 64 allows achieving an increase in training set F1-scores by $0.3 - 0.5\%$ of F1-score on both training and validation sets. Worth noting that H increase linearly affects training duration, similarly to input vector length X_{FP}, therefore, a trade-off decision should be made. Expanding the embedding dimension further to $H \in [80, 96, 112]$ yields no improvements in validation metrics, however, increases computational cost as seen from the Epoch time column in Table 5.2. We suggest using $H \approx 64$ for production realizations.

5.4 Emulation Module

Passing PE samples through an emulator produce a JSON report about executable interaction with an operating system. It contains data on invoked API calls, manipulations with registry hives and filesystem objects, initiated network connections, in-

Embedding dimension	Training F1-score	Validation F1-score	Epoch time (s)
16	0.9774	0.9659	190
32	0.9794	0.9662	311
48	0.9800	0.9647	394
64	0.9804	**0.9679**	483
80	0.9807	0.9675	591
96	0.9809	0.9677	611
112	0.9809	0.9674	741

Table 5.2: Dependency of model performance from embedding dimension H. F1-scores are reported as mean across last 10 training epochs.

cluding parameters (for instance, HTTP request addresses). Emulation report dataset parameters like size, diversity, and success rate are described in Section 4.1.1.

We utilize only API call sequence as a representation of executable behavior on the system. However, we acknowledge that the epistemic capacity of the decision heuristic can be enriched by the utilization of complementary optics mentioned above, which is the subject of discussion of future work recommendations in Section 8.

5.4.1 API Call Preprocessing

API call sequence vector representation. To acquire a numeric value of API call sequences, we select the top most common calls based on variable vocabulary size V, whose evaluation is reported in this Section below. All API calls are label-encoded, and rare calls out of vocabulary replaced with a dedicated label. The final sequence is either truncated or padded using a dedicated label to a fixed length N.

Input vector length. Experiments around input vector length express the same pattern as observed in the file path module and visualized in Figure 5.4. Namely, linear dependency of training time and length, observing plateau of metrics with increased lengths. Nonetheless, we observe that slightly longer sequences than $N = 100$ are reasonable, probably because some of the crucial binary functionality may be revealed later in the code after the first 100 API calls are made. Therefore, we have chosen $N = 150$ for our final realization as an optimal trade-off between the model's computational demands and the ability to preserve the necessary amount of knowledge from a sequence. Yet longer vectors lengths might be beneficial, especially for real-world classifiers.

Embedding dimension. Similarly to previous hyperparameters, we observe that larger embedding dimension values (if compared to file path model) yield better results. However, we observe that after some threshold, an increase in embedding dimension results in worse performance. Dependency between embedding dimension and computational needs is reported in Figure 5.5. The graph clearly emphasizes that the model's efficiency is the best around $H = 96$.

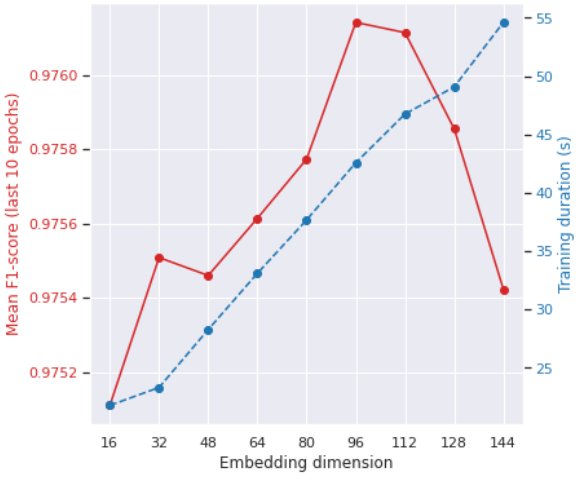

Figure 5.5: Performance and computational value of API call sequence module based on value of embedding dimension H.

Potentially, this can be explained by hypothesizing that at some point it is harder to follow convolution layers to find relevant combinations in increasingly high dimensional space, which no more contributes to better knowledge extraction, but a higher lookup domain.

Number of preserved API calls. The choice of vocabulary size V has a direct effect on the diversity of data the model receives. The higher the value of V, the larger dataset percentage is preserved (with the respect that less unique API calls are replaced by a single label representing rare calls). Beware that an unreasonably high V might bring noise, and result in overfitting of a model since rare calls will not be represented by a population that generalizes well to be a trustworthy information source for a model.

Top API calls	100	150	200	300	400	500	600	700
Dataset %	95.53	97.67	98.73	99.48	99.74	99.85	99.91	99.94
Val. F1-score	0.9707	0.9712	0.9725	0.9740	0.9752	0.9747	**0.9759**	0.9754

Table 5.3: Dataset diversity preserved and validation set's F1-score based on choice of top API calls.

Statistics behind dataset diversity and respective model performance are reported in Table 5.3. Experiments show even though preserving only 100 most common calls contain more than 95% API calls within a dataset, the model still benefits from relatively large vocabulary size values, so we have chosen $V = 600$ for our final configuration.

This observation might be explained by a distribution of API calls per sample,

where verbose executables with hundreds of calls bias call frequency, whereas executables with modest API sequences perform more unique function combinations. Model is still able to use such relatively rare sequences for correct sample classification having larger V sizes.

5.5 Fusing Hybrid Solution

As discussed at the beginning of this Chapter, we consider three late fusion model architectures:

- logistic regression (LR)
- ensemble of gradient boosted decision trees (GBDT)
- one layer feed-forward neural network (FFNN)

Model	AUC	F1-score	Recall	Precision	Accuracy	Training time
LR	0.9988	0.9877	0.9895	0.9859	0.9821	**0.12 s**
GBDT	**0.9990**	0.9890	**0.9908**	0.9872	0.9840	7.29 s
FFNN, 15	0.9988	0.9904	0.9894	0.9910	0.9861	2.81 s
FFNN, 50	0.9988	0.9905	0.9894	0.9916	**0.9862**	8.31 s
FFNN, 100	0.9988	**0.9906**	0.9877	**0.9919**	0.9853	13.00 s

Table 5.4: Validation set metrics when assessed against different late fusion model architectures. Integer after FFNN represent number of neurons in a hidden layer.

Our tests do not express a significant difference in late model performance with metrics reported in Table 5.4. Although the late fusion model ψ performs relatively simple non-linear mapping $[0,1]^4 \to [0,1]$, FFNN with gradient boosted decision trees report better results than logistic regression. The need for semi complex heuristic is because, in addition to weighting relative module prevalence, the late fusion model needs to select a correct threshold for optimal False-Positive, False-Negative, and detection rates. As a tradeoff choice, we selected an FFNN with a single layer and 15 hidden neurons as a late fusion model for our final evaluations since it has close to optimal scores but noticeably faster training time.

5.6 Hybrid Model Performance

Experiments show that simultaneous utilization of static and dynamic techniques enhances classification performance, allowing techniques to complement each other. This

observation can be inferred from experiments when various enabled module combinations are compared. Validation set detection error tradeoff (DET) and a receiver operating characteristic (ROC) curves are reported in Figure 5.6.

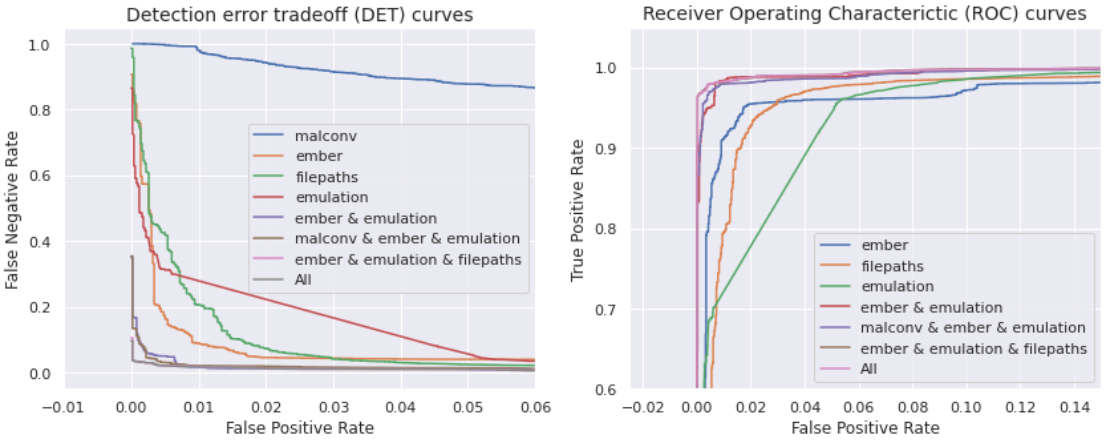

Figure 5.6: Validation set detection error tradeoff (DET) and receiver operating characteristic (ROC) curves based on different combinations of enabled modules in hybrid solution.

Each separate module except MalConv can achieve a high-quality classification capability, with an AUC score of Ember GBDT module 0.9860, filepath module 0.9886, emulated API call module 0.9872 (MalConv AUC is 0.6920. We separately discuss causality behind this phenomenon below in Section 5.6.1). Therefore, we exclude MalConv from further analysis and final decision heuristic since static property analysis is exceptionally well by the Ember GBDT module alone.

We observe a noticeable increase in detection abilities by combining different types of modules. A visible gap in DET and ROC curves appears when multiple modules are combined, with measurable significance, reported in Figure 5.7 which displays a detection rate with a fixed false positive rate (FPR).

Simultaneous utilization of static, dynamic, and contextual information provides an added epistemic value for ML-based decision heuristic since yield detection rates above the cumulative capabilities of individual modules. For instance, given a FPR of 0.02% Ember GBDT detects 14.14% samples, the filepath module identifies 4.23% malevolent paths, emulation module uncovers 27.76% of malicious API call chains. However, combining all three together model identifies 96.41% of malware samples, reporting more than 50% increase in detection rate if compared to all three modules together, but independently.

Additionally, *composite utilization of static, dynamic, and contextual data addresses independent method weaknesses, allowing to detect samples that can bypass a specific detection heuristic, minimizing false positive and false negative rates.* Figure 5.7 shows that simultaneous utilization of different malware analysis methodologies al-

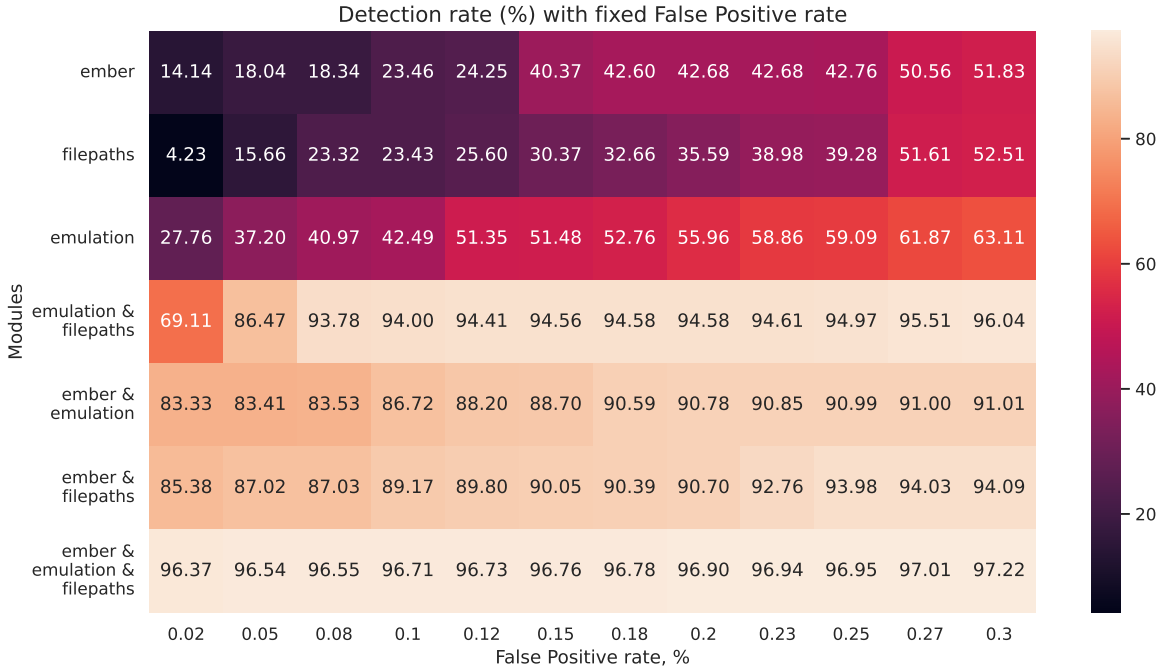

Figure 5.7: Detection rate (%) on validation set with fixed false positive rate (FPR) based on different combinations of enabled modules in hybrid solution.

lows achieving the same detection rate with a noticeably lower FPR. Given the hybrid employment of Ember feature engineering and API call sequences, it is possible to identify 66% of malware in a validation set with just two false positive cases in 10 000 samples. In contrast, the same detection rates for both models result in more than 30 false alerts. We separately outline hybrid solution implications on false negative rates in Appendix A on page 65.

We proceed with evaluation of composite solution's realistic utility based on a test set collected three months after initial data acquisition (used for module training and configuration analysis). F1-score, Precision, Recall, Accuracy, and AUC scores on all sets are reported in Table 5.5 with late fusion model decision threshold 0.95 that resembled a FPR $\approx$ 0.5% on validation set. Classifier's confusion matrix on out-of-sample test set can be observed in Figure 5.8. Out of all samples, `3.85%` are false negatives and `1.68%` are false positives (respective metrics on in-sample validation set are `0.91%` and `0.53%`).

5.6.1 Evolving Nature of Malevolent Techniques

Reported results on the train and validation sets allow us to conclude that model has little to no overfitting. However, we still observe a decrease of F1 and AUC scores by $\approx$ 4%, despite the same collection method and represented malware families and the relative frequency of samples within the classes. We cannot state the causality of this

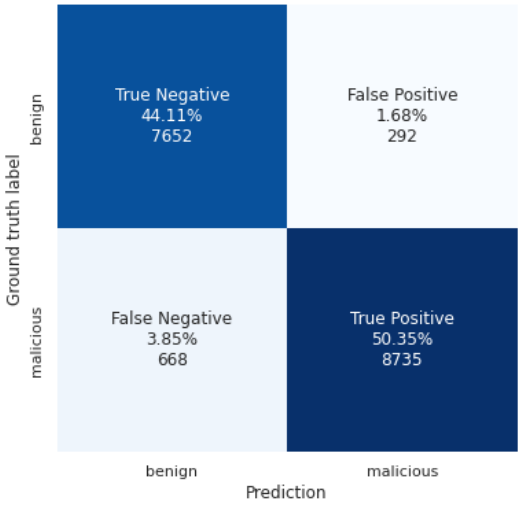

Figure 5.8: Test set confusion matrix representing model performance of hybrid solution with enabled Ember GBDT, API call sequence, and filepath modules using decision threshold of late fusion model 0.98.

Metric	Train set	Valid. set	Test set
F1-score	0.9979	0.9890	0.9479
Recall	0.9960	0.9810	0.9290
Precision	0.9999	0.9974	0.9677
Accuracy	0.9971	0.9842	0.9447
AUC	0.9979	0.9870	0.9461

Table 5.5: Hybrid solution final metrics on all sets with enabled Ember GBDT, API call sequence, and filepath modules using decision threshold of late fusion model 0.98.

phenomenon. However, we assume that it arises from evolving nature of malevolent logic. Additional evidence for this assumption comes from the weak detection abilities of the MalConv module. While our tests show poor MalConv metrics if compared to other modules, the AUC score reported by Anderson and Roth [5] was as high as 0.998, while our experiments yielded only 0.691. Additionally, we would like to emphasize that MalConv and Ember GBDT were trained on the same dataset.

Validation dataset	Year	MalConv, AUC	Ember GBDT, AUC
Anderson and Roth [5]	2017	0.99821	0.99911
Our validation set	2022	**0.69068**	0.98603

Table 5.6: Comparison of AUC scores as reported by Anderson and Roth [5] and our experiments.

Data used in the original paper was collected in 2017, whereas our training and validation sets come from real-world systems at the beginning of 2022. Offensive techniques are excessively volatile because of a never-ending cat-and-mouse game between defensive solutions and threat actors. Infiltration tactics that were successful five years ago are not practical today. Malware authors adapt and implement different techniques to accomplish their goals.

The drastic decrease in MalConv detection rates emphasizes a few facts:

- detection heuristic utility of manually engineered representations that involve domain knowledge is higher than statistical significance inferred by a neural net-

work;

- it is essential to perform continuous telemetry acquisition and periodic update of algorithm's parameters to maintain high-quality detection capabilities of contemporary threats.

6. Adversarial Attacks

Modern ML algorithms are a result of more than 50 years of research. Discriminative ML models discover probability distribution $p(y|x)$, which represents a mapping of some specific input data vector x, to an output vector y. Historically ML algorithms were developed assuming that environment and input data are benign during training and evaluation. However, recent advancements in adversarial input generation revealed a severe challenge for ML models [20].

Adversarial inputs are data samples generated by potential "adversaries" to affect the ML model's behavior [20]. Successful adversarial example (AE) contains carefully chosen information perturbations that result in a different ML algorithm's output y. It is important to emphasize that AE still needs to possess the original data class characteristics. AE could be a modified road sign that fools the classifier but appears the same for the human's eye or a malware sample that evades anti-virus software yet executes malicious logic.

Adversarial attacks are a rapidly growing field. Since the problem was independently revealed by Szegedy et al. [48], and Biggio et al. [8] in 2014, there have been around 5000 papers on adversarial attacks and defenses[†]. Several sensitive ML deployment domains, where adversaries are especially dangerous:

- Face recognition
- Autonomous vehicles
- Financial and trading algorithms
- Malware classification

Nowadays, the Artificial Intelligence (AI) research industry has acknowledged the problem of adversarial examples, and initiatives are present to systematically reduce potential damage to ML applications. Appearance of "Adversarial Machine Learning Threat Model ($ATLAS$[‡])" from MITRE in the same format as "ATT&CK Matrix for

[†] https://nicholas.carlini.com/writing/2019/all-adversarial-example-papers.html
[‡] https://atlas.mitre.org/

Enterprise"* widely attributed in conventional cyber-security problem space is one of notable examples.

6.1 White-Box attacks

When an adversary has full access to model parameters and architecture, we call such threat model as *white-box* [20]. The generation of adversarial examples in such a setting is similar to the ML algorithm's optimization problem. However, as opposed to cost minimization, an adversary needs to identify the most vulnerable cost regions with respect to input parameters. Finding specific parameter perturbations that shift input to high-cost regions eventually leads to source-target-misclassification.

Therefore, parameter optimization algorithms can be used for adversarial example generation as well. For instance, Kolosnjaj et al. [27] weaponize version of gradient descent. The general idea behind adversarial example x^* is to solve the problem of the smallest possible perturbation that causes misclassification:

$$x^* = x + \texttt{argmin}\{||z|| : f(x+z) = t\}, \tag{6.1}$$

where x represents an input that originally is correctly classified, $||\cdot||$ is a norm that defines similarity constraints between x^* and x. Norm and, consequently, z value defines how the algorithm should change the original sample. For example, on an image, l_0 norm might be used to produce noticeable changes, but for several pixels only, whereas l_∞ norm to modify every pixel by a tiny amount [20]. t defines a target class, it can be any different than the original class $f(x)$ in case of an *untargeted* attack, or have a specific class value for *targeted misclassification* [20].

Goodfellow et al. [20] describe three canonical examples to achieve this goal. First, the Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) algorithm is a quasi-Newton method used for parameter optimization. The other two algorithms are the Fast Gradient Sign Method (FGSM) and Jacobian Saliency Map Approach (JSMA).

JSMA and L-BFGS are iterative algorithms and produce stealthier perturbations that are harder to detect. However, it is achieved at a price of higher computational cost. FGSM, on the contrary, is much faster and may work better than the previous two if the gradient is relatively small. Needless to say that any of mentioned methods may fail to fool the classifier. The exception is L-BFGS, which almost always succeeds given enough computation time since it is essentially a brute-force approach [20].

*https://attack.mitre.org/

6.2 Black-Box attacks

While white-box attacks are essential for understanding the nature of adversarial examples, the situation when an attacker has access to the inner functionality of the target model is relatively rare. The more realistic threat model is a *black-box* scenario when the adversary does not possess knowledge about the inner architecture or training of the target model [35]. Usually, an attacker may have only the model's predictions via a limited API interface. In research, such a model is called the *oracle* O. It is crucial to explicitly distinguish between two potential outputs from the oracle-predicted label $\tilde{O}(x)$ of input x and probability vector $O(x)$. Label, in this case, is the most probable of N classes:

$$\tilde{O}(x) = \underset{0 \cdots N-1}{\operatorname{argmax}} O(x)$$

Papernot et al. [35] were the first to prove that it is possible to generate an adversarial example provided such a threat model. Assumptions of such model are (a) adversary does not posses knowledge about inner state of model, having access only to prediction $\tilde{O}(x)$, and (b) adversary cannot collect a dataset comparable with target model's training data. They overcome both limitations by (a) training a substitute model and (b) generating a synthetic training set acquired using adversary inputs and labels from the oracle.

Adversarial examples are generated using the FGSM algorithm on the substitute model after training on synthetic augmented data for a manually defined number of epochs p. They validate the attack strategy against the remote Deep Neural Network (DNN) model hosted by Oracle using either MNIST (hand-written digits) or GTSRB (road sign) datasets. Attack reaches up to 84.24% success rate on the target model.

Such a technique appears to be efficient due to the "transferability" property first observed by Szegedy et al. [48]. Transferability is not DNN specific and holds across many machine learning models. For instance, the substitute logistic regression model can successfully mimic the decision boundary of the support vector machine and decision tree models having similar perturbation norms [34].

6.3 Adversarial Malware

The presence of adversaries in the malware classification domain is a real-world fact. Therefore, building a robust model that can defend against adversarial inputs is essential for this field. It is shown that attacks harnessed in the image classification domain case be transferred to ML-based classifiers. MalConv [40] model's open parameters and similarity with image classification architectures bootstrapped adversarial research for

malware evasion.

Kolosnjaji et al. [27] was one of the earliest to weaponize a version of gradient descent in a padding attack and prove that it is possible to generate adversarial malware samples. They prove that it is possible to craft targeted perturbations on malware samples given the white-box threat model. By injecting less than 1% original sample bytes, Kolosnjaji et al. decreased MalConv accuracy by over 50% [27].

Still, they use naive file modification, injecting perturbations as an overlay, appending new bytes at the end of the file. Such a technique is trivial to detect with a simple manual check. In opposition to that, modifying the Portable Executable (PE) structure is a complex task. As shown in Section 2.1, PE files have a specific structure with different headers and sections. The operating system expects to find specific bytes in particular places. Otherwise, the operating system will not execute PE.

A more sophisticated PE modification is shown by Kreuk et al. [29], who injects adversarial bytes to unused parts of already existing sections. They utilize FGSM to generate AE and show that it is possible to achieve more than 99% evasion rate on MalConv model trained using EMBER dataset [5].

White-box attacks are essential from a conceptual point of view. They reveal the action space and technical boundaries of the evasion problem. However, as observed in the wild or a report by Karantzas and Patsakis [24], practical evasions of defensive solutions happen preemptively, without knowledge of inner architecture and heuristics used by a security product. Therefore, we argue that the threat model of interest in modern adversarial malware research is mostly black-box. We provide an in-depth review of notable techniques used to generate adversarial malware samples in a black-box manner, with reinforcement learning (RL) based algorithms discussed in Section 6.3.1 and genetic algorithms in Section 6.3.2.

These two algorithmic families are not exhaustive for this problem space, with other techniques are known to be possible, for instance, utilization of generative recurrent neural network (RNN) by Ebrahimi et al. [18], or generative adversarial networks (GAN) by Hu et al. [22]. However, we focus on RL and genetic algorithms since these have the most applied realizations for Windows malware samples.

6.3.1 Reinforcement Learning Algorithms

Anderson et al. [4] were, to our knowledge, the first to introduce reinforcement learning based black-box attacks with functional malware samples. They formulated the evasion problem as a sequence of discrete timesteps t, where for each step, an evasion agent may select an action $a_t \in \mathbb{A}$ to produce state vector s_t from a malicious PE sample, where $\mathbb{A}$ represents an action set, modifications agent can produce with bi-

nary. Afterward, a sample is exposed to the environment, which consists of acquiring a score from a malware classifier to produce a reward r_t in response to a previously applied action. Based on reward, the agent incrementally learns policy $\pi(a|s_{t+1})$, that maximizes expected return:

$$V^{\pi}(s_t) = \mathbb{E}_{a_t}[Q^{\pi}(s_t, a_t)|s_t],$$

where Q^{π} embodies action transformation to reward.

It is interesting to note that after the RL agent is trained on 50K malware samples, its capabilities are still poor, with only a few percent increase in evasion rates against random action applications (for example, 24% versus 23% respectively on a subset of samples as reported by Anderson et al. [4]). However, this reveals another outstanding trait of this publication besides conceptual novelty - efficient action space on PE sample that alters characteristics of a specimen while preserving malicious functionality, allowing to achieve high evasion rates even with random manipulations.

Furthermore, we see that core of this action space is utilized by Song et al. [45] while improving RL agent functionality. They develop a multi-armed bandit (MAB) framework, but in addition to binary rewriter, implement a systematic action minimizer that cancels already applied non-essential actions, minimizing attack footprint on the final adversarial sample.

In addition to efficient adversarial sample generation, this work has noteworthy implications for the explainability of ML-based malware classifiers since revealing essential actions demonstrates the model's decision heuristic. For example, they show that both MalConv and Ember GBDT evasions are achieved mainly by overlay byte appending, therefore emphasizing the statistical feature engineering of these classifiers. At the same time, analysis of evasive samples against three commercial products reveals unique patterns like modifying some sections or appending just 1 byte to sections result in misclassification, proving that hash-based signatures and section name detection heuristics are in place.

Similarly, Fang et al. [19] develop an idea of RL agent utilization for sample generation but reduce and increase the sophistication of action space. One of the potential problems behind the poor performance of the RL agent in [4] is the exponentially ample search space. Without action minimization as introduced in [45] or a limited set of highly efficient actions like in [19], the RL agent cannot find a practical set of modifications within a realistic timeslot.

6.3.2 Genetic Algorithms

An alternate branch of adversarial malware research is influenced by genetic algorithms, where Xu et al. [50] were among the first to reveal the relevancy of this algorithmic

family on malicious sample generation. His team formed adversarial PDF documents capable of evading detectors trained on non-adversarial samples, and later this idea was successfully ported to adversarial malware generation from Windows PE binaries [14, 30].

Similar to RL, genetic algorithms allow defining model evasion with a black-box threat model as an optimization problem. For instance, GAMMA (Genetic Adversarial Machine learning Malware Attack) as formulated by Demetrio et al. [14] applies a set of mutations from action space $s \in S$ at a generation q and incrementally modifies initial malware sample x. Furthermore, they construct a constrained optimization problem:

$$\min_{s \in S} F(s) = f(x \oplus s) + \lambda \times C(s),$$

subject to $q \leq T$, where $f(x \oplus s)$ represents the model output on the manipulated sample, $C(s)$ penalty term that evaluates the amount of injected bytes, with hyperparameters $\lambda > 0$ that mediates between the two terms and T as an upper bound for a maximal number of generations.

The solution of minimization problem is achieved through a genetic algorithm that iterates over three steps representative for biological evolution: *selection*, *crossover*, and *mutation* [14]. The *selection* chooses N best candidates of action set population at generation q based on objective function, *crossover* function takes selected N actions and returns an novel set of N candidates, with consequent *mutation* step applying changes at random, with low probability. The combination of *crossover* and *mutation* produces samples that have significant differences from previous generation to properly explore the space of applicable solutions.

GAMMA action space is similar to one observed in RL-based attacks [4, 19, 45] and consists of manipulations on compiled binaries. Arguably, this is one of the most notorious drawbacks of contemporary automated adversarial malware research since real-world threat actors possess the source code of malicious samples and can utilize AI to build malicious samples with pre-compilation modifications. Luckily, we notice a shift in this trend. To give an instance, Kucuk et al. [30] explore a genetic algorithm adversarial attack that is capable of source code modification, and Ceschin et al. [10] provides a set of practical, simple techniques on a pre-compiled malware in a manual manner.

7. Adversarial Robustness of Hybrid Model

As discussed in Chapter 6, contemporary adversarial attacks achieve high evasion rates against SoTA solutions like Ember GBDT [5] and MalConv [40] by altering structure of PE files, while preserving malicious functionality. We evaluate the efficiency of adversarial malware against the hybrid solution built in this work. Section 7.1 describes specifics of adversarial sample generation for our experiments, and Section 7.2 reports performance of our solution on adversarial dataset.

7.1 Adversarial Sample Generation

A large portion of adversarial malware research does not provide artifacts for reproducibility, reviewing the only conceptual description of attack methodology. As discussed in Section 6.3, the creation of adversarial malware samples is a subtle process with a high degree of nested dependencies and configuration variability. A direct replica of the attack without research artifacts like specific algorithm implementation or code sample is highly improbable. Luckily, attacks described by Demetrio et al. [14] are represented in accompanied `secml_malware` library [13] focused on adversarial robustness evaluation of Windows malware classifiers.

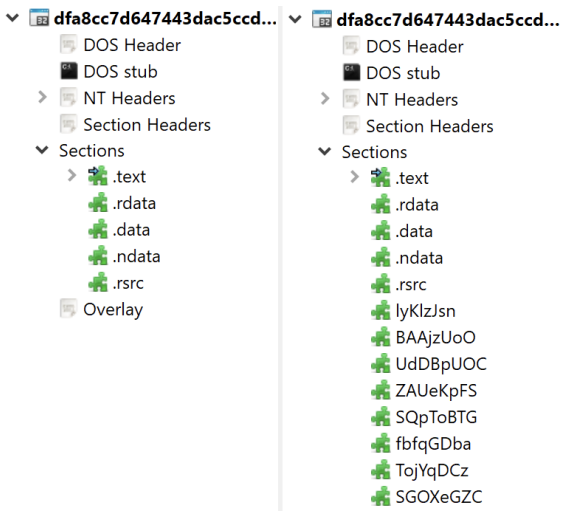

Figure 7.1: Structure of PE file before (left) and after (right) GAMMA attack (observed in PEbear executable file parser).

We generate adversarial samples using section injection with GAMMA algorithm discussed in Section 6.3.2. Files after attack have additional PE sections as presented in Figure 7.1, which alter static properties of file like byte histogram and strings, however

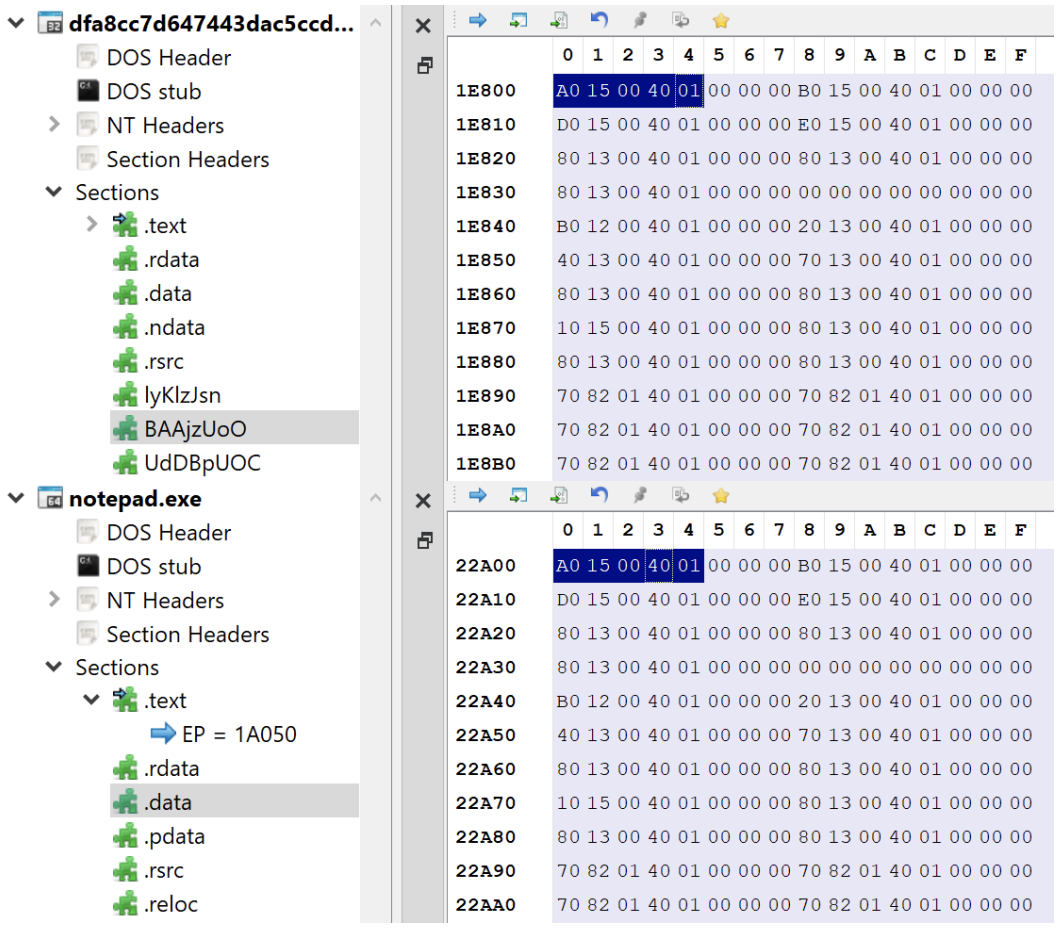

Figure 7.2: Added sections are taken from provided goodware samples.

does not affect the execution logic [14]. We configure the algorithm to use `.rdata` and `.data` sections from legitimate software which is represented by a dozen of default Microsoft applications, for instance, `notepad.exe`, `calc.exe`, and `nslookup.exe`. As visualised in Figure 7.2, one of the adversarial sample sections is a one to one copy of `notepad.exe` `.data` section. Attack configuration is available in accompanying repository*.

GAMMA needs an *oracle* to evaluate modified samples. It is executed in two separate modes, targeting MalConv and Ember GBDT. We launch an attack only against successfully detected malware samples in the validation set for each target, which are 12747 samples for MalConv and 12604 for Ember GBDT. Since the GAMMA algorithm performs multiple queries of intermediate sample versions during the attack, processing the whole corpus against MalConv takes about one day. Against Ember, the GBDT attack lasted four days. Therefore, our options to evaluate different attack configurations are limited by resource constraints. However, we form an adversarial

*https://github.com/dtrizna/quo.vadis/blob/main/evaluation/adversarial/secml_malware/gamma_run.py

corpus using a variable number of injected sections, with Ember GBDT as an oracle, namely 5, 10, and 15 sections.

Target	MalConv (15 sections)	Ember GBDT (15 sections)	Ember GBDT (10 sections)	Ember GBDT (5 sections)
Total samples	6887	8896	8995	9020
Successful	4023	5399	5438	5464
Errors	2864	3497	3557	3556
Success rate	58.41%	60.69%	60.46%	60.58%

Table 7.1: Adversarial sample emulation statistics.

Our experiments show that not all adversarial samples are functional after modifications. We observe that on average about 40% of binary files after section injections are not functional, with detailed statistics reported in Table 7.1. Based on manual analysis, we can conclude that those represent packed malware samples, and we assume that section injection interferes with the unpacking routine. We perform the final analysis only on functional adversarial malware, considering broken samples as unsuccessful and discarding from further statistics. Additionally, some of the GAMMA executions fail with `AttributeError, FitnessMin` exception, which are subject of `secml_malware` library limitations, therefore the number of total samples in Table 7.1 is variable.

7.2 Performance on Adversarial Malware

The focus of adversarial robustness tests is to evaluate whether simultaneous utilization of static, dynamic, and contextual parameters of executed malware will provide the necessary epistemic background to lessen the effects of an adversarial attack. We evaluate an absolute number of evasive samples, evasion rates, and detection accuracy based on different configurations of enabled modules in a hybrid model. We acknowledge the high efficiency of the GAMMA attack against static classifiers. As reported in Figure 7.3, detection rate of MalConv and Ember GBDT drop to 40% and 72% respectively. Recall from Section 7.1 that attack is launched only against successfully detected samples. Therefore, original samples from this set were detected with a 100% rate.

Enabling dynamic and contextual hybrid model units severely reduces the efficiency of the attack. As seen in Figure 7.3, adding a dynamic analysis module based on API call sequence emulation in conjunction with a static model severely reduces the number of evasive samples (note that the y scale is logarithmic). Although the static model reports a benign label for a subset of adversarial samples, the late fusion model

Modules	Ember GBDT	Emulation	GBDT & Emulation	All modules
Original set	0	24	5	61
Adversarial set	1515	127	236	80
Evasion rate, ϵ	28.06%	1.91%	4.28%	0.35%

Table 7.2: Absolute count of evasive samples and evasion rates for both original and adversarial 5399 malware images forming an functional adversarial set after GAMMA attack with 15 section injection.

builds a robust heuristic to distinguish malevolent logic based on API calls even if the verdict produced by the emulation module contradicts the static model prediction.

The adversarial algorithm is capable of producing **2376** samples marked as benign by MalConv and **1515** benign executables for the Ember GBDT model. By adding emulation-produced API calls to a hybrid model, evasive sample counts drop to **295** and **236** for MalConv and Ember, respectively. Contextual information in form of file paths decreases counts further to **249** and **80**.

Quantitative statistics for an attack against the Ember GBDT model with 15 injected sections are reported in Table 7.2. As mentioned above in Section 7.1, GAMMA produces adversarial samples only from already detected malware by the target model. Therefore, Table 7.2 reports zero evasive samples for Ember GBDT in original set. However, adding modules to a hybrid solution shifts the decision boundary. Thus, we observe that some of the samples from the original set were evasive for the hybrid model before GAMMA manipulations. For instance, 5 samples were marked as benign by Ember GBDT and API call models, and 61 by adding a filepath model.

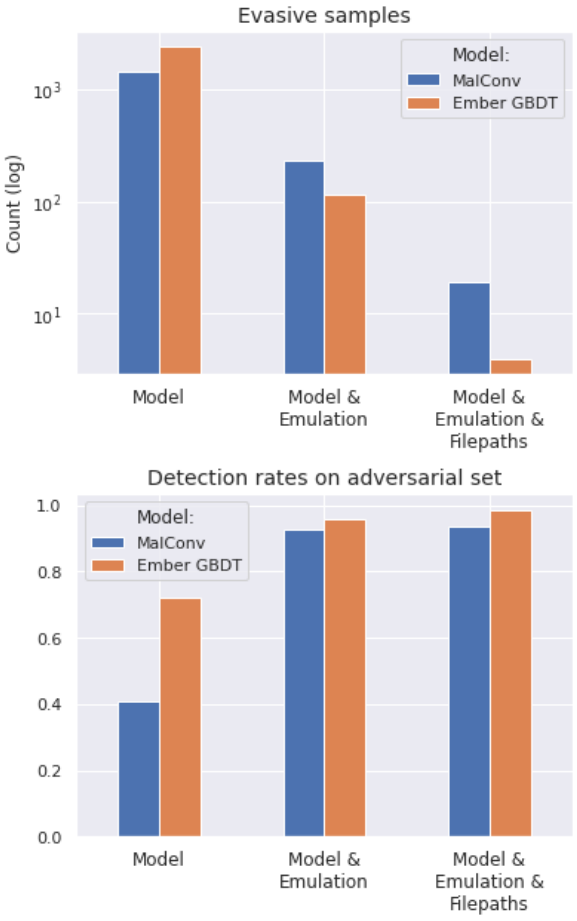

Figure 7.3: Chart representing absolute counts of evasive samples and detection rates based on enabled modules in hybrid solution.

Therefore, we use the *evasion rate* [32, 15] to illustrate attack's efficiency ratio against a specific model setup. If the effect of the GAMMA attack on sample is denoted as δ, true label of sample as y, and model as formulated in Eq. 5.1, then:

$$e' \in A : \psi(\phi(e + \delta)) \neq y,$$

$$e \in A : \psi(\phi(e)) \neq y,$$

with e' representing a number of evasive samples in an adversarial set, and e evasive samples in original set before execution of the attack. The evasion rate is calculated as follows:

$$\epsilon = \frac{\Delta e}{|A|} = \frac{e' - e}{|A|},$$

where $|A|$ is total number of functional samples produced by attack, and Δe represents of evasive samples added by the attack.

The evasion rate shows that in case of hybrid model, GAMMA's contribution to evasion abilities is even less significant since some were evasive before adversarial manipulations. Thus, adding emulation-based analysis drops the evasion rate from 28.06% to 4.28%, file path module reduces it further to 0.35%, practically eliminating the attack's effect.

Injected Sections	Ember GBDT	Emulation	GBDT & Emulation	All modules
5	0.7822	0.7901	0.9561	**0.9843**
10	0.7775	0.7893	0.9555	**0.9833**
15	0.7194	0.9765	0.9555	**0.9852**

Table 7.3: Classification accuracy dependency on number of injected sections by GAMMA.

We examine whether attack configuration allows bypassing this property of the hybrid model. We evaluate a more subtle form of attack by inserting fewer sections into the original binary. Results of these experiments against Ember GBDT are reported in Table 7.3. It is seen that the number of injected sections has a negative linear dependency with Ember GBDT accuracy. However, these changes do not affect other modules and overall composite solution metrics (except for an anomalous gap in the emulation module's accuracy between 10 and 15 sections).

Empirical evidence shows that the GAMMA attack is not efficient against hybrid architecture that incorporates static, dynamic, and contextual features of PE. We assume this extrapolates to other adversarial techniques targeted at altering PE properties that modify the feature space of static detectors. These attacks do not affect PE properties evaluated by emulation and file path module. A successful attack against

such a hybrid system should incorporate multipurpose action space that allows altering static, dynamic, and contextual properties and is the subject of discussion in Chapter 8.

8. Future work

Filepath module architecture. The promising direction of deep learning model architecture optimization might be parsing directory and filename data in separate arms of neural network with distinct trainable parameter sets $\theta_{DIR}, \theta_{FN}$, with a further concatenation of two separate data representations in a final FFNN with θ_{OUT}. For example, `svchost.exe` under `"[drive]\windows\system32"` is present on every system, however if spotted within `"[drive]\users\[user]\appdata"` raises immediate suspicion. Therefore, the same filename under different folders might turn around the results. The current pipeline should be able to distinguish both files. However, controlled experiments in training two separate parameter sets might be fruitful. Additionally, targeted feature engineering might be beneficial, like providing directory permissions, for example, whether it is world-writable or not.

Filepath module error correction. Filepath module might benefit from careful error correction, with oversampling of specific paths and additional data augmentation with patterns to emphasize important path components for the model. Such work can yield targeted results on false negative and false positive minimization without producing overfitting.

Extension of emulator capabilities. Additional work on boosting the capabilities of the kernel emulator might be fruitful. For instance, robustness against anti-debugging techniques. In addition, manual review of erroneous emulation reports from our or additional malware corpora might introduce emulator modifications to combat self-defending malware and reveal its actual functionality.

Improvement of API call optics. Our implementation of the API call sequence module ignores call arguments. Representation of this information might bring valuable information for security detection heuristics. Additionally, at least one paper outlines that incorporating this information might make behavioral attacks less successful [41]. Another pre-processing improvement might consider multiple API call windows rather than one sequence per sample. Longer sequences can be divided into n separate win-

dows with k calls each (last one padded) rather than cropping the first k API calls. A malicious score then could be assigned if any of the API call windows are considered malicious. The module will be more robust against evasion techniques like injecting malicious code into legitimate software or suspending malicious logic by introducing legitimate API calls at the beginning of the application - like writing empty files or reading default registry values.

Limitations of API call sequence approach as a behavior representation. While we represent PE behavior on a system with an API call sequence, not all image functionality is expressed by API calls. For example, code can implement string de-obfuscation or anti-debugging functionality using just assembly instructions defined within `.text` section, without referring to the operating system libraries at all. Providing such instructions for a machine learning model is a subject for assembly level analysis, an emerging field of research. For instance, there is evidence that graph neural networks are capable of representing the behavior of executable from assembly instruction graphs [49] like one visualized in Figure 8.1.

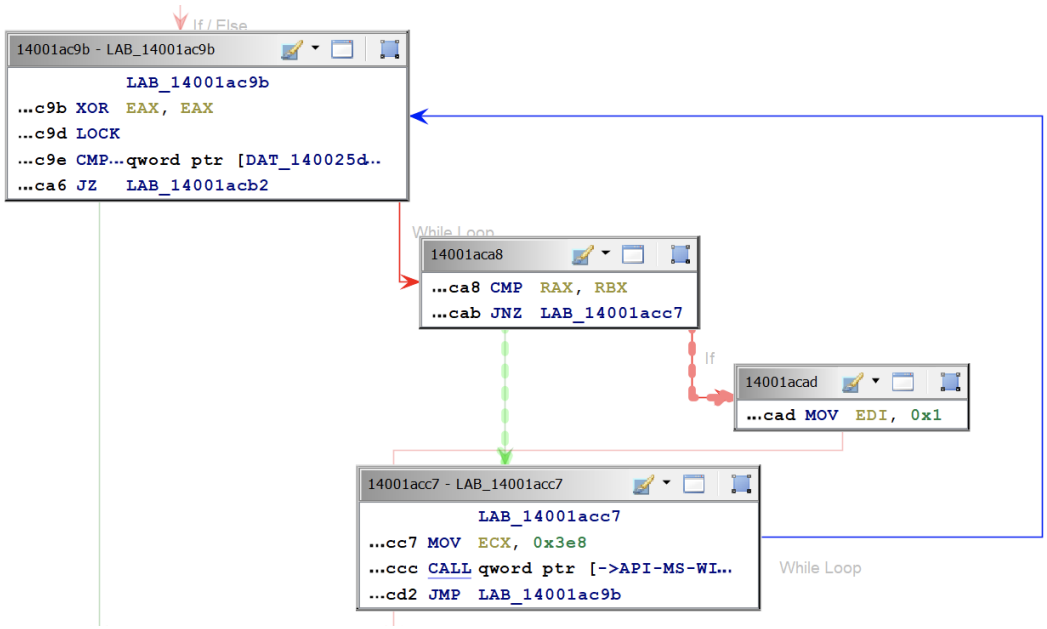

Figure 8.1: Part of `notepad.exe` functionality represented as an assembly instruction graph.

Extended modularity of hybrid solution. Additional visibility sources might be crucial to minimize ambiguity in the model decision heuristic. We consider this as promising direction, with additional optics from the emulator like network communications, files and registry manipulations. These are the indicators human analysts refer to during forensic or incident investigations and are essential for dividing malicious and legitimate activity. Since the emulation report dataset is released publicly,

we contribute and expect the development of efficient approaches in this direction. Additional contextual data like process creation chains is of high interest too.

Improvement of adversarial attacks. We have shown that existing adversarial attacks have a minor effect on behavioral and contextual analysis. Moreover, practically all the existing attacks alter already compiled malicious samples. On the contrary, real-world threat actors possessing the power of AI for malware creation will have the ability to modify source code directly. Therefore, by building more robust malware classifiers, we encourage to consider extending the action space of adversarial algorithm that affects the behavioral properties of a malicious sample, presumably with modifications on a source code level. These might include:

- hiding parts of application data behind XOR procedure with predefined key;
- encode parts of an image with base64 to modify byte distribution;
- adding futile cycles or code parts that do not affect original execution flow but are present in legitimate software, such as common system files and registry modifications.

9. Conclusions

In this work, we presented a hybrid machine learning architecture that employs the Windows Portable Executable's static, behavioral, and contextual properties. We hypothesized that the heterogeneous nature of hybrid analysis would yield improved performance and adversarial robustness by providing diverse telemetry to narrow the visibility gap present in current state-of-the-art solutions.

Our experiments reveal that simultaneous utilization of static, dynamic, and contextual information provides added epistemic value for ML-based detection heuristic. A hybrid solution can detect a malevolent sample even if none of the individual components express enough confidence to classify input as malicious. For instance, given a fixed false positive rate of 0.1%, Ember GBDT, filepath, and emulation API call modules report detection rates of 23.46%, 23.43%, and 42.49% respectively. In contrast, composite utilization of all three modules detects 96.69% of samples, boosting detection rates above the cumulative capabilities of individual modules.

Moreover, we observe that the hybrid model addresses independent method weaknesses, improving the reliability of separate modules, thus minimizing false positive and false negative rates. This property directly transfers to increased adversarial robustness. We analyzed the efficiency of the GAMMA adversarial malware attack against our solution. We reported inefficiency of the attack, with evasion rate decrease from 28.06% against Ember GBDT to 0.35% given additional use of contextual and behavioral executable file properties. This emphasizes the limited scope of contemporary adversarial malware attacks, illuminating the need to explore more sophisticated action space.

Bibliography

[1] A. F. Agarap. Deep learning using rectified linear units (relu). arXiv, Neural and Evolutionary Computing (cs.NE), 2019.

[2] R. Agrawal, J. W. Stokes, M. Marinescu, and K. Selvaraj. Robust neural malware detection models for emulation sequence learning. arXiv, Artificial Intelligence (cs.AI), 2018.

[3] R. Agrawal, J. W. Stokes, K. Selvaraj, and M. Marinescu. Attention in recurrent neural networks for ransomware detection. In *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3222–3226, 2019.

[4] H. S. Anderson, A. Kharkar, B. Filar, D. Evans, and P. Roth. Learning to evade static PE machine learning malware models via reinforcement learning. arXiv, Cryptography and Security (cs.CR), 2018.

[5] H. S. Anderson and P. Roth. Ember: An open dataset for training static pe malware machine learning models. arXiv, Cryptography and Security (cs.CR), 2018.

[6] B. Athiwaratkun and J. W. Stokes. Malware classification with lstm and gru language models and a character-level cnn. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2482–2486, 2017.

[7] F. Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, page 41, USA, 2005. USENIX Association.

[8] B. Biggio, I. Corona, B. Nelson, B. I. Rubinstein, D. Maiorca, G. Fumera, G. Giacinto, and F. Roli. Security evaluation of support vector machines in adversarial environments. arXiv, Machine Learning (cs.LG), 2014.

[9] M. Botacin, F. D. Domingues, F. Ceschin, R. Machnicki, M. A. Zanata Alves, P. L. de Geus, and A. Gregio. Antiviruses under the microscope: A hands-on perspective. *Computers & Security*, 112:102500, 2022.

[10] F. Ceschin, M. Botacin, G. Lüders, H. M. Gomes, L. Oliveira, and A. Gregio. No need to teach new tricks to old malware: Winning an evasion challenge with xor-based adversarial samples. In *Reversing and Offensive-Oriented Trends Symposium*, ROOTS'20, pages 13–22, New York, NY, USA, 2020. Association for Computing Machinery.

[11] T. Chen and C. Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[12] A. Cherepanov. Operation Groundbait: Analysis of a surveillance toolkit, 05 2016.

[13] L. Demetrio and B. Biggio. secml-malware: A python library for adversarial robustness evaluation of windows malware classifiers. arXiv, Cryptography and Security (cs.CR), 2021.

[14] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando. Functionality-preserving black-box optimization of adversarial windows malware. *IEEE Transactions on Information Forensics and Security*, 16:3469–3478, 2021.

[15] L. Demetrio, S. E. Coull, B. Biggio, G. Lagorio, A. Armando, and F. Roli. Adversarial exemples: A survey and experimental evaluation of practical attacks on machine learning for windows malware detection. *ACM Trans. Priv. Secur.*, 24(4), Sept. 2021.

[16] F. N. Ducau, E. M. Rudd, T. M. Heppner, A. Long, and K. Berlin. Automatic malware description via attribute tagging and similarity embedding. arXiv, Machine Learning (cs.LG), 2020.

[17] C. Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, USA, 2008.

[18] M. Ebrahimi, N. Zhang, J. Hu, M. T. Raza, and H. Chen. Binary black-box evasion attacks against deep learning-based static malware detectors with adversarial byte-level language model. arXiv, Cryptography and Security (cs.CR), 2020.

[19] Z. Fang, J. Wang, B. Li, S. Wu, Y. Zhou, and H. Huang. Evading anti-malware engines with deep reinforcement learning. *IEEE Access*, 7:48867–48879, 2019.

[20] I. Goodfellow, P. McDaniel, and N. Papernot. Making machine learning robust against adversarial inputs. *Commun. ACM*, 61(7):56–66, June 2018.

[21] R. Harang and E. M. Rudd. SOREL-20M: A large scale benchmark dataset for malicious PE detection. arXiv, Cryptography and Security (cs.CR), 2020.

[22] W. Hu and Y. Tan. Generating adversarial malware examples for black-box attacks based on gan. arXiv, Machine Learning (cs.LG), 2017.

[23] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv, Machine Learning (cs.LG), 2015.

[24] G. Karantzas and C. Patsakis. An empirical assessment of endpoint detection and response systems against advanced persistent threats attack vectors. *Journal of Cybersecurity and Privacy*, 1(3):387–421, 2021.

[25] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[26] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. arXiv, Machine Learning (cs.LG), 2017.

[27] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. *CoRR*, abs/1803.04173, 2018.

[28] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert. Deep learning for classification of malware system call sequences. In *AI 2016: Advances in Artificial Intelligence*, volume 9992, pages 137–149, 12 2016.

[29] F. Kreuk, A. Barak, S. Aviv-Reuven, M. Baruch, B. Pinkas, and J. Keshet. Deceiving end-to-end deep learning malware detectors using adversarial examples. arXiv, Machine Learning (cs.LG), 2019.

[30] Y. Kucuk and G. Yan. *Deceiving Portable Executable Malware Classifiers into Targeted Misclassification with Practical Adversarial Examples*, pages 341–352. Association for Computing Machinery, New York, NY, USA, 2020.

[31] A. Kyadige, E. M. Rudd, and K. Berlin. Learning from context: A multi-view deep learning architecture for malware detection. In *2020 IEEE Security and Privacy Workshops (SPW)*, pages 1–7, 2020.

[32] R. Labaca-Castro, L. Munoz-Gonzalez, F. Pendlebury, G. D. Rodosek, F. Pierazzi, and L. Cavallaro. Realizable universal adversarial perturbations for malware. arXiv, Cryptography and Security (cs.CR), Artificial Intelligence (cs.AI), 2021.

[33] Mandiant. Speakeasy: portable, modular, binary emulator designed to emulate Windows kernel and user mode malware. https://github.com/mandiant/speakeasy, 11 2021.

[34] N. Papernot, P. McDaniel, and I. Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. arXiv, Cryptography and Security (cs.CR), 2016.

[35] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami. Practical black-box attacks against machine learning. arXiv, Cryptography and Security (cs.CR), 2017.

[36] R. Pascanu, J. W. Stokes, H. Y. Y. Sanossian, M. Marinescu, and A. Thomas. Malware classification with recurrent networks. *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1916–1920, 2015.

[37] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[38] W. Pearce, N. Landers, and N. Fulda. Machine learning for offensive security: Sandbox classification using decision trees and artificial neural networks. arXiv, Cryptography and Security (cs.CR), 2020.

[39] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[40] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas. Malware detection by eating a whole exe. arXiv, Machine Learning (stat.ML), 2017.

[41] I. Rosenberg, A. Shabtai, Y. Elovici, and L. Rokach. Query-efficient black-box attack against sequence-based malware classifiers. In *Annual Computer Security Applications Conference*, ACSAC '20, pages 611–626, New York, NY, USA, 2020. Association for Computing Machinery.

[42] E. M. Rudd, F. N. Ducau, C. Wild, K. Berlin, and R. Harang. Aloha: Auxiliary loss optimization for hypothesis augmentation. arXiv, Cryptography and Security (cs.CR), 2019.

[43] J. Saxe and K. Berlin. eXpose: A character-level convolutional neural network with embeddings for detecting malicious URLs, file paths and registry keys. arXiv, Cryptography and Security (cs.CR), 2017.

[44] M. Schultz, E. Eskin, F. Zadok, and S. Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings 2001 IEEE Symposium on Security and Privacy*, pages 38–49, 02 2001.

[45] W. Song, X. Li, S. Afroz, D. Garg, D. Kuznetsov, and H. Yin. MAB-Malware: A reinforcement learning framework for attacking static malware classifiers. arXiv, Cryptography and Security (cs.CR), 2021.

[46] Soni, Anuj and Zeltser, Lenny. FOR610: Reverse-engineering malware: Malware analysis tools and techniques. SANS Institute, 2022.

[47] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[48] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. arXiv, Computer Vision (cs.CV), 2014.

[49] S. Vasisht, P. Tully, and J. Gibble. Annotating Malware Disassembly Functions Using Neural Machine Translation, 11 2021.

[50] W. Xu, Y. Qi, and D. Evans. Automatically evading classifiers. In *Proceedings of the 2016 network and distributed systems symposium*, volume 10, 2016.

[51] M. C. Yang and M. C. Chen. Theoretical investigation of composite neural network. arXiv, Machine Learning (cs.LG), 2019.

[52] Y. S. Yen, Z. W. Chen, Y. R. Guo, and M. C. Chen. Integration of static and dynamic analysis for malware family classification with composite neural network. arXiv, Cryptography and Security (cs.CR), 2019.

[53] P. Yosifovich, M. E. Russinovich, D. A. Solomon, and A. Ionescu. *Windows Internals, Part 1: System Architecture, Processes, Threads, Memory Management, and More (7th Edition)*. Microsoft Press, USA, 7th edition, 2017.

Appendix A. False Negative Rate Analysis

In Section 5.6 we discussed hybrid model performance, with detection rates per module reported in Figure 5.7. The visualization shows that adding MalConv to a decision heuristic still slightly increases detection rates, despite poor independent performance of MalConv. However, it is important to consider false negative rates, representing a ratio of missed malware samples. Analysis based on fixed false negative rates displayed in Figure A.1 allows us to see that adding MalConv to final decision increases false positive rates with the same number of false negatives.

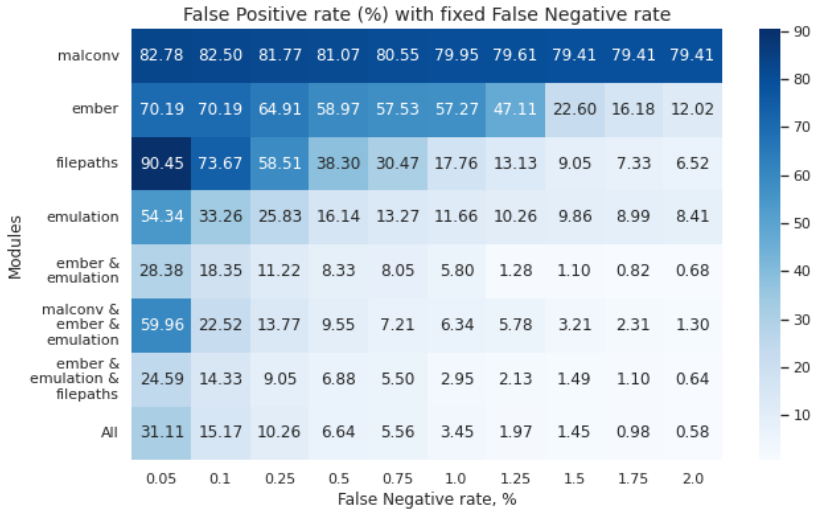

Figure A.1: False positive rate dependency on fixed false negative rate.

Increased false positive rates are especially noticeable on low false negative demands. For instance, Ember GBDT and emulated API call models with a 0.05% false negative rate (which represents a state of missing only one malware image per 2000 detected samples) yield 28.38% of false positives. However, adding MalConv as a third module increases false positive rates up to 59.95%. Therefore, by adding flawed components like MalConv, we can slightly increase the detection rate, but this comes at the cost of noticeably more false positive alerts, contributing to the final solution's poorer quality and alert fatigue problem widely acknowledged in security operation centers.