Neural Malware Detection

Sean Park

A dissertation submitted in fulfilment of the requirements

for the degree of

Doctor of Philosophy

Internet Commerce Security Laboratory

Federation University

December 2019

Table of Contents

Declaration

This thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Sean Park

December 2019

Abstract

At the heart of today's malware problem lies theoretically infinite diversity created by metamorphism. The majority of conventional machine learning techniques tackle the problem with the assumptions that a sufficiently large number of training samples exist and that the training set is independent and identically distributed. However, the lack of semantic features combined with the models under these wrong assumptions result largely in overfitting with many false positives against real world samples, resulting in systems being left vulnerable to various adversarial attacks.

A key observation is that modern malware authors write a script that automatically generates an arbitrarily large number of diverse samples that share similar characteristics in program logic, which is a very cost-effective way to evade detection with minimum effort. Given that many malware campaigns follow this paradigm of economic malware manufacturing model, the samples within a campaign are likely to share coherent semantic characteristics. This opens up a possibility of one-to-many detection. Therefore, it is crucial to capture this non-linear metamorphic pattern unique to the campaign in order to detect these seemingly diverse but identically rooted variants.

To address these issues, this dissertation proposes novel deep learning models, including generative static malware outbreak detection model, generative dynamic malware detection model using spatio-temporal isomorphic dynamic features, and instruction cognitive malware detection. A comparative study on metamorphic threats is also conducted as part of the thesis. Generative adversarial autoencoder (AAE) over convolutional network with global average pooling is introduced as a fundamental deep learning framework for malware detection, which captures highly complex non-linear metamorphism through translation invariancy and local variation insensitivity. Generative Adversarial Network (GAN) used as a part of the framework enables one-shot training where semantically isomorphic malware campaigns are identified by a single malware instance sampled from the very initial outbreak. This is a major innovation because, to the best of our knowledge, no approach has been found to this challenging training objective against the malware distribution that consists of a large number of very sparse groups artificially driven by arms race between attackers and defenders. In addition, we propose a novel method that extracts instruction cognitive representation from uninterpreted raw binary executables, which can be used for one-to-many malware detection via one-shot training against frequency spectrum of the Transformer's encoded latent representation. The method works regardless of the presence of diverse malware variations while remaining resilient to adversarial attacks that mostly use random perturbation against raw binaries.

Comprehensive performance analyses including mathematical formulations and experimental evaluations are provided, with the proposed deep learning framework for malware detection exhibiting a superior performance over conventional machine learning methods. The methods proposed in this thesis are applicable to a variety of threat environments where artificially formed sparse distributions arise at the cyber battle fronts.

Acknowledgements

Praise be to Prof. Iqbal Gondal, the most merciful, the most gracious, for blessing me with the courage, opportunity and intellect to undertake this research. I am profoundly indebted to my supervisor Prof. Joarder Kamruzzaman for his constant guidance, insightful advices, helpful criticisms and valuable suggestions. He has given me sufficient freedom to explore research challenges of my choice and guided me when I felt lost. Without his insights, encouragements and endless patience, this research would not have been completed.

My sincere gratitude goes to all my family members, Jamie, Gene, and Ewan for the great support during the entire period of my PhD research.

An important acknowledgment goes to Dr. Jon Oliver for proofreading, reviewing and providing valuable support to make this dissertation possible despite his busy schedule. I thank Internet Commerce Security Lab (ICSL) and Trend Micro for providing an excellent competitive environment and resources necessary to undertake this research. Finally, I thank all staffs and post-graduate students of Federation University for their inspirations during the last few years.

List of Publications

- Sean Park, Iqbal Gondal, Joarder Kamruzzaman, Jon Oliver, "Generative Malware Outbreak Detection" IEEE International Conference on Industry Technology. Melbourne, 2019.
- Sean Park, Iqbal Gondal, Joarder Kamruzzaman, and Leo Zhang, "One-Shot Malware Outbreak Detection using Spatio-Temporal Isomorphic Dynamic Features" 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering. Melbourne, 2019.
- Sean Park, Iqbal Gondal, Joarder Kamruzzaman, Jon Oliver, "Instruction Cognitive One-Shot Malware Outbreak Detection" 26th International Conference on Neural Information Processing of the Asia-Pacific Neural Network Society. Sydney, 2019.
- Sean Park, Iqbal Gondal, Joarder Kamruzzaman, Jon Oliver, "Comparative Study on Machine Learning Methods to Detect Metamorphic Threats" Malware Analysis using Artificial Intelligence and Deep Learning. Springer, 2020.

General Declaration – Published works

I hereby declare that this thesis does not contain any material that has been accepted for the award of any other degree or diploma at a university institution. I also state that, to the best of my knowledge and belief, this thesis does not contain any material previously published or written by another person, except where due references are made in the text of the thesis.

Four original papers (published and accepted) were part of this research in peer reviewed ranked conferences. I was fully responsible of developing and writing all the papers in the thesis under the supervision of Prof. Iqbal Gondal and Prof. Joarder Kamruzzaman. The following table gives level of my contribution for chapters 3, 4, 5 and 6:

Thesis Chapter	Publication Title	Publication Status	Contribution	Contribution %
3	Generative Malware Outbreak Detection	Published	Developed analytical model and experimental setup. I wrote the initial draft and incorporated the suggestions & recommendations from the supervisors to prepare the final draft	80%
4	One-Shot Malware Outbreak Detection using Spatio-Temporal Isomorphic Dynamic Features	Published	Developed analytical model and experimental setup. I wrote the initial draft and incorporated the suggestions & recommendations from the supervisors to prepare the final draft	80%
5	Instruction Cognitive One-Shot Malware Outbreak Detection	Published	Developed analytical model and experimental setup. I wrote the initial draft and incorporated the suggestions & recommendations from the supervisors to prepare the final draft	80%
6	Comparative Study on Machine Learning Methods to Detect Metamorphic Threats	Accepted	Developed analytical model and experimental setup. I wrote the initial draft and incorporated the suggestions & recommendations from the supervisors to prepare the final draft	80%

Student Signature: [illegible] Date: 17/12/2019

The undersigned hereby certify that the above declaration correctly reflects the nature and context of the student and co-authors contribution to this work.

Principal Supervisor Signature: [illegible] Date: 17/12/2019

List of Figures

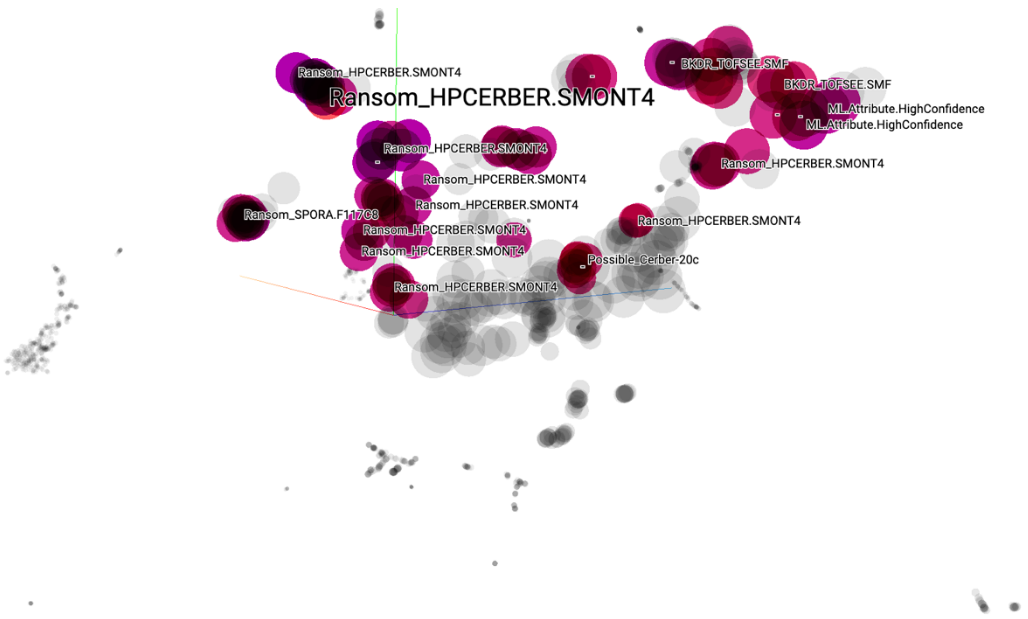
List of Tables

Acronyms

AAE	Adversarial Auto-Encoder
GAN	Generative Adversarial Network
CFG	Control Flow Graph
DFG	Data Flow Graph
API	Application Programming Interface
DGA	Domain Generation Algorithm
IR	Intermediate Representation
CPU	Central Processing Unit
DCGAN	Deep Convolutional Generative Adversarial Network
KNN	K-Nearest Neighbour
HDBSCAN	Hierarchical Density-Based Spatial Clustering of Applications with Noise
HMM	Hidden Markov Model
LLPO	Likelikhood Per Opcode
NLP	Natural Language Processing
DBN	Deep Belief Network
LSTM	Long Short-Term Memory
RNN	Recurrent Neural Network
GRU	Gated Recurrent Unit
URL	Uniform Resource Locator
CNN	Convolutional Neural Network
DAE	De-noising Auto-Encoder
CAE	Convolutional Auto-Encoder
VAE	Variational Auto-Encoder
BERT	Bidirectional Encoder Representations from Transformers
GPU	Graphical Processing Unit
LLGC	Learning with Local and Global Consistency
MIST	Malware Instruction Set
APK	Android Package
SVM	Support Vector Machine

ROP	Return Oriented Programming
FCG	Function Call Graph
RBF	Radial Basis Function
LCS	Longest Common Sub-sequence
MSA	Multiple Sequence Alignment
DLL	Dynamic Link Library
IAT	Import Address Table
MLP	Multi-Layer Perceptron
WEKA	Waikato Environment for Knowledge Analysis
EM	Expectation Maximisation
SVD	Singular Value Decomposition
RBM	Restricted Boltzmann Machine
PDF	Probability Density Function
KDE	Kernel Density Estimator
PCA	Principal Component Analysis
SELU	Scaled Exponential Linear Unit
RL	Reinforcement Learning
IoT	Internet Of Things
MSE	Mean Squared Error
TP	True Positive
TN	True Negative
FP	False Positive
FN	False Negative
FFT	Fast Fourier Transform
BBL	Basic Block
TLSH	TrendMicro Locality Sensitive Hashing
ROC	Receiver Operating Characteristic

Chapter 1: Introduction



Cyber-attacks [1] have been increasingly prevalent over benign internet community ever since the dawn of internet. The majority of cyber-attacks consist of a complex life cycle, which includes exploitation of software vulnerabilities, infiltration into the victim's machine, and exfiltration of information. Malware [2] plays a crucial role in this, enabling automatic execution at each step of the cyber-attack while evading detections by security software. From a cyber-attack defender's point of view, the fundamental goal is to identify the nature of arbitrary software to determine the maliciousness of the code and apply appropriate detection or prevention mechanisms. This is a challenging task in current threat landscape due to the volume of the samples to investigate, increased complexity of the code, and usage of obfuscation techniques [3] [4] [5]. In this chapter, we describe the background, scope and motivations, research challenges, research objectives, and overview of the contributions.

1.1 Background

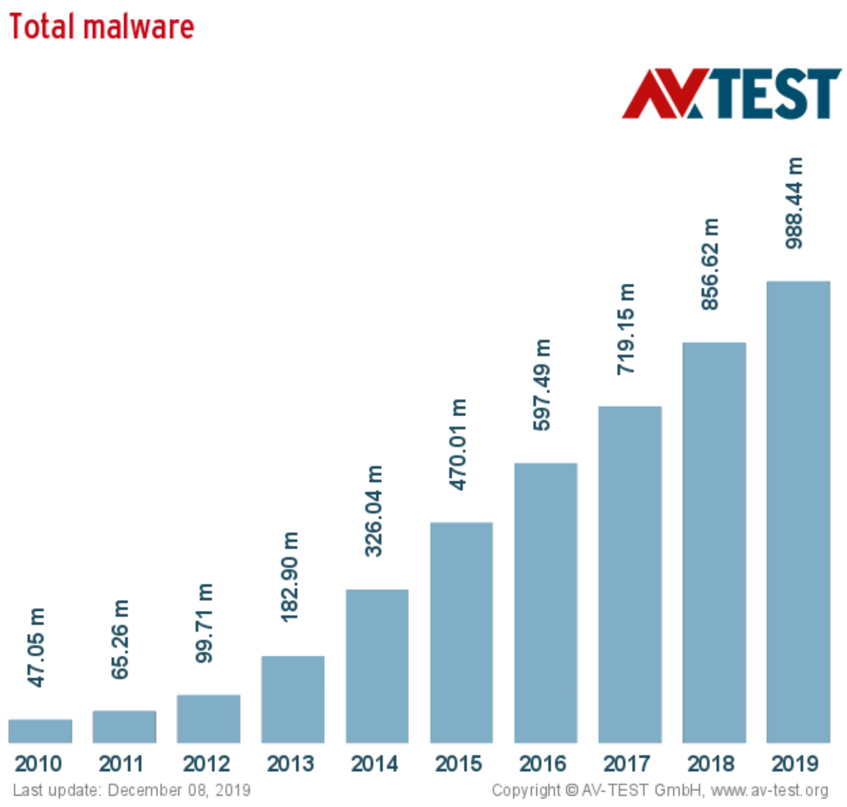


Figure 1.1. source: https://www.av-test.org/en/statistics/malware/

As shown in Figure 1.1, the volume of malware has sharply increased in the last decade with the next year's number expected to exceed 1 trillion based on the statistics provided by one of the AV benchmarking firms, AV-TEST.

The complexity of the malware code has dramatically increased in a battle against anti-malware systems. For instance, Cerber Windows malware has evolved over the past few years to evade detection systems by constantly mutating its code as shown in Figure 1.2.

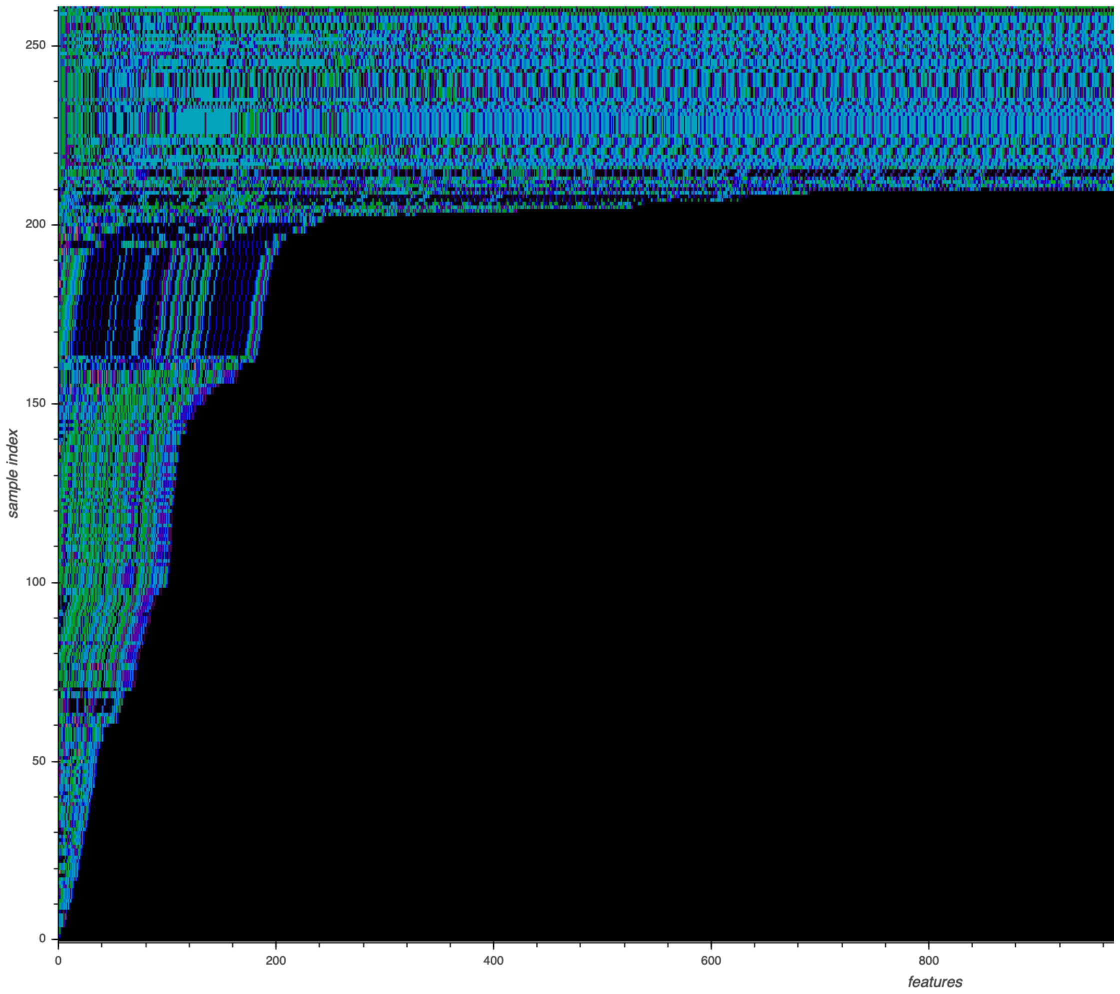


Figure 1.2. The evolution of Cerber malware family running in Windows operating system. Each horizontal line visualises the sequence of instructions for a Cerber variant. The figure shows 258 unique Cerber variants.

Internet Security Threat Report Volume 24 published by Symantec in 2019 [6] demonstrates increased diversity in malware distribution including formjacking that targets payment data, crytojacking that surreptitiously mines cryptocurrencies, constant ransomware prevalence, continued targeted attacks, attacks on vulnerable cloud systems, and IoT infections. Not only lies the diversity in the number of malware families, but also in the code patterns as shown in Figure 1.2.

In a nutshell, current malware detection problem is summarised with scalability of analysis, complexity of patterns, and diversity. Firstly, the sheer volume of incoming samples exponentially increases every year where automated precision detection is required. Particularly this scalability problem renders triaging probabilistically suspicious samples increasingly important in defence pipeline. Secondly, the key observation in malware domain is that the complexity of the static and dynamic pattern has significantly increased after decades of anti-cognition evolution that involves encryption, oligomorphism, polymorphism, and metamorphism. Combined with the scale of the volume, the malware analysis complexity exacerbates the malware problem, which then accelerates the need for highly accurate detection. Finally, the arms race between detection and evasion created a giant problem space where various custom anti-cognition scripts generate a theoretically infinite number of different malware samples. This diversity fundamentally defines the silent battle between the attackers and the defenders today.

Program cognition is an understanding of a given software sample, which can be as deep as cracking the hidden encryption key embedded within a suspicious file, or as simple as grouping samples based on their statistical characteristics. Deep program analysis such as malware reverse engineering has a great value in understanding the purpose of the software. At the same time, providing an accurate and timely detection covering a large number of malware variants forms a significant part of program cognition. Program cognition introduces extra complexity compared to the analysis of legitimate software due to the obfuscation and anti-analysis techniques deployed in various levels to hide the nature of the code. Anti-cognition techniques such as obfuscation and anti-analysis not only makes it difficult to analyse the code, but they also thwart the detection and automated analysis. The following sub-sections provide an overall insight into how program cognition can be accomplished in the presence of anti-cognition.

1.1.1 Static Analysis

One of the primary analysis methods is static analysis that analyses the program by inspecting the code line by line without executing it. Unlike dynamic analysis, static analysis enables analysts to explore all possible control flow paths without being restricted to the evaluated control paths that would have been executed at runtime. However, static analysis passes the burden of anti-cognition problem to the analysts whereas a large part of the anti-cognition problem is automatically circumvented in dynamic analysis.

The fundamental semantic building blocks that static analysis needs to find in order to achieve reversing goals include control flow graph (CFG) and data flow graph (DFG) from which high level design constructs and purpose of the program can be derived.

1.1.2 Dynamic Analysis

Dynamic analysis is a process to analyse a program by executing it in a controlled environment such as virtual machine or emulated machine, observing the behaviour and extracting information enough to make the decision of maliciousness of the program and to create actionable intelligence such as a pattern for detection and a remote location for malicious traffic.

Modern program logics are divided into modules that share similar functionality in order to split the load of programming complexity. Rather than tracing the program instruction by instruction, it is naturally a much more efficient to monitor Application Programming Interface (API), which is a set of entry points to the functions implemented in each module. The monitoring of function calls provides a quick overview of the program’s behaviour towards system resources such as file system, registry, network, and processes. These activities then help analysts to find high level design constructs and can be translated into the purpose of the program. Function call monitoring can be implemented using various hooking techniques within a virtual machine, or setting the trigger rules in binary instrumentation [7].

Although dynamic analysis efficiently reveals surface level design features of the program, it fundamentally lacks in identifying all control flow paths as it runs through a subset of the code blocks during execution. This poses a significant problem in reversing since it fails to discover a logic bomb [8] that will get activated at later time, and time-based mutation logic such as random domain generation algorithm (DGA) [9]. For example, the malware used in 2013 South Korea cyber-attack that simultaneously paralysed multiple financial services network and broadcasting systems was designed to get activated at a specific time [10]. Dynamic analysis approach will fail to analyse the piece of code that gets unpacked and executed when the predefined time is triggered. In addition, dynamic analysis is vulnerable to anti-analysis techniques with which the malware shows a convoluted behaviour and therefore prevents program analysis once it finds itself running within a controlled environment [7]. For example, malware can detect the analysis environment and cease its execution by fingerprinting various artifacts exposed by analysis platforms such as the debugger flag enabled for a debugging session, and the specific name of network interface adapter, virtual hard disk, file or registry key that is present only in controlled environments. Dinaburg et al. [11] created a transparent hardware based virtualized emulation system, Ether, dedicated for malware analysis in an attempt to prevent convoluted execution of malware that occurs when malware detects the presence of emulation environment. However, EtherUnpack and EtherTrace created as part of Ether project addresses the program cognition problem in a limited scope by considering specific unpacking approaches only. In addition, hardware virtualization does not provide a sufficient level of fine-grained program analysis.

1.1.3 Taint Analysis

While function call monitoring provides key checkpoints in the control flow, information flow tracking provides the knowledge of how the program data is processed. Using a dynamic taint system [12] [13], an analyst can introduce a taint source by marking interesting data, propagate the taint as the program runs, and check the taint status when the code reaches a point where predefined conditions meet. For example, a taint system can detect Windows login password theft by marking the user-entered password as the taint source, monitoring the data read access operation, and checking where the data ends up. This approach can be generalised to discover many different information theft attempts including key logging, password theft, network sniffing, stealth backdoors, spyware and rootkits.

Dynamic taint analysis technique has been used to acquire greater detail of program logic at instruction level by setting taint source, taint sink, and taint track policy [14] [15]. It opened up possibilities that binary code segments can be structurally and conceptually connectable in a way that high level program cognition is achievable. However there are several challenges to overcome to acquire accurate analysis such as under-tainting, over-tainting, and taint sanitization [12]. In addition, despite its fine-grained information flow tracking capability, taint analysis inherits most limitations dynamic analysis possesses because it essentially analyses the program's behaviour by executing the code.

1.1.4 Symbolic Execution

Schwartz et el. [12] summarised a key technique, called symbolic execution, to reason about the behaviour of a program by creating a logical formula for the program execution. King et al. [16] and Hom et al. [17] conducted a comprehensive study on symbolic execution. Symbolic execution combined with constraint solver [12] [18] gives us a normalized view of a given program code segment using Intermediate Representation (IR), which allows us to generically represent all binary code operations and bind the code and data relationships.

Several critical aspects of malware analysis have been addressed by a set of tools [19] using an emulator based on dynamic binary instrumentation, and symbolic execution. These tools demonstrate how these two techniques can be used to reveal trigger-based malware behaviour [20], to extract hidden malware code [21], to identify privacy breach [22], and to detect malware's hooking behaviours [23]. One of the problems inherent to symbolic execution in program analysis is denial-of-service attack to dynamic analysis, especially to fine-grained dynamic analysis techniques [19]. Since fine-grained symbolic execution requires significantly more processing power than native execution, malware can exploit this fact to effectively halt the analysis. Due to this limitation, symbolic execution based approach does not scale well for automated malware cognition process.

Most symbolic execution and taint analysis approaches utilise Intermediate Representation (IR) in order to make shadow variables explicit and to account for implicit operations. However, this extra step requires each native code to be translated to IR by hand, which is implemented incompletely due to rich set of instructions in modern CPUs.

Although symbolic execution offers the finest granularity of program analysis, its capability to achieve the malware program cognition goals is mainly limited by the burden of computational complexity and the incomplete symbolic emulation of native code.

1.1.5 Concolic Analysis

Concolic analysis is a hybrid technique that combines symbolic execution with dynamic analysis, in which program is analysed under a concrete execution path, and symbolic execution is used in conjunction with an automated theorem prover or a constraint solver based on constraint logic programming to generate new test cases that cover other concrete paths [24]. The main drawback of concolic analysis is manual setup of program inspection points while snapshots need to be managed for each inspection point. Therefore this approach is difficult to automate and incurs intolerable performance overhead, making it infeasible to apply in a large scale malware analysis.

Although symbolic execution combined with constraint solver promises to meet malware program cognition goals, the approach causes significant performance overhead due to IR translation layer and makes it difficult to automate the program analysis. It was mainly designed to help in-depth analysis of a single program such as condition identification of trigger-based programs and information flow tracking on a small input data set, instead of producing general program cognition constructs such as function call monitoring and information flow tracking. In addition, the emulation required for symbolic execution forces all machine instructions to be rewritten in order to execute each instruction symbolically. A complex instruction set in modern CPU architectures renders it nearly infeasible to precisely emulate all instructions.

1.2 Motivation

This thesis aims to develop malware detection methods that can be deployed under practical constraints present in the production environments and requirements, which include timely detection with acceptable accuracy and chronological detection.

Therefore the scope of this thesis is limited to static and dynamic malware detection, and triaging malware, as opposed to deep malware analysis.

Many previous studies have suffered from imperfect labelling, which caused various errors in evaluation at different levels. However, the representations given by the instruction sequence provide an attractive approximation of semantics, which can aid in mitigating the labelling problem primarily caused by the use of statistical features. For instance, Figure 1.3 shows the variants from two heterogeneous malware families. Regardless of the family names, their instruction-wise characteristics are similar, thereby invariant to labelling.

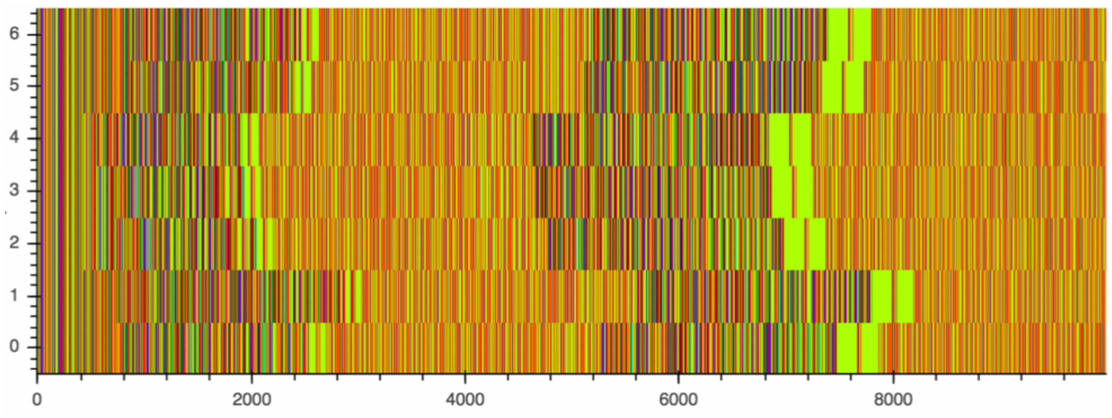


Figure 1.3. Three Blackhole and four Freezer variants

The key observation in most malware campaigns is that malware authors try to maximise the return on investment given that their time and resources are limited, which forces them to create maximum diversity under these constraints. This drives the malware authors to develop a script that automatically generates a large number of samples for distribution. Metamorphism plays a key role in this process, creating a set of samples with diverse appearances. As a result, what the world sees is a malware campaign that consists of a series of outbreaks that attempt to infect the victim machines. The majority of serious campaigns follow this paradigm of economic malware generation model. For example, Figure 1.4 illustrates four distinct patterns of malware samples visualised using instruction sequence, all of which are clearly recognisable and separable by a human being with decent visual cortex system and unharmed brain function.

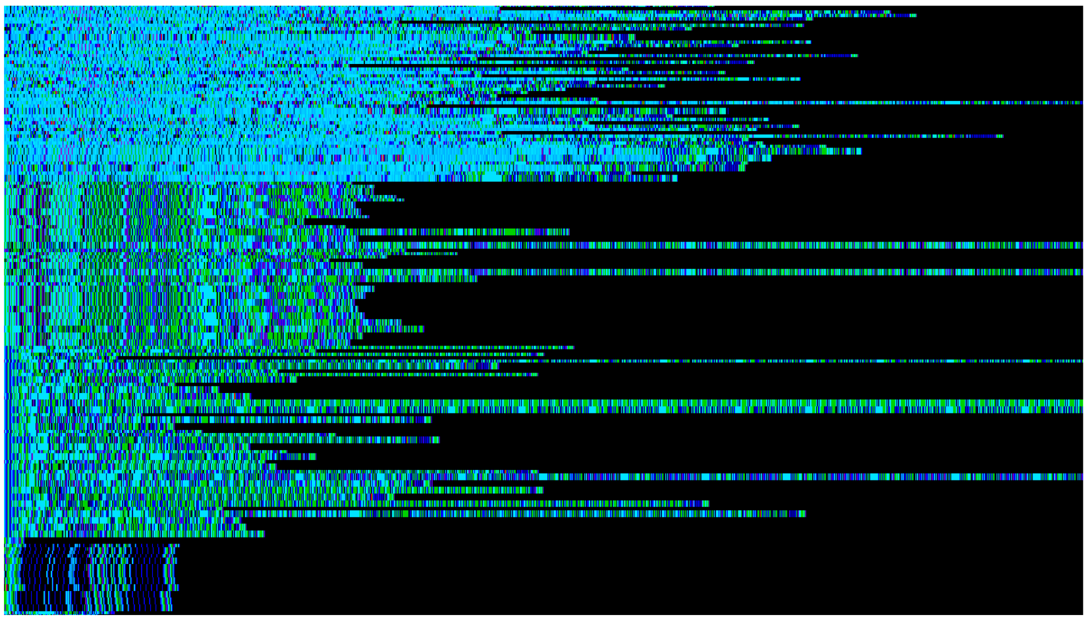


Figure 1.4. Visualisation of instructions from four different malware families (Ransomware Cryptesla, Emotet banking Trojan, Tofsee backdoor, Ransomware Cerber, respectively from top to bottom). Each horizontal line is colour-coded instruction sequence for each malware sample.

A crucial insight here is that humans can also identify many different patterns of the same kind based on a single sample. Human brain has a great generalisation capability by this one-shot training. Given the recent breakthroughs achieved in deep learning which mimic human neural network system, there is a significant potential that deep learning can be leveraged in detecting these highly complex non-linear patterns. For example, one-shot training has been recently introduced in deep learning space via meta-learning or generative adversarial network [25].

Recent studies show that anti-cognition techniques are increasingly problematic. Considering that many malware variants are created by a single metamorphic script, the metamorphism used by the variants spawned from a script exhibits coherent characteristics within several campaigns as depicted in Figure 1.4. Deep learning, if proven to work in one-shot training setting, is expected to be resilient to anti-cognition techniques because re-training the model with a single additional sample for new campaign requires little effort.

To enable the detection under production constraints, accurately extracting instruction sequence out of raw binary executables without 3rd party tools such as IDA Pro [26] is critical, which is a challenging task. In addition, many recent machine learning based detection techniques using raw binary executable as a feature have been found to significantly rely on statistical features rather than instruction-based semantic features, which also exhibited vulnerabilities to adversarial attacks. In tackling this semantic detection problem from raw binary executables, Transformer has shown a great potential that semantic signals based on instructions can be extracted using self-attention mechanism.

If deep learning can provide the aforementioned capabilities in this section, it is expected to cluster well under complex non-linear problem setting such as metamorphic malware samples. The hypothesis is that state-of-the-art neural networks can learn:

- A generalised representation coherent to malware campaign out of a limited training set
- A mapping from highly structured raw binary executable to semantic instruction signals

They also have a potential to provide a good clustering method for metamorphic threats.

1.3 Research Challenges

Due to the key malware problems described in section 1.1, a variety of machine learning based malware detection methods have been proposed over the last decade. However, many difficult challenges are still remaining. This section will present key research challenges.

Anti-cognition techniques such as metamorphism are identified as one of the primary challenges in malware detection problem. The task involves devising a method that automatically captures highly complex non-linear patterns present both statically and dynamically. Given that anti-cognition is a part of the arms race, it is also crucial to produce a long-term strategic method resilient to short-term evasive tactics.

In traditional machine learning regime, the underlying assumptions are as follows:

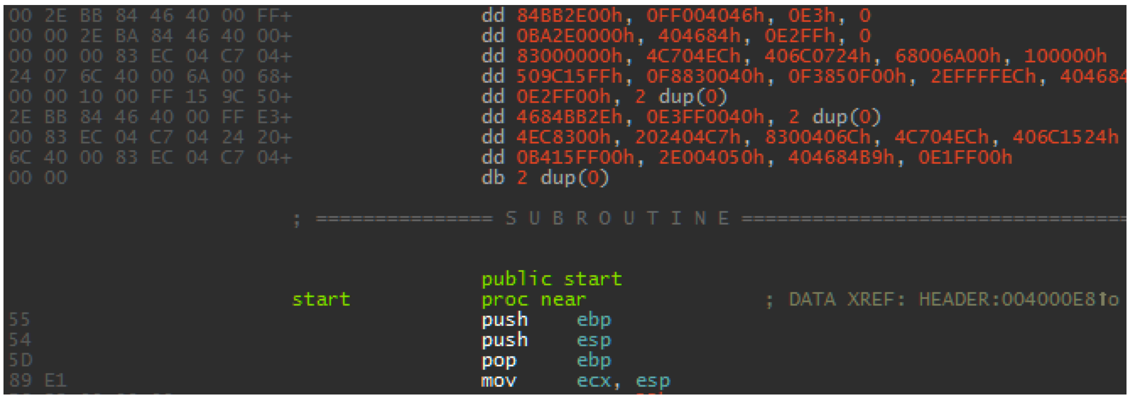
- A sufficiently large number of training samples exist
- Training set represents the underlying data distribution being modelled.

This problem setting works well against natural and continuous datasets with abundant training samples. However, given that the nature of the malware battle is arms race, malware distribution consists of a large number of very sparse groups. Therefore, none of the above assumptions for machine learning holds true. Almost all previous studies are based on these assumptions. The sparse sample distribution driven by the arms race poses a big challenge to machine learning based detection.

The ultimate goal of malware defence is to provide a maximum number of detections from a minimal number of known training samples. In in-the-wild malware campaigns, the very initial outbreak delivers the most valuable piece of information about the malware campaign. The concept of one-shot training is critical in detecting the malware variants that belong to the same malware campaign, which were generated by the identical metamorphic script. It is an extremely challenging task to estimate the data distribution of the campaign with sufficient accuracy using a handful number of training samples.

Resiliency against adversarial attacks forms a critical part of the malware defence. While metamorphism has been previously used to target manual signatures and traditional machine learning, the recent metamorphism is increasingly sophisticated especially with the advent of deep learning based detections. The previous approaches based on hand-crafted statistics and information-theoretic metrics such as N-gram have been countered with reinforcement learning with random mutation [27]. Moreover, many studies [28] [29] [30] [31] [32] [33] have proposed adversarial attacks primarily targeting the vulnerabilities of machine learning based detection methods. It is essential to provide anti-adversarial attack mechanisms when machine learning is used.

One of the common malware problems arises when the trained model is not designed to distinguish the legitimate instructions from data. A significant number of malware samples deliberately insert arbitrary amount of high entropy data in between the code fragments largely in an attempt to evade traditional detection methods (see Figure 1.5). Unfortunately this old tactic also poses yet another obstacle for the machine learning models utilising raw executables as a feature. This randomised data scattered within the code section works as a significant amount of noise that contributes to the incorrect decision. For this reason, the models without instruction cognitive capability will essentially make a random coin flip decision with a certain probability against the malware samples with this tactic. Forcing the disassembler to parse the packed data in Figure 1.5 produces logically insensible sequence of instructions along with intermittent invalid opcode exceptions.



```
00 2E BB 84 46 40 00 FF+                  dd 84BB2E00h, 0FF004046h, 0E3h, 0
00 00 2E BA 84 46 40 00+                  dd 0BA2E0000h, 404684h, 0E2FFh, 0
00 00 00 83 EC 04 C7 04+                  dd 83000000h, 4C704ECh, 406C0724h, 68006A00h, 100000h
24 07 6C 40 00 6A 00 68+                  dd 509C15FFh, 0F8830040h, 0F3850F00h, 2EFFFFECh, 404684
00 00 10 00 FF 15 9C 50+                  dd 0E2FF00h, 2 dup(0)
2E BB 84 46 40 00 FF E3+                  dd 4684BB2Eh, 0E3FF0040h, 2 dup(0)
00 83 EC 04 C7 04 24 20+                  dd 4EC8300h, 202404C7h, 8300406Ch, 4C704ECh, 406C1524h
6C 40 00 83 EC 04 C7 04+                  dd 0B415FF00h, 2E004050h, 404684B9h, 0E1FF00h
00 00                                     db 2 dup(0)

                        ; =============== S U B R O U T I N E =================================


                                          public start
                        start             proc near                ; DATA XREF: HEADER:004000E8↑o
55                                        push    ebp
54                                        push    esp
5D                                        pop     ebp
89 E1                                     mov     ecx, esp
```

Figure 1.5. A disassembly of a malware sample that has a large amount of packed data embedded within the code section.

Metamorphism lies at the heart of malware problem today. We are fighting against the diversity than anything else when it comes to malware detection. In addition, no research has been conducted in the context of metamorphism on the clustering of metamorphic samples. It is a challenging task to accurately cluster datasets that contain metamorphic groups.

1.4 Research Objectives

The goal of this research is to provide timely and accurate malware detection method resilient to anti-cognition methods and adversarial attacks. Inspired by the research challenges discussed in the previous section, the following key research objectives are formulated in this thesis:

Objective 1: Create a model that learns to statically detect semantically similar malware variants by identifying highly complex non-linear patterns created by various anti-cognition techniques, and by using one-shot training while remaining resilient to adversarial attacks.

Objective 2: Investigate the static detection model to dynamically detect semantically similar malware variants, discovering multiple heterogeneous malware families that share similar dynamic execution characteristics.

Objective 3: Develop instruction cognitive representation that disambiguates legitimate instructions from uninterpreted raw binary executables, and perform one-to-many malware detection using one-shot training while remaining resilient to adversarial attacks.

Objective 4: Conduct comprehensive comparative studies of various machine learning methods on their capabilities in clustering the variants of malware families that show similar characteristics in their core instructions.

1.5 Contributions

With regard to the research objectives formulated in the previous section, Figure 1.6, Figure 1.7, and Figure 1.8 illustrate the overall research contributions made in this thesis in relation to the first three objectives. The main contributions of the thesis are briefly described below.

- **Generative Outbreak Detection Model**

To fulfil Objective 1, a generative Adversarial Auto-Encoder (AAE) (Figure 1.6) is developed. This novel method generalises malware detection using a single training sample from each outbreak. Adversarial autoencoder is trained over prepared instructions to reach Nash equilibrium in a non-cooperative minmax game, producing smooth approximated nearby representations for scarce number of training samples. This unique one-shot problem setting is not aligned with traditional arrangement of large training set, which makes practical sense considering the ultimate goal is to maximise the ratio of malware variant detection against known training malware samples. The model shows near-production level performance, outperforming all baseline traditional machine learning models. A paper has been published [34] on this work.

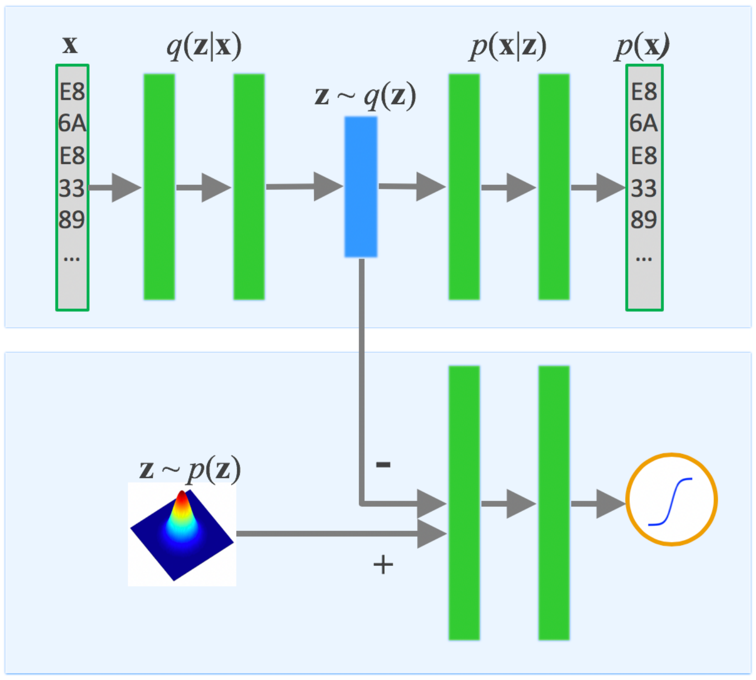


Figure 1.6. Adversarial autoencoder architecture used for malware outbreak detection. The input, **x**, and the reconstructed input, *p*(**x**) are instruction sequence feature.

- **Spatio-Temporal Isomorphic Dynamic Outbreak Detection Model**

To address Objective 2, DCGAN [35] was deployed to handle multi-dimensional dynamic API call events on top of adversarial autoencoder at its foundation. The model is jointly trained with stochastic gradient descent by minimising the reconstruction loss on spatio-temporal input x over latent representation z (Figure 1.7), which allows resiliency over various perturbations introduced in dynamic execution such as multi-threaded events and missing events caused by anti-cognition techniques and sandbox limitations. The model has been further extended to cope with variable length inputs, allowing arbitrary sequence of API call events by utilising global average pooling over the last convolutional layer. A paper has been published [36] on the outcomes of this work.

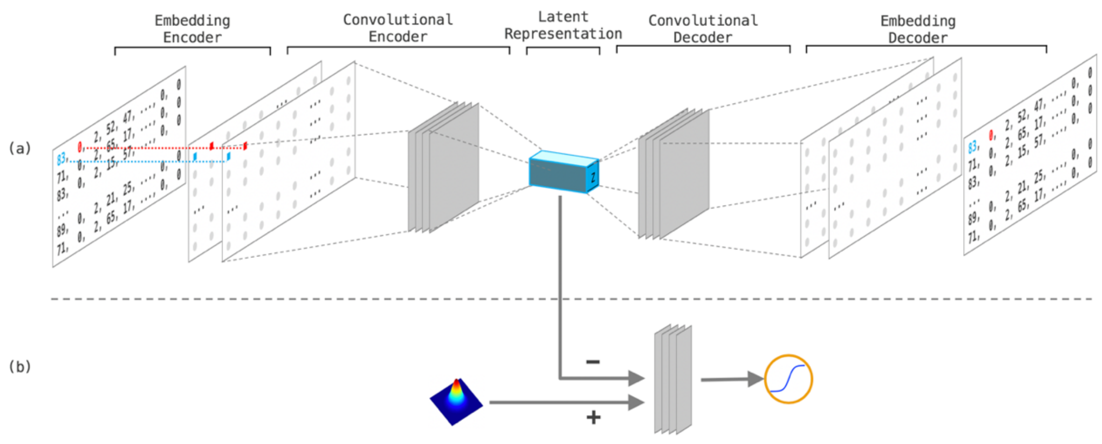


Figure 1.7. Generative adversarial autoencoder with 2D API call event feature and embedding. (a) Autoencoder with each value in 2D feature mapped to an embedding. Note that the value in blue gets mapped to an embedding of size two at its corresponding position marked in dotted blue line. The embedding value is chosen from the embedding lookup table for each symbol at the input feature. The same applies to the value in red. (b) Discriminator with positive samples from Gaussian normal distribution and negative samples from the latent representation, z, obtained from the input feature.

- **Instruction Cognitive Detection Model**

To fulfil Objective 3, a deep learning model based on Transformer is trained to produce instruction cognitive representation that can directly transform the raw binary executable into a sequence of legitimate instructions while generating empty signals for data regions. The model is resilient to various adversarial attacks against machine learning detections which are based on raw binary executables. The model has been demonstrated to perform static malware detections using the Fourier transform of the instruction cognitive representations, showing its capability for one-to-many detection (Figure 1.8). A paper has been published [37] on this contribution.

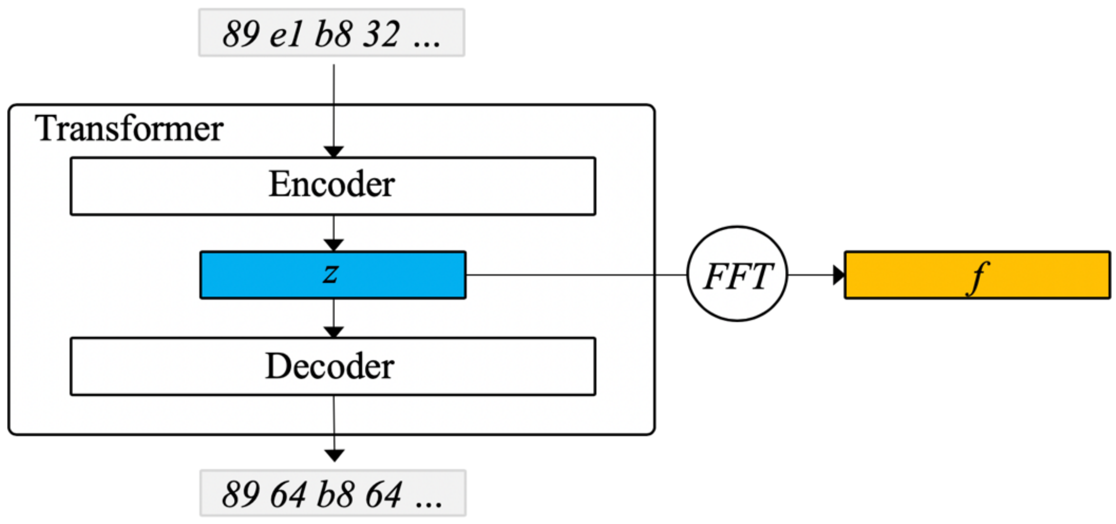


Figure 1.8. Model architecture using Transformer. Frequency spectrum of approximated encoded latent representation is used as the feature for malware detection.

- **Comparative Study on Clustering Metamorphic Malware**

To fulfil Objective 4, a comparative study on various machine learning methods has been conducted over interpretable family-wise metamorphic malware samples that possess similar instruction-wise characteristics instead of arbitrarily chosen malware dataset from the black box. Several top-performing clustering models from representative domains have been evaluated, including KNN, HDBSCAN, SDHASH, and AAE. The evaluation has been conducted with appropriate benchmarking criteria that fairly tests different models. Studies have uncovered which model best generalises the distribution with metamorphism in place. The study has been submitted and accepted [38] as a book chapter.

1.6 Structure of the Thesis

This dissertation is organised as follows:

Chapter 1 gives overview, research motivation, research objectives and thesis structure.

Chapter 2 reviews a number of existing studies on malware anti-cognition techniques and detection methods. The review covers deep learning studies relevant to the research objectives. In addition, various detection approaches specific to machine learning and deep learning are reviewed with pros and cons.

Chapter 3 proposes a generative malware outbreak detection model that detects similar malware variants within a campaign using a single training sample from each campaign. It explains how to apply one-shot learning in the static malware detection context. The efficacy of the model is validated against various supervised machine learning models.

Chapter 4 extends generative detection approach from Chapter 3 by generalising the inputs to handle multi-dimensional features from dynamic execution and variable length event sequence. The model's efficacy on cross-family as well as intra-family similarity detection is tested against OS X malware dataset.

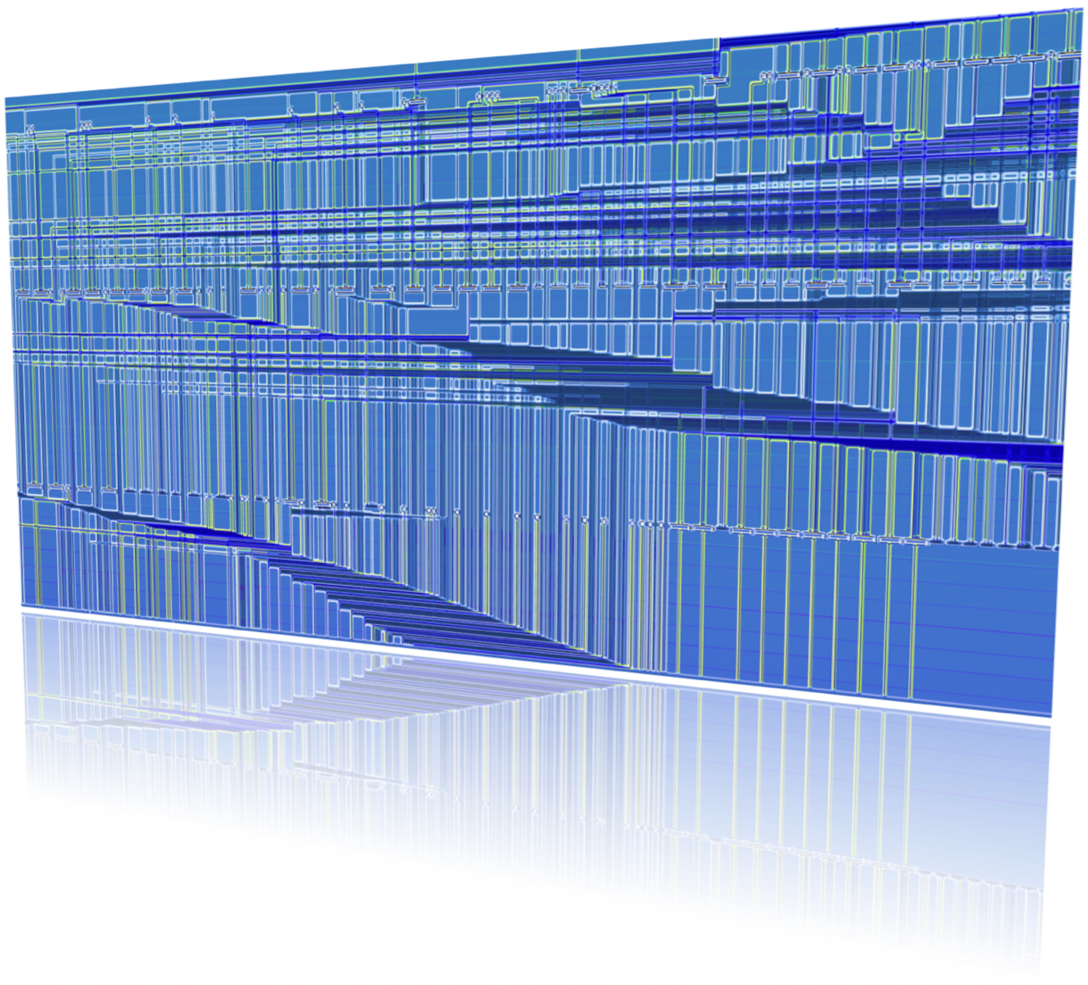
Chapter 5 introduces a novel method of extracting instruction cognitive signals by cleverly taking advantage of Transformer's sequence-to-sequence mapping capability with self-attention mechanism. The model is validated with the feature Fourier-transformed from instruction cognitive representation.

Chapter 6 provides the significance of clustering capability on metamorphism followed by the increasing need for threat hunting under current threat landscape. It provides a comprehensive comparison among several key clustering methods.

Finally, Chapter 7 provides some concluding remarks on the efficacy of the research undertaken and outlines some possible research directions based on the findings in this thesis.

The essence of the malware problem is discussed in the following chapter through a comprehensive review of previous studies on a variety of malware techniques and detection methods based on machine learning.

Chapter 2: Literature Review



In the history of malware battle, understanding what lies at the heart of the arms race is crucial in order to stay on top of the fight. This chapter discusses the key malware strategies, recent deep learning development, and various approaches taken by traditional machine learning and deep learning methods. The significance of a semantic feature is also discussed by reviewing many previous studies on detection over raw binary executables and its resiliency against adversarial attacks. This chapter concludes with a discussion on metamorphism and key research challenges. Overall diagram of literature review is shown in Figure 2.1.

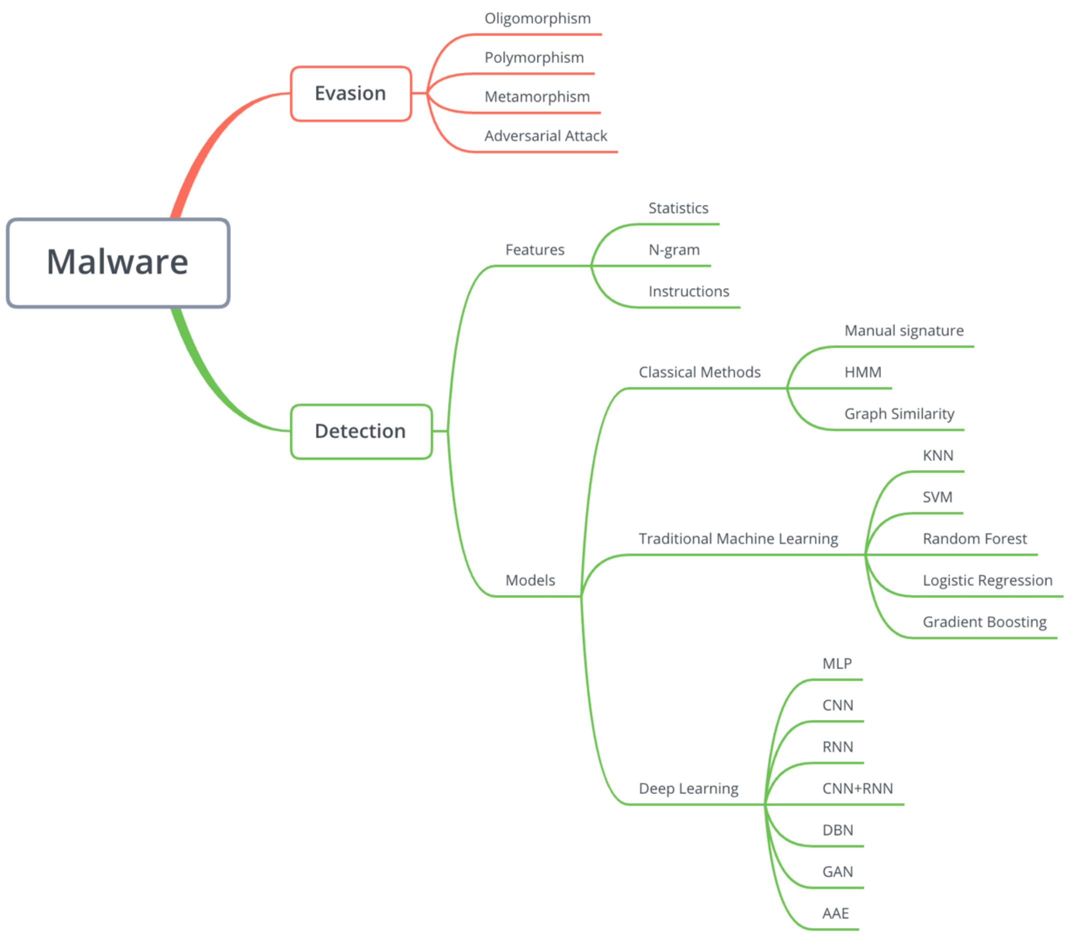


Figure 2.1. Overview of literature review

2.1 Malware Evolution

In this section, the evolution of core malware techniques is described along with several key malware analysis techniques to tackle them.

Guo et al. [3] conducted a comprehensive study on the packer problem where combinations of multiple different custom packers create a significant amount of diversity, which increases the complexity in writing detection signatures. The study also highlighted several techniques that help automatically acquiring the original unpacked binary code through the use of dirty page execution, unpacker memory avoidance, stack pointer check, and command line argument access. Roundy et al. [4] provided a more comprehensive survey on various anti-cognition techniques, primarily focusing on the packers and unpacking mechanisms.

Sharif et al. [39] proposed an idea to defeat the malware analysis that captures the trigger-based malware behaviour, by encrypting the code conditionally dependent on the input. The authors showed that the approach is capable of concealing the malware behaviour by obfuscating the key, which hinders deep malware analysis including malware unpacking.

Royal et al. [40] showed a novel method, PolyUnpack, to extract the hidden code from a packed executable file based on a simple concept. The algorithm keeps track of the execution of the program instruction by instruction, and iteratively identifies the hidden code when the current instruction in the memory is absent in the statically identified instruction set, which indicates that the unpacked code is being executed.

In 2008, Borello et al. [5] studied on various ways to generate metamorphic code, which summarised the anti-cognition trend and later became a foundation of future malware anti-cognition researches. The authors highlighted the difficulty of static malware detection in the presence of metamorphism, posing an N-P complete problem. Several key anti-cognition methods were identified in this work, including instruction substitution, instruction permutation, variable substitution, dead code insertion, and control flow modification. In 2010, You et al. [41] further elaborated on malware obfuscation techniques explaining the brief history of obfuscation from mere encryption, oligomorphism, polymorphism, and finally metamorphism. The study shows a wider spectrum of metamorphism in different scales of code. Later, Rad et al. [42] provided a more concise explanation from encryption to metamorphism.

In an attempt to defeat metamorphic malware, Wong et al. [43] proposed a novel method that utilises Hidden Markov Model (HMM) against the opcode sequence of the program, providing the likelihood per opcode (LLPO) as a score measure. The model demonstrated its efficacy with better capability to detect similar malware samples that exhibit similar LLPO. However, this method was soon defeated by Lin et al. [44] who used mere junk code insertion obfuscation.

Runwal et al. [45] also proposed a novel method using opcode graph similarity in order to tackle the metamorphic malware problem. The approach takes straightforward Euclidean distance between opcode transition probabilities within the graph as a similarity measure. The authors also suggested potential counter-attacks on the approach such as uncommon opcode removal and random dead code insertion.

The core strength in evasive malware lies in diversity, which is primarily driven by metamorphism in the recent trend. Although early detection attempts described in this section include clever tricks such as HMM and graph similarity, a more sophisticated approach is in need to capture the complex non-linear relationship buried within the executable file where deep learning has a strong potential. A brief description of deep learning techniques and their applications in malware detection are presented in the following sub-sections.

2.2 Deep Learning

Deep learning [46] is a group of methods that train neural networks with multiple layers by backpropagation [47], which iteratively minimises the loss function using stochastic gradient descent [48]. Deep learning arguably forms the backbone of the modern artificial intelligence today. There has been a surge of research interests in deep learning in the early 21st century, driven by a series of successes mainly in natural language processing (NLP) and computer vision. In 2006, Hinton et al. [49] created a computationally efficient and yet deep neural network called, Deep Belief Network (DBN), which ignited the resurgence of research in the field of neural networks. Although DBN’s performance is superseded by many recent algorithms, it possessed many critical attributes of modern neural networks such as energy-based learning and unsupervised generative modelling.

Long Short-Term Memory (LSTM) [50] has been widely used for sequence modelling such as machine translation and natural language processing. LSTM learns via backpropagation through time using several memory gates of the recurrent neural network (RNN). It is designed to perform supervised tasks against both continuous and discrete sequence inputs. Later, more efficient RNN, Gated Recurrent Unit (GRU) [51], was proposed with less number of gates. Graves [52] further extended LSTM applications to multidimensional data such as images and videos. Despite its success, RNN suffers from vanishing gradient problem [53] for applications with long range dependencies during backpropagation through time, which makes it impractical to use in long sequences such as program instructions. However, RNN has proven to work well for short sequences such as word prediction and document sentiment analysis. In cybersecurity field, RNN can detect URLs of Domain-name Generation Algorithm (DGA) [54] used to obfuscate the Command and Control location on the internet.

Another big stream of research in deep learning is based upon Convolutional Neural Network (CNN) [55] that possesses several characteristics similar to the human visual cortex system such as local receptive fields, shared weights, and spatial subsampling. CNN's resiliency to local distortion and shift provides a firm foundation of high precision classification and clustering for multi-dimensional data such as images and videos. Due to this resiliency to variations, at the core of malware detection models proposed throughout the chapters in this thesis exists CNN.

Goodfellow et al. [56] invented a clever Generative Adversarial Network (GAN) where non-cooperative generator and discriminator are jointly trained in a minmax game with its goal to reach Nash equilibrium [57], the state where both the generator and the discriminator cannot reduce the error any further. GAN is the foundation of modern artificial intelligence, producing many state-of-the-art results in many fields including realistic image generation, neural style transfer, cross-domain transformation, image super resolution, and pose generation. GAN is a generative model where its goal is to produce realistic data resembling the training data. Its capability to capture highly complex non-linearity present in the data has proven to work well for semi-supervised [57] and purely unsupervised tasks [58].

Despite its success, GAN is known to be difficult to train. Vanilla GAN is trained by performing gradient descent on approximated Kullback-Leibler (KL) divergence, which does not always converge, exhibits model collapse producing limited varieties of samples, and results in overfitting due to the imblance caused by the joint training. Many approaches have been invented to improve the GAN training such as f-GAN [59], instance noise [60], and Wasserstein GAN [61] at the optimisation level. Seeing the simultaneous gradient descent performed by GAN as non-conservative vector field, Consensus optimisation [62] suggests a general method to find the Nash equilibria by optimising with the vector field combined with the gradient of loss. The malware detection models proposed in this thesis use Consensus optimisation for this reason. GAN provides a general framework for a large number of neural network models. Naturally Radford et al. [35] created a deep convolutional network using GAN, DCGAN. DCGAN served as a basis for a wide variety of applications that deal with multi-dimensional data. Due to the dimensionality of the input used in this thesis, the overall DCGAN architecture is deployed in the deep learning models proposed in this thesis.

Autoencoders [63] play a fundamental role in unsupervised learning that leverages deep learning. The purpose of Autoencoder is to produce a compressed representation, z, of the input data while eliminating the noise in order to achieve dimensionality reduction. z is then used for distance calculation during clustering or similarity search. Autoencoder is trained with a single objective of minimising the reconstruction error. Earlier implementations such as denoising autoencoder (DAE) [64] used dropout [65] as an approximation of Bayesian inference to deal with uncertainty at the input. Multi-dimensional data with local spatial relationship works better with stacked convolutional autoencoder (CAE) [66] since the architecture is simply aware of the structure of the input. Kingma et al. [67] proposed variational autoencoder (VAE) using stochastic variational inference. Makhzani et al. [58] proposed an autoencoder using GAN framework, called adversarial autoencoder (AAE). AAE is jointly trained with GAN's adversarial objective along with regular reconstruction minimisation objective. The regularisation performed by adversarial objective guides the aggregated posterior distribution, $p(z)$, produced by the encoder to match the chosen prior distribution, which is normally random normal distribution. This extra adversarial regularisation in AAE turns out to be effective for one-shot training that will be explained in Chapter 3. The compressed representation on which the prior, *p(z),* is imposed captures information well even for an extremely small number of malware training samples.

Although deep learning has proven to work well with continuous data such as speeches, images, and videos where gradients can be propagated back to the direction guided by the optimisation objectives, it is difficult to define the gradients for discrete data such as symbols that contain highly complex relationships in between them. The ability to train over discrete symbols is critical in applying deep learning to cybersecurity problems including malware detection due to the fact that the source of input is nearly always in the form of symbols such as machine instructions, programming language statements, or command line commands. One-hot encoding [68] has been extensively used to deal with this problem. However, this method is neither computationally scalable for training when the number of independent categories goes high nor allows backpropagation since the mapped representation in the model space is not close to each other for similar data points. Guo et al. [69] proposed an entity embedding where the mapping is learned as part of standard supervised learning process. This approach overcomes the limitations of one-hot encoding including high cardinality feature problem and the lack of distance measure in the model space.

Combining the idea of unsupervised learning with discrete inputs, a discrete autoencoder is a natural approach in deep learning. While discrete data is nicely handled with entity embedding, sequence to sequence modelling required for autoencoder reconstruction error calculation is challenging. RNN for this problem is suboptimal for this task due to the vanishing gradient problem against long sequences. Zhang et al. [70] proposed a simple yet clever technique for this problem using pure convolutional and deconvolutional autoencoder with entity embedding for discrete symbols. The approach combines the entity embedding with standard convolutional autoencoder with reconstruction error computed by the Cosine distance between the original embedding and the reconstructed output. This trick enables unsupervised learning to be applied against many symbolic inputs that arise in cybersecurity problems.

While Autoencoder produces a compressed latent representation, z, of the data obtained through unsupervised learning, we need a mechanism to compare different values of z to perform similarity search. Semantic hashing [71] provides an approximation of the latent representation that can be used with Hamming distance [72] for efficient distance comparison. The computational efficiency of semantic hashing can be a great boost for practical systems with the sacrifice of minor accuracy.

In recent years, many innovative ideas on how deep learning can be utilised apart from straight forward classification and clustering tasks have been explored. Image-to-image translation using conditional adversarial GAN [73] demonstrated impressive image transformations such as photo synthesis from labelled maps, aerial to map conversion and edges to photos. Zhang et al. [74] demonstrated the use of GAN to generated realistic photos from textual descriptions. Cross-domain image transfer proposed by Zhu et al. [75] shows great results on image style changes using CycleGAN. Super-resolution [76] and image inpainting [77] are some of the new areas where GAN shows impressive results. DeepFake [78] produced some of the realistic and yet fake videos constructed based on CoupledGAN.

In the NLP space, Vaswani et al. [79] proposed a novel sequence transduction architecture called, Transformer, that is solely based upon attention mechanism without any recurrence or convolution layers. Vanishing gradient problem has been one of the significant drawbacks of sequence models. The key breakthrough made by Transformer is the use of self-attention mechanism that enables to learn the long range dependencies by allowing the efficient traversal of intra-embedding signals. A pre-trained deep bidirectional Transformer, called BERT [80], shows state-of-the-art results for various sequence modelling tasks such as language inference. This thesis takes advantage of Transformer's cross domain transformation capability to identify signals of instructions from highly structured raw binary executable files.

Despite a number of remarkable successes, deep learning as well as traditional machine learning have not yet demonstrated the capability to logically understand the complex algorithms let alone primitive arithmetic. For instance, Kaiser et al. [81] proposed an algorithm that learns basic arithmetic operations such as addition and multiplication with a neural network architecture called, Neural GPU. The model is capable of performing other algorithmic tasks including copying sequences, reversing sequences, duplicating sequences, and sorting bits. Essentially the model adopted a deep recurrent neural network that consists of convolutional gated recurrent units, combined with clever learning tricks such as parameter sharing relaxation, dropout, and gradient noise. Price et al. [82] further improved the generalisation ability of neural GPU by tweaking the model hyperparameters and the training algorithm. Nonetheless, no machine learning model to date including neural GPU and its descendants have proven to perform this basic arithmetic tasks with 100% accuracy, which a human child with basic math education can achieve with perfect accuracy. This strongly suggests that machine learning is not at the level of understanding the semantics of many logical problems yet. Complex conceptual inference tasks such as malware reverse engineering and deep semantic program analysis are beyond the capability of deep learning today.

Traditional machine learning methods typically rely on hand-crafted statistical features whereas deep learning lets the model automatically discover non-linear features out of the feature-rich original data. In recent years, deep learning has shown many different ways to leverage machine learning including generative models and cross-domain transformation. Due to these major breakthroughs, there is a high potential that deep learning can be leveraged in malware detection problem where the perception of complex non-linear pattern is required.

2.3 Traditional Machine Learning Methods for Malware Detection

There have been many malware detection approaches using traditional machine learning. Gandotra et al. [83] provided a compact summary of the major challenges of malware threats and machine learning based detections in response to them. Bazrafshan et al. [84] conducted a survey on various evasion methods as well as heuristic malware detection methods, discussing individual features and machine learning approaches. Narudin et al. [85] also conducted an evaluation of various machine learning based malware detection using MalGnome Project samples. Nath et al. [86] provided a nice summary of recent malware types and trends in addition to various features used by machine learning methods.

Gavrilut et al. [87] implemented rudimentary perceptron iteratively trained with primitive weight update algorithm. Although it failed to address many critical aspects of malware detection problem, the algorithm shows the fundamentals of modern machine learning based malware detection. Raman et al. [88] showed how statistical features extracted from the executable file header can be used for malware classification task. The authors proposed a Random Forest with simple feature selection method that iteratively adds the feature with higher accuracy in the previous step.

Metamorphism constitutes one of the core problems in malware detection. Focusing on the predictable opcode patterns identifiable from metamorphic malware, Santos et al. [89] proposed a method that takes advantage of opcode histogram as a feature and cosine distance as the similarity metric. The feature vector is created by selecting the most relevant opcodes using Mutual Information followed by calculating the vector's weighted term frequency. Despite its simplicity, this research successfully spotted the significance of instructions as a more semantic feature in tackling the heart of the modern malware detection problem. Santos et al. [90] further extended the idea of opcode sequence feature by deriving weighted term frequency features calculated by Mutual Information [91]. They identified polynomial kernel classifiers and decision trees as the best performing models while Bayesian networks yielded high false positives.

Although all the above approaches show the fundamentals of how machine learning can be leveraged in malware detection problem, they significantly lack the capability to deal with the complex diversity of malware.

Santos et al. [92] argued the problem of machine learning researches geared towards a large labelled training dataset. The authors utilised Learning with Local and Global Consistency (LLGC) that learns the smooth transition of underlying intrinsic structure based on both labelled and unlabelled samples. Using the byte N-gram as a feature, the authors showed that the semi-supervised learning contributed to a reduction in the number of required labelled samples when detecting unknown malware. As far as we know, this is the first research that highlighted the significance of unknown malware detection out of a small number of known malware samples, which is one of the key aspects of this thesis.

Rieck et al. [93] approached the problem of unknown novel malware detection by combining similarity based clustering with class assignment tasks. The authors proposed a novel feature representation called, malware instruction set (MIST), which enables expressive characterisation of malware behaviour. The authors further describe how to map MIST into vector space suitable for machine learning by way of embedding using instruction Q-grams and normalised embedding function. The embedding definition provided by Rieck et al. not only enables geometric distance comparison but also offers explicit vector representation where causal inference of the decision is possible. In order to allow clustering, the authors proposed an iterative algorithm that discovers a prototype in a group that exhibits homogeneous behaviour. The authors further proposed a nearest cluster identification method for classification purpose in the environment where the labels are not reliable. The authors, then, showed how to determine the malware behaviour from the combination of novel clustering and classification algorithm, by incrementally identifying the belonging classes in an iterative loop.

Yerima et al. [94] proposed a Bayesian based classification method for static zero-day Android malware detection. Features are extracted from APK permissions and code-based properties, and prominent features are selected using information gain method. The approach decides the maliciousness based on Bayes theorem over selected features. Notably the result shows that the number of features ranked by information gain heavily affects the model performance. A method to update the model with regression test is desired in order to adopt this approach in the field.

Anderson et al. [95] proposed graph similarity based malware detection using the instructions traced with a dynamic instrumentation tool. The authors converted instruction traces into Markov chain of instructions with transition probabilities using the adjacency matrix. Unique instruction identifiers were used while eliminating the operands, which removes the sensitivity to register allocations and compiler artifacts. This approach is much more fine-grained than utilising statistical features such as N-gram. Graph kernel, combining Gaussian kernel and spectral kernel, is computed to find the similarity of given two instances of a sample. The authors used SVM to perform the classification. The result shows that the Markov chain with combined graph kernels outperforms N-gram based detections by a large margin. Although the proposed method is novel with good performance, the computational complexity is prohibitive in a real-time setting.

Xu et al. [96] focused on detecting the kernel rootkits that implants hooks and the execution of exploits on the vulnerable application process such as return-oriented programming (ROP). Given that a combination of these activities mostly occur at the beginning phase of infection, this unique approach utilising the memory access patterns is valuable research. The underlying idea behind the approach is that the control flow and data structure of a legitimate application process will be convoluted once it is compromised, which will lead to the changes in memory access patterns. Since modern processors can run billions of instructions per second, collecting the memory access patterns is one of the key challenges in this work. The authors addressed this problem by collecting the histogram of each memory region access at function call level with hardware-assisted framework. The result for kernel rootkits shows that Random Forest records 100% detection rate with less than 1% false positives. This is sensible considering the memory changes made by the kernel rootkits are concentrated to a dedicated region of kernel virtual memory space that leaves a unique signature. For user level memory corruption attacks, the best performing model, Random Forest, against SVM and Logistic Regression also recorded 99% detection rate with less than 5% false positives. In short, the authors introduced a novel approach that detects common infection scenarios occurring within a local machine by leveraging the distinguishable memory access patterns.

Jacob et al. [97] explained the efficacy of behavioural signature based detection over static counterpart. The authors conducted a survey of different reasoning techniques They divided the malware detection approaches into simulation-based verification and formal verification, which are directly connected to dynamic and static detection, respectively. The authors concluded with the necessity of a common model of reference that combines both dynamic and static modes.

Given that both static and dynamic views of the sample exhibit different aspects of underlying characteristics, it would be a good idea to consolidate both features. Islam et al. [98] attempted the malware classification based on integrated static and dynamic features. Shijo et al. [99] proposed a similar method where the feature vector is created by combining the frequency of API call strings and N-gram of behavioural API calls. SVM and Random Forest models were tested with integrated features against individual features. The result shows the integrated feature has higher accuracy than the one for individual features.

A sample in malware detection problem is a binary executable file that contains program logic along with metadata. The syntactic structure of the program logic is best represented by a function call graph (FCG). Kong et al. [100] proposed a classification model based on similarity distances of attributed FCGs. The model is trained by performing expectation maximisation that iteratively updates the model parameters and graph matching matrix that consists of matching nodes. The authors used a clever optimisation method that maximises inter-class distance while minimising intra-class distance for each attribute type using maximum margin principle, which dictates that the malware in the same family be closely clustered while clusters formed by different malware families have large margins to separate them. The authors then take the ensemble of classifiers for each attribute using Adaboost algorithm that iteratively assigns higher weights to those instances misclassified previously. Later, the model is trained to learn the confidence level that minimises the error of the classification results obtained through either SVM with Gaussian kernel or k-nearest neighbour (KNN). They found ensemble learning has the advantage of finding the best performing individual classifier even though the benefit by taking ensemble is not significant. The classifiers are then extended to deal with zero-day malware and benign legitimate samples by introducing new clusters based upon KNN classifier. The results generally show decent performance. However there was noticeable misclassification between Rbot and Sdbot, which is largely caused by different naming for essentially the same malware family. Kong's approach essentially builds up the knowledge of classes by combining the knowledge of clusters based on the similarity of syntactic and semantic features. This method is conceptually identical to several papers that constitute our thesis.

Sahs et al. [101] proposed a one-class SVM based Android malware detection using permissions and control flow graphs as a feature, given that substantially large number of benign samples are available than that of the malicious samples. The spirit of this research is aligned with semi-supervised approach taken by Santos et al. [92].

Comar et al. [102] proposed a method that detects novel malware using statistical features derived from layer 3 and layer 4 network flow data. The approach attempted to detect zero-day malware using one-class support vector machine (SVM) with for each type of malware, which found that supervised weighted linear kernel outperforms RBF kernel and Random Forest on this task. It is a novel approach to use per-malware one-class classifier in an attempt to probabilistically determine the similarity of a given sample towards each malware category, thereby detecting any zero-day malware. However, even the best hyperparameter shows below 50% F1 score, which is nowhere near comparable with the latest models.

Ki et al. [103] proposed a malware detection method based on API call sequence clustering. The authors used common API call sequence as malware signature, which was extracted using Longest Common Sub-sequences (LCS) algorithm. The authors first grouped samples through a sequence alignment technique called, Multiple Sequence Alignment (MSA). Then they profiled the critical API call sequences such as DLL injection, IAT hooking, Anti-debugging, and Screen Capture to construct the signature database, against which test API call sequence is matched. However, taking hand-crafted API call sequences as behavioural signature is vulnerable to commonly used obfuscations such as bogus API call insertion.

In the study on the efficacy of feature parameters against malicious behaviour detection, Canali et al. [104] suggested a systematic way of extracting common signatures of a given dataset, which consists of atom, structure, and cardinality of a signature. The authors considered behavioural operations (e.g. system calls) as signature atom while a combination of N-gram and bags were used to represent signature structure. The authors tested 215 behavioural detection models using the defined representations, with the best results produced by 2-bags of 2-tuples of actions with arguments, which detects 99% detection accuracy against unknown malware samples with 0.4% false positives. The authors concluded that different data models should be analysed by a comprehensive experimental evaluation because there is no general rule that dictates the relationships between different configuration of the data model.

Egele et al. [7] provided a gentle introduction to dynamic malware detection by conducting a comprehensive survey on various mechanisms of collecting the dynamic event traces, a diverse set of sandbox tools with detailed setup instructions, and dynamic malware detection methods.

Firdausi et al. [105] tested several state-of-the-art machine learning techniques to develop a proof-of-concept malware detection method based on behavioural events collected from sandbox. The experiment shows that J48 decision tree using term frequency-weight without feature selection outperforms KNN, SVM, and multi-layer perceptron (MLP). The best result of 97.3% precision and 95.9% recall suggests that the approach fits to threat hunting rather than practical detection system. Faruki et al. [106] constructed features using API call N-grams out of call traces obtained from Cuckoo sandbox [107]. The authors experiments with various classifiers available in WEKA [108] including Random Forest, Sequential Minimal Optimisation, J48 decision tree, Naïve-Bayes, and Voted Perceptron. The result shows that Voted Perceptron with API call tri-gram records the best performance. These two approaches essentially used API call sequences as a feature against a large corpus of training samples for classification tasks.

Although API call trace was used, the method used by Tian et al. [109] takes a slightly different approach from Faruki's [106] in feature construction. The feature vector is created by taking both the local and global frequency of each API call. The evaluation was performed using WEKA with Random Forest as the best performing model.

Wu et al. [110] introduced the use of clustering in dynamic malware detection for Android, utilising K-means and EM (Expectation Maximisation) algorithm. The authors used metadata from application's manifest files as well as API call traces as feature. The proposed method first clusters using K-means and expectation maximisation (EM) with the number of clusters determined by Singular Value Decomposition (SVD) on the low rand approximation. Then it determines the maliciousness by computing the distance to the reference samples. The experiment shows K-means combined with KNN outperforms other models.

Unlike previous approaches, Ahmed et al. [111] proposed spatio-temporal feature set as a sophisticated data model for the feature. To overcome the dynamic detection evasion through garbage calls, the authors considered the entire call trace as a whole rather than focusing on the statistics of individual API calls. Spatial information is created by taking statistical and information theoretic measures for the API call arguments. Temporal information is created by implementing a discrete time Markov chain represented by the probability transition matrix for the call sequence. Information Gain was used for classification determination. A 3rd order Markov chain was chosen based on the insight from sample autocorrelation function that indicated the 3rd order dependence. The authors also investigated minimal feature subset in order to accommodate large scale training and prediction.

2.4 Deep Learning Methods for Malware Detection

In this section, prominent approaches for static and dynamic malware detection based on deep learning methods are discussed.

Li et al. [112] experimented with Deep Belief Network (DBN) where stacked autoencoder using Restricted Boltzmann Machine (RBM) [113] is trained with layer-wise greedy algorithm followed by standard binary classification with softmax cross entropy at the output layer using backpropagation. RBM performs dimensionality reduction that converts complicated high dimensional data to low dimensional latent representation with non-linear mapping. The approach recorded 92.1% accuracy over KDDCUP'99 dataset [114]. Although it did not use layer-wise noises for autoencoder and the dataset consists of handcrafted sparse features, it demonstrated the efficacy of DBN. Nonetheless the result is far from sufficient to be useful in practical detection systems.

David et al. [115] demonstrated the use of DBN for behavioural malware detection over a relatively fine-grained number of target classes. The model was trained layer-wise with dropout to generalise the classification to compensate for the use of the relative small dataset. The authors have applied deep denoising autoencoder in a straight forward way, using the produced latent representation as signature. Although this technique does not deal with complex malware problems such as one-shot training and metamorphism, the approach is novel in the sense that unsupervised deep learning was first leveraged in generating the signature.

Saxe et al. [116] approached the malware detection problem by treating it as a binary classification problem, using a deep neural network of fully connected layers. They used a variety of handcrafted statistical features from the executable file. For example, byte entropy histogram feature they used contains a set of sliding windows of entropy histogram spanning the entire binary executable file, which approximately separates the compressed or encrypted portion from legitimate instructions and data. Other features such as hashed import table entries are the metadata from the executable header. The model consists of two fully connected layers with a single sigmoid output layer over an aggregated input constructed from the described feature set. The model also uses dropouts to improve generalisation, which also addresses overfitting problem to a certain extent. The authors defined *threat score* derived from the original classifier's output score by incorporating the confidence level. Bayesian rule is applied using probability density functions (PDF) of scores given the classifier's output prediction. The authors used kernel density estimator (KDE) with Epanechnikov kernel to derive these PDFs in order to calculate the threat score. The experiment was conducted with 4-fold cross-validation, achieving an average of 95.2% detection accuracy and 0.1% false positives. The authors then performed a time split experiment to estimate the model's capability of detecting the novel malware and the variants of previously known malware, which showed significant degradation in performance. As the authors pointed out, the classifier should be retrained frequently to incorporate the changes introduced by the evolving malware variants.

Dahl et al. [117] addressed the difficulty of training classifiers against large sparse binary features. The primary contribution of this work is the application of a very sparse random projection technique [118] on high dimensional input data using sparse projection matrix, which significantly reduces the dimensionality that allows the processing by the following classifiers. For classifiers, the authors experimented with multinomial logistic regression and deep neural network over 136 target classes, each of which corresponds to malware families. In a nutshell, deep neural networks trained on random projections produced a 43% reduction in error rate compared to the baseline logistic regression that uses all sparse features. The authors also investigated PCA-based input dimensionality reduction, which produced significantly worse result let alone its computational complexity of $O(N^3)$ is prohibitive. Randomised PCA algorithm was chosen for this experiment due to the difficulty of computing singular value decomposition (SVD) on the data’s covariance matrix. For binary classification setting, the larger number of random projection dimension generally tend to produce a better detection result.

Using convolutional neural network, Davis [119] first adopted raw binary samples as features, which contains program code in it. However, the binary distribution of a malware outbreak can significantly vary depending on the packed code and data, and the layout of sections can change with little effort by the attacker. Using highly structured content such as an executable file as a sequence of raw bytes is less likely to generalize the distributions of malware and its variants.

Park [120] argued that on-time malware detection failure is primarily due to the diversity of the malicious code, which is achieved through metamorphism. The author points out that metamorphism messes up data distribution when the program instructions are used as a base feature. The author introduced a method that detects malware families through the metamorphic patterns using stacked de-noising autoencoder and semantic hashing. Fourier transform was demonstrated to capture a wide spectrum of metamorphic instruction patterns used in different malware families. Using the features generated by Fourier transform applied to the program instructions, the author showed that the model successfully captures intra-function metamorphism.

Malware classification with LSTM and GRU language models and a character-level CNN Athiwaratkun et al. [121] conducted an evaluation of the efficacy of various language models in dynamic malware detection. The authors benchmarked different networks such as ESN (Echo State Network), RNN, LSTM, and CNN combined with auxiliary techniques such as temporal max pooling and attention mechanism. The result indicates that LSTM with temporal max pooling and logistic regression for output classifier performs the best. Character level convolutional neural network showed the worst performance, which suggests that capturing the temporal positional relationship is critical in sequence classification.

Huang et al. [122] created the first deep learning based model for dynamic malware detection called, MtNet. The model implemented with MLP serves as a binary classifier outputting malicious and benign probabilities as well as a multi-class classifier outputting fine-grained malware family probability distribution. The features used include null-terminated tokens in the memory, sparse binary vector of individual API calls combined with their parameter values, and 3-grams of API calls. Thereafter Mutual Information is used to obtain the best relevant feature set. The authors reduced the input dimension further using random projection as suggested by Dahl et al. [117]. The dual task model shares the weights in the hidden layer except for the softmax output layer. The model is simultaneously trained through standard backpropagation using stochastic gradient descent with the loss function given by the weighted sum of individual output loss functions. The experimental result shows that multi-task model outperforms a single-task model, suggesting that the malware family classification helps regularising the network and learning better feature abstractions for binary classification.

Kolosnjaji et al. [123] investigated a deep learning approach against dynamic malware detection problem by using a binary classifier implemented with a combination of convolutional neural network and recurrent neural network. Kernel system calls retrieved by Cuckoo sandbox were used as a primary feature while labels were retrieved from VirusTotal. Interestingly, the authors performs feature pre-processing which limits the number of repetitive calls occurring in a loop and one-hot encoding of API calls in the binary feature vector. The proposed model first produces sub-sampled latent representation using two convolutional layers with filter size of 3x60, which has a similar effect of 3-gram of system calls. It then feeds the output feature map from the convolutional layers to each step of the LSTM to model the sequential dependency in the system call sequence. The final output is calculated by taking the softmax cross entropy on the mean-pooled LSTM's last output to get the class label. This hybrid model outperforms other standard deep learning models while beating traditional machine learning models with a large margin.

2.5 Understanding Raw Binary Executables

When raw binary executables are used as a primary source of feature, it is critical to integrate their semantics contained within program instructions. Accurately consolidating these semantics into the detection pipeline has been a big challenge. Some of the prominent recent researches in this field are reviewed in this section.

Shin et al. [124] first attempted function recognition in raw binary executables with deep learning. The authors demonstrated that identifying the function signatures can be done with bidirectional LSTM. Although the result shows a respectable performance on legitimate binary files, the model was not tested against malware binary files, which poses greater challenges such as polymorphism and metamorphism. The model's efficacy against malware anti-cognition techniques is questionable especially when it struggles with simple variation in compiler optimisations.

Using RNN, Pascanu et al. [125] proposed a dynamic malware detection method that automatically generates the latent features out of raw inputs to avoid manual hand-crafting of features. The authors deployed a novel non-linear down-sampling method for RNN called, temporal max-pooling, which is derived by taking the maximum of hidden state across all time steps for each neuron in the memory cell. The authors also used a technique called, half-frame, that combines the memory state in the middle to the output hidden state, which increases the memory capacity. A separate MLP classifier was independently trained, leaving RNN as a pure feature extractor. The experiment conducted against a raw sequence of API calls shows that it outperforms the 3-gram model by a large margin. Although this primitive model demonstrated the efficacy of sequence modelling using RNN, the approach is not suitable for a long sequence of data since vanishing gradient problem persists.

In order to reduce the feature dimensionality, Kan et al. [126] first used the concept of instruction ID in feature construction by grouping native CPU instructions into 206 unique categories based on the nature of the operation. The authors implemented a binary classifier that consists of embedding layer for the input instruction features, followed by a series of convolutional layers, max-pooling layer, fully-connected layer, and softmax output layer. This lightweight CNN model shows superior accuracy over a large dataset compared to traditional machine learning models with N-gram.

HaddadPajouh et al. [127] experimented LSTM based binary classifier over ARM processor instruction for the Internet Of Things (IoT) environment. The most prominent instructions were selected based on Information Gain, after which word embedding technique [128] was applied to transform the input to continuous numeric representation. The authors found that LSTM with 2 layers outperforms other configuration of LSTM while beating conventional machine learning methods by a large margin including Random Forest, SVM, KNN, and decision tree.

In all aforementioned approaches in this section, differing degrees of in-depth analysis of the binary executable was required to retrieve the domain-specific features. It is ideal to have the model automatically derive the knowledge itself and consolidate the semantics of instructions into the model. The following works address this crucial aspect by directly dealing with raw binary executable files without domain knowledge.

The main contribution by Le et al. [129] is the elimination of complex feature engineering that requires expert domain knowledge, by using the raw binary file as an input to the model. The authors created a multi-class classifier using a combination of CNN, unidirectional LSTM and bidirectional LSTM. Treating the binary executable file as an image, Le et al. scaled it down to 10,000 bytes using an image library. CNN layers were used as a pre-processing step to capture local sequences and patterns within the bytecode on a spatial level while LSTM layers model their longer distance relationships throughout the file. The best performance was achieved with CNN-BiLSTM. As the authors are aware, the data model does not consider the semantics of the code in the raw binary file, which contributes to the distortion in the classification.

In the same spirit, Kalash et al. [130] also implemented a binary classifier treating the raw executable as an image similar to the approach taken by Le et al. [129], using VGG-16 image recognition model [131] over binary files. Zhao et al. [132] also used grey scale images as a feature, obtained through code mapping, texture partitioning, and texture extracting. The authors used a standard CNN followed by two fully connected layers like previous image-based malware classification.

Motivated to avoid domain-specific feature engineering, Raff et al. [133] proposed an approach that detects malware with CNN using the features automatically extracted from raw binary executables. Unlike the other approaches by Le et al. [129] and Kalash et al. [130], the authors have taken advantage of embedding [69] to map the symbolic inputs to the representation in the model space that can be learned via backpropagation. The authors implemented a model called, MalConv, primarily based upon 1-

dimensional convolutional network to best capture the spatial invariance occurring within the same class malware while discarding RNN based approach simply due to the excessive sequence length present in the raw executable binary. The model converges only within 3 epochs and shows a better performance than N-gram based approach in general. The authors also pointed out that MalConv is more resilient to N-gram approach since N-gram is fragile to even a single byte change. The correlation between the classifier's prediction and the reactive portion within the binary executable file has been investigated using sparse class activation map proposed by Zhou et al. [134]. It shows the activation relevant to the classification comes from mostly executable header along with a variety of other parts including the code section. This indicates that MalConv pays attention to instructions as well as metadata whereas the activation from N-gram based approach [135] mostly comes from the metadata in the executable header. With batch normalisation posing a negative impact during the experiment, the authors suggested that multi-modal nature of executable file causes multiple modes of convolutional layer activations, which violates the underlying assumption of batch normalisation.

Marek et al. [136] further improved MalConv from Raff et al. [133] using various techniques including fixed embedding instead of typical learnable entity embedding [69] and convolution stride tuning to reduce the computational load. The authors claim they have slightly increased the performance from MalConv by using power-of-two strides in convolutional layers, Scaled Exponential Linear Unit (SELU) activation and removing the De-convolution regularisation. However, they experimented with unpacked raw binary executables dataset as opposed to the original dataset that includes a large number of packed metamorphic samples, which poses a relatively less challenge.

Chen [137] applied transfer learning using CNN trained from image space for malware detection. The author first resized and reshaped the original binary file to a fixed shape required by pre-trained ImageNet [138], then performed transfer learning where a portion of the layers is frozen and the last few layers are trained against prepared malware images. The author demonstrated the model works against multi-family malware classification problem as well as binary classification. The author also points out that the approach allows debugging what caused the prediction by checking the positive coefficients of super pixel [139].

2.6 Resiliency against Adversarial Attacks

Although the majority of the models described so far reported respectable detection accuracy, none of them has demonstrated the capability to understand instructions, thereby vulnerable to overfitting and further to adversarial attacks. As machine learning methods evolve, the attacks adapt to exploit the vulnerabilities of the detection models, which renders the topic of semantic cognition a very important part of the battle. This section zooms into various techniques targeting machine learning based detections.

In recent works, adversarial machine learning was identified as a threat to deep learning based models, exhibiting vulnerabilities to gradient-based attacks. Non-differential models are also susceptible to the evasion attacks by genetic algorithms. Anderson et al. [27] created a more generic framework using reinforcement learning (RL), which aims to evade static malware detection. The proposed model overcomes the limitations of previous approaches where the target model must be fully differentiable with weight and gradient information available to the attacker, or the attacker has unlimited access to probe the target model possibly with the access to the score. The authors trained an RL agent which learns to generate evasive malware variants by mutating a seed malware file at each step of the iteration, using statistical features including metadata from executable header and statistics from the file body were used for mutation. Caution has been taken during mutation in order to avoid breaking the functionality of the binary sample. The result shows a substantial evasion rate, demonstrating the model's efficacy in discovering the detection model's weaknesses and in mitigating this threat by performing adversarial training.

Kolosnjaji et al. [28] suggested an adversarial attack exploiting the vulnerability of deep neural network that utilises the raw executable bytes as a feature. Raff's model [133] was chosen as an attack target model. Kolosnjaji's model iteratively performs gradient descent by optimising the injected padding bytes one at a time, which essentially discovers bytes that need to be modified in order to flip the classifier's prediction. They recorded 60% evasion rate against MalConv model implemented by Raff et al. [133]. Although Kolosnjaji's attack against deep neural network is conceptually valid, the fact that instructions in the code section cannot be freely modified by the algorithm significantly limits the efficacy of the attack. Crafting a malicious binary in code section still remains challenging to adversarial attack researchers. In addition, this attack works when the fully trained defence neural network is available, which is unlikely to happen in practice.

Grosse et al. [29] also introduced a method to induce misclassification of the detection model by perturbation of the malware binary. The authors adopted adversarial example crafting algorithm based on the Jacobian matrix of the deep neural network put forward by Papernot et al. [30]. Although they demonstrated the attack against a simple feed forward deep neural network, the approach works against arbitrary differential classification neural network. The authors suggest potential defence mechanisms for this adversarial attack including defensive distillation [140] and adversarial training [141]. However, finding the gradients for embeddings of the discrete symbols and making the perturbed adversarial samples functionally intact remain as a challenge for the adversarial attack to work. Especially functional integrity requires many constraints to be defined for the training, which needs further research. Moreover, this white box based attack, which is not a stringent for the attack to be practical in itself, targeted over-simplified hypothetical defence model. More generic and plausible model should be deployed to support the efficacy of the attack.

Hu et al. [31] proposed a GAN-based attack model named, MalGAN, against black box detection system. The authors used a hypothetical binary feature vector which MalGAN can only add while disallowing the removal of features to retain the sample functional integrity. Since the detection system is not available, the authors used a substitute detector to fit the black box detector and provide the gradient information to train the adversarial example generator. Finally MalGAN is trained by minimising the generator loss, which reduces the predicted probability of malware being malicious and pushes the substitute detector's probability to recognise the malware as benign. The scary aspect of this approach is the generated adversarial example distribution, which is fully controlled and retrainable by MalGAN, is unknown to the substitute detector. This makes it extremely difficult for the detector to learn the patterns.

Biggio et al. [32] proposed an evasion attack against machine learning at test time with limited knowledge where surrogate dataset can be collected from alternate sources. The training objective is to minimise the approximated classifier's output probability where ‘zero’ indicates benign and ‘one’ indicates malicious. Like many adversarial evasion techniques, the authors employed iterative gradient descent algorithm to minimise the approximated classifier’s probability of correct decision. In addition, the authors introduced an extra training objective using density estimator that penalises the generated samples in low density regions. This extra constraint guides the adversarial example generation towards the region where negative class is concentrated, which has a similar effect as what mimicry attacks do to intrusion detection systems [33]. They demonstrated the efficacy of the attack against malicious PDF files.

Nonetheless adversarial attacks without the capability of generating legitimate instructions while maintaining the functional integrity will have little impact on various malware detection approaches described in this thesis such as generative detection and instruction cognitive detection. Particularly instruction cognitive representation has a filtering effect on the sequences of invalid instructions, reacting to the recognised portion of the input binary sequence only, which will neutralise all adversarial evasion attempts described in this section.

2.7 Metamorphism

Metamorphism has been one of the major tactics to defeat detection [41]. The crux of various metamorphic techniques is in its global spatial translation with local context intact. In order to deal with these variations, the model must be able to recognise the coherent pattern shared across the variants generated by the same metamorphic script. This section reviews several critical previous research addressing those techniques related to metamorphism.

Sebastián et al. [142] devised a malware labelling method over a large scale malware dataset. Primarily using Anti-virus vendor detection labels as a feature, the approach did not show a promising way of gaining confidence on unlabelled samples. Shalaginov et al. [143] provided a good introduction on applying machine learning to classification problem using n-gram as a feature. Li et al. [144] described the fundamental risk of blindly clustering using labels obtained from Anti-virus voting, which describes the problems of biased or misconfigured clusters. Their research points out the significance of choosing the right ground truth cluster samples based on their underlying features for the clusters to build.

Rieck et al. [93] established an embedding method in vector space using instruction *q*-gram from arbitrary malware features, which has become a fundamental building block for many machine learning research studies later on. Hu et al. [145] created a system called MutantX-S using prototype-based clustering algorithm over opcode n-gram features. Although the time complexity of the clustering method is close to linear, the confidence for unlabelled samples has not been discussed in detail.

Pai et al. [146] studied the efficacy of K-means and Expectation Maximization (EM) clustering over opcode sequence. They created a separate Hidden Markov Model (HMM) instance for each family in an attempt to produce similar representations for intra-cluster samples. However, the experiment did not show the resiliency of the model since the number of benign samples used in testing was significantly low to prove its accuracy on false positives.

Dilokthanakul et al. [147] demonstrated the use of variational autoencoder (VAE) over image clustering. Although deep learning has recently proven to be effective in many fields, discrete symbol data as input has remained challenging.

2.8 Research Challenges

A large number of researches have leveraged the use of statistical features with N-gram nominated as the most advanced data model within this category. Previous sections strongly suggest that more semantic features resilient to adversarial attacks are required for accurate detection.

Many approaches have treated malware detection as a binary classification problem where samples are labelled in one of two classes. As pointed out in several papers, acquiring correct labels itself is a challenging task due to the volume of the samples and heterogenous labelling techniques, which causes confusion in model training let alone interpretation of the result is difficult.

Given that the number of variations generated by metamorphic engine can be infinitely large in theory, those approaches that do not take the domain knowledge into account are highly likely to overfit over the limited number of training samples while drastically fail for the real world samples. Given that malware runs as a campaign that consists of large number of samples generated by an automated metamorphic script, the samples within a campaign are likely to share similar characteristics within it, which opens up a possibility of one-to-many detection.

One of the significant gaps in the previous researches is that nearly all approaches tried to solve a binary classification problem over a large training set with the assumption that the dataset has unbiased distribution across different styles of samples. However, there is no magic combination of formulas that will reliably detect all past, present, and future malware samples. The problem must be addressed chronologically. Combined with one-to-many detection concept, the crucial insight in malware detection problem is that the ultimate goal is to identify as many variants as possible that appear in the future from the smallest possible number of known malware samples present today. The higher the ratio of detected malware variant count over the number of training malware samples is, the better the model is considered to perform.

One of the common malware problems arises when the trained model is not designed to distinguish the legitimate instructions from data. A significant number of malware samples deliberately insert arbitrary amount of high entropy data in between the code fragments largely in an attempt to evade traditional detection methods. As pointed out by Zak et el. [148], disambiguating instructions by their binary opcode is critical for model generalization.

The research challenges are summarised below.

- Statistical features without semantic context expose the detection to a variety of adversarial evasion.
- Diversity of distributed malware takes complex non-linear form.
- Extremely small number of samples are available during malware outbreak, from which following variants must be detected.

- Lack of fine-grained level of instruction identification within raw binary executables allow adversarial attacks.

In this thesis, all of the research challenges will be addressed by leveraging deep learning.

2.9 Theoretical Background

For the given problems outlined in this chapter, a supervised learning method can be adopted. The training set $\mathcal{D} = (\boldsymbol{X}, \boldsymbol{Y})$ consists of (x_i, y_i) where $x_i \in \mathbb{R}^N$ is the feature vector and $y_i \in \{1, \dots, K\}$ is the label for the sample x_i. Binary classification can be implemented by simply setting $K = 2$. The objective is to find a function $f_\theta: \boldsymbol{X} \to \boldsymbol{Y}$ where θ is learnable model parameters. The prediction for an unseen sample is made based on equation (1).

$$z = \underset{k \in \{1,\dots,K\}}{argmax} f_k(x) \tag{1}$$

Parametric non-linear classifiers such as SVM and deep learning typically use gradient descent method to find optimum parameters via iterative parameter update. For the model function $f_\theta(x_i)$ to be learned, the cost function is defined as equation (2).

$$J_\theta = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(f_\theta(x_i), y_i) \tag{2}$$

The loss function, $\mathcal{L} : \boldsymbol{Y} \times \boldsymbol{Y} \longmapsto \mathbb{R}^N$, measures the difference between the predicted output for the input x_i and the class label y_i. Information-theoretic cross entropy function is commonly used to compute the difference between the predicted probability distribution and the class probability distribution as shown in equation (3).

$$H_\theta(y, z) = -\sum_{i=1}^{N} y_i \log f_\theta(x_i) \tag{3}$$

With learning rate, $\alpha \in \mathbb{R}$, and parameters θ_j where $j \in \{1, \dots, L\}$, the gradient descent update is described as equation (4).

$$\theta_j := \theta_j - \alpha \nabla J_\theta \tag{4}$$

In statistical machine learning, the implicit assumption is that the samples are independently drawn from an identically distributed joint probably distribution set:

$$p(X, Y) = p(Y|X)p(X) \tag{5}$$

The *Independent and Identically Distributed (I.I.D)* assumption is formally described in equation (6) and (7) respectively.

$$(x_i, y_i) \sim P(X, Y), \forall i = 1, \dots, N \tag{6}$$

$$(x_i, y_i)\ independent\ of\ (x_j, y_j), \forall i \neq j \in \{1, \dots, N\} \tag{7}$$

However, in the context of malware detection, samples are heavily biased towards benign samples with much higher diversity within the benign class, which severely breaks the assumption in equation (6). This imbalanced dataset makes the unsupervised learning more viable.

In unsupervised setting, instead of learning the parameters for a function $f_\theta: \boldsymbol{X} \rightarrow \boldsymbol{Y},$ we define a function $g_\theta: \boldsymbol{X} \rightarrow \boldsymbol{Z}$ where $\boldsymbol{Z}$ is the latent representation that map similar samples to the points in linearly close proximity in latent space. The distance between two points $\boldsymbol{a} = (a_1, a_2, \dots, a_M)$ and $\boldsymbol{b} = (b_1, b_2, \dots, b_M)$ in latent space is computed using linear distance metrics such as *Euclidean distance*. See equation (8).

$$d(\boldsymbol{a}, \boldsymbol{b}) = \sqrt{\sum_{i=1}^{M} (a_i - b_i)^2} \tag{8}$$

The prediction for an unseen sample $\boldsymbol{x_i} \in \mathbb{R}^N$ where N is the dimension of the input feature vector, is made by choosing $\boldsymbol{z_i} \in \mathbb{R}^M$ where M is the dimension of the latent representation according to the equation (9).

$$\boldsymbol{z_i} = \min_{j \in \{1,\dots,N\}} d(g_\theta(x_i), g_\theta(x_j)) \tag{9}$$

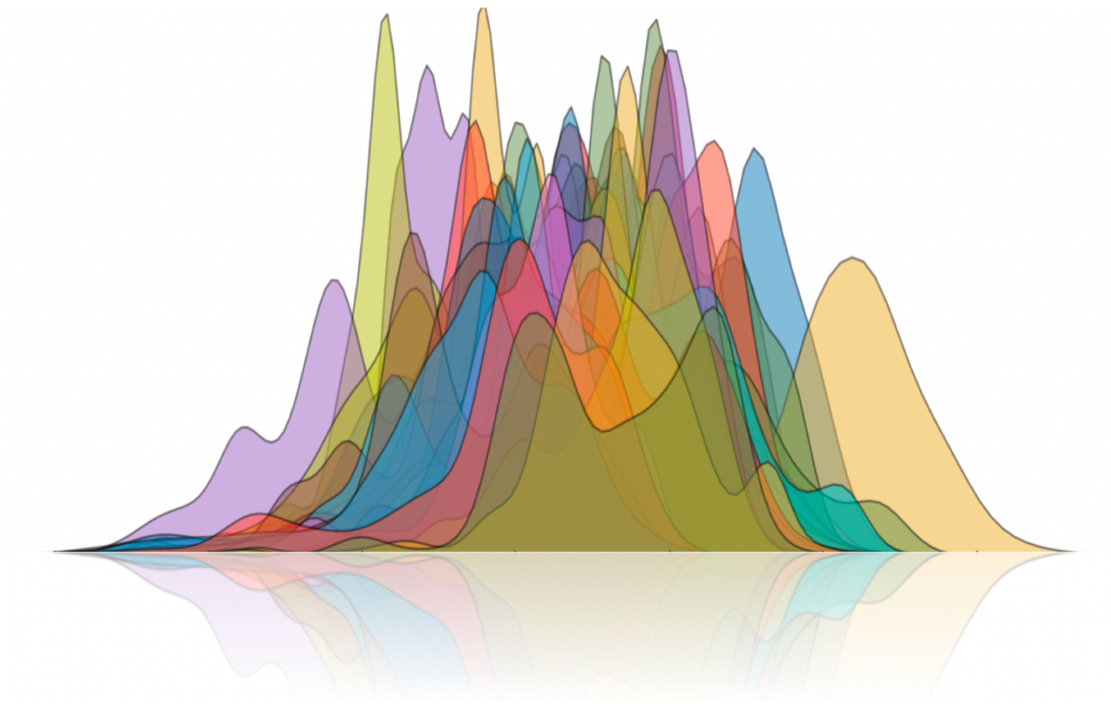
Note that g_θ is a non-linear transformation whose parameters are learned via training whereas the distance is calculated using a fixed metric.

2.10 Conclusion

In this chapter, we have reviewed the core strategies taken by malware to evade detection, the strengths and weaknesses of deep learning, and many recent malware detection techniques using traditional machine learning and deep learning. We have also highlighted the need of a more accurate model when deep learning is used with raw binary executables in order to stay resilient against adversarial attacks.

Having identified the key research challenges through the comprehensive review conducted in this chapter, we will tackle the static malware detection problem by leveraging generative adversarial autoencoder in the next chapter.

Chapter 3: Generative Static Malware Detection



The work in this chapter was published in following paper:

Sean Park, Iqbal Gondal, Joarder Kamruzzaman, Jon Oliver, "Generative Malware Outbreak Detection" 2019 IEEE International Conference on Industry Technology. Melbourne, 2019.

In chapter 2, various previous machine learning methods have been presented to detect malware. This chapter will present a method that generalises the static malware detection using one-shot training to detect the variants within a malware campaign.

Recently several deep learning approaches have been attempted to detect malware binaries using convolutional neural networks and stacked deep autoencoders. Although they have shown respectable performance on a large corpus of dataset, practical defence systems require precise detection during the malware outbreaks where only a handful of samples are available. This chapter demonstrates the effectiveness of the latent representations obtained through the adversarial autoencoder for malware outbreak detection. Using instruction sequence distribution mapped to a semantic latent vector, a model can provide a highly effective neural signature that helps detecting variants of a previously identified malware within a campaign mutated with minor functional upgrade, function shuffling, or slightly modified obfuscations. The method presented in this chapter demonstrates how adversarial autoencoder can turn a multiclass classification task into a clustering problem when the sample set size is limited and the distribution is biased. The model performance is evaluated on OS X malware dataset against traditional machine learning models.

3.1 Introduction

Timely malware detection is critical, especially in current threat environment where malware outbreaks are a daily routine. This chapter considers static features for malware detection. Although malware packing problem [3] is crucial when reverse engineering the detailed functional characteristics of the malware is the objective, the surface level static features are sufficient to differentiate malware families from benign samples when the ultimate goal is solely the detection.

The observations in the malware battle forefront are malware mutates over time to bypass static signature-based detection by upgrading its functions or applying new metamorphic (or obfuscation) techniques. The downside for attackers is the malware mutation requires their time and effort to develop new variants. Due to this development cost, minor tactical modification to the original malware code frequently occurs and is seen in the form of an outbreak while a major strategic code change rarely occurs across a longer period of time. This inevitably causes a similar pattern of instruction sequence either generated by a metamorphic engine or upgraded from previous functions. As a result, there appears a phenomenon typically seen for the instruction sequence of malware samples from a campaign as shown in Figure 1.1. The method presented in this chapter will exploit the presence of this unique pattern of instruction sequence in the malware samples of a campaign whether or not it forms unpacking routine, metamorphic components, or pure functional modification.

Traditional machine learning algorithms such as SVM, Random Forest and Gradient Boosting commonly use metadata as features such as executable file header fields, n-gram of raw binary file and entropy of sections because they are optimised to work with independent sparse features. In the meantime, encoded high dimensional data such as sequence of program instructions constitutes the substantial body of a sample in malware detection context, which contains rich information of the sample's identity. Although several previous techniques utilised the instruction sequence as a feature [149][150][84][151], the use of n-gram simply discards the sequence order information, leaving those approaches vulnerable to a trivial histogram matching attack [152]. However, adversarial autoencoder used in this chapter, like many other deep learning models, fully takes advantage of the input samples with the sequence order retained. In this chapter, we use the sequence of program instructions as a feature.

One critical aspect of malware outbreak detection is that there are a scarce number of samples we can train with. The goal of this chapter is not only detecting malware variants but also detecting them with extremely small number of samples captured at the very early outbreak as it should be in real world detection scenario. The method as well as various other machine learning models will be evaluated with this scarce training dataset setting.

This chapter presents a novel method that detects similar malware samples with high accuracy on malicious samples and low false positives on benign samples, using a single sample of a kind for training with adversarial autoencoder. The techniques described in this chapter are applicable to the other domains, such as IoT (Internet Of Things), that require well-generalised detection using a handful of malware samples.

3.2 Method

This section discusses features, dataset, model architecture, and training methods.

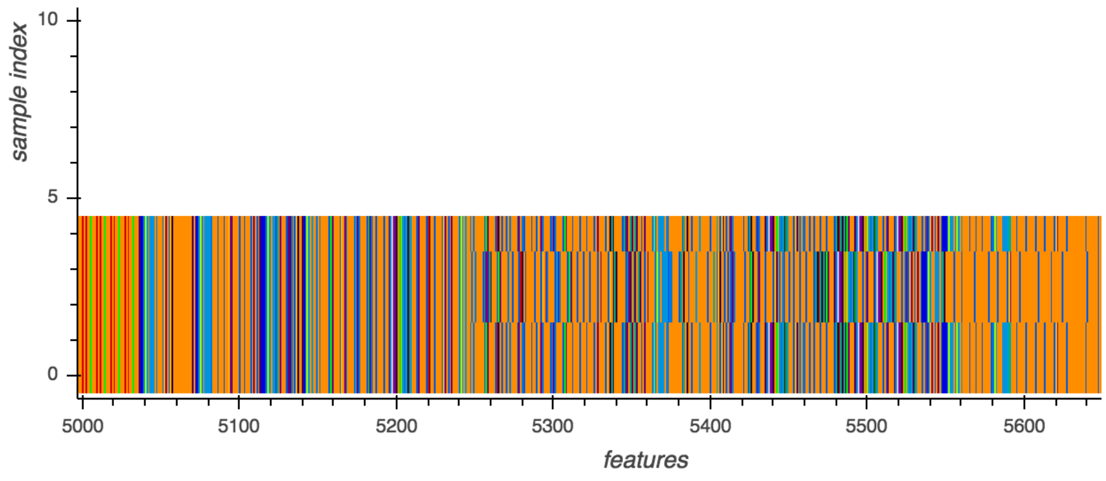


Figure 3.1. Each row represents a per-sample feature, which is a sequence of instructions of a malware sample. Each normalised instruction is rendered as vertical bar with a unique colour to differentiate between different instructions. X axis is the feature. Y axis is the sample number.

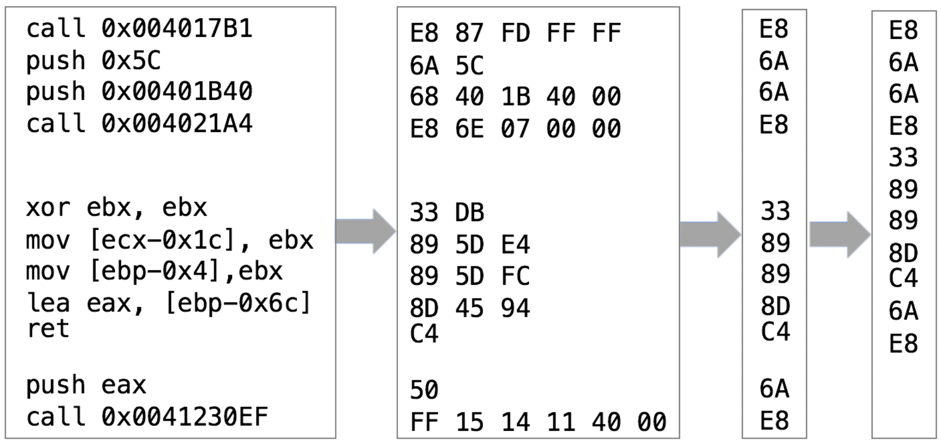
3.2.1 Static Features

Today malware samples are automatically generated by a custom tool created by the attacker. It renders hard-coded static signature based detection obsolete. A run of automatic malware generation tool essentially creates a batch of the functionally same malware in a different look, which can involve different obfuscations [41][153] such as dead code insertion, register reassignment, code transposition and integration and control flow obfuscation. Nonetheless the distribution of the program instruction sequence remains relatively intact. Figure 3.1 shows three unique variants of *MAC.OSX.CallMe* family. The samples in Figure 3.1 are described below.

MAC.OSX.CallMe.A (3 samples)
MAC.OSX.CallMe.E (1 samples)
MAC.OSX.CallMe.F (1 samples)

As shown in Figure 3.1, *MAC.OSX.CallMe* variants have identical instruction sequence until a variation was introduced at approximately instruction 5250. Despite this variation it is visually clear that parts of instructions of sample 2 and sample 3 are merely shifted from the instructions of the rest of the samples. In short, the instruction sequences appear very similar to each other for these three *MAC.OSX.CallMe* variants. Visual analysis on the majority of malware families shows that program instruction sequence plays a significant role in identifying the variants during outbreaks.

Based on this insight, we use the instruction sequence as the sole feature for the model proposed in this chapter. The steps to construct a feature vector for a sample are described below and example values are shown in Figure 3.2.



```
call 0x004017B1
push 0x5C
push 0x00401B40
call 0x004021A4


xor ebx, ebx
mov [ecx-0x1c], ebx
mov [ebp-0x4],ebx
lea eax, [ebp-0x6c]
ret

push eax
call 0x0041230EF
```

```
E8 87 FD FF FF
6A 5C
68 40 1B 40 00
E8 6E 07 00 00


33 DB
89 5D E4
89 5D FC
8D 45 94
C4

50
FF 15 14 11 40 00
```

```
E8
6A
6A
E8


33
89
89
8D
C4

6A
E8
```

```
E8
6A
6A
E8
33
89
89
8D
C4
6A
E8
```

Figure 3.2. Example values for each step of the feature extraction.

1) Extract function-wise raw instruction bytes using IDA Pro [26]: It is critical to extract the original raw features. Failing that, sample distribution will change, which will significantly affect the clustering result. In this chapter, a list of function bytes are extracted from each malware sample using a custom IDA Python script.
2) Create a sample by combining the extracted functions: Each individual data sample per malware sample needs to be created. This is done by concatenating the functions present in the executable file in the order they appear. A blind concatenation of functions is vulnerable to code transposition and integration metamorphism [41]. How to overcome this problem is discussed later in this section using a model with translation invariant property.
3) Map each instruction byte to a unique instruction ID: Instruction's operands are ignored. For example, both push 0x5C and push eax are mapped to a unique ID, 6A. Note that this unique ID is computed using a custom table instead of being assigned directly from the instruction's opcode because the opcode in CPU architecture may include a portion of a byte or span across multiple bytes. The

rationale for this pre-processing is it reduces noises in the distribution while remaining immune to several obfuscation techniques such as register and memory reassignment [41].

3.2.2 Adversarial Autoencoder

The model consists of two independent modules. Firstly the latent representation for the instruction sequence feature that is resilient to metamorphism is acquired by adversarial autoencoder. Then the class number is computed for the latent representation via HDBSCAN with predefined threshold.

Over the past few years, Generative Adversarial Network (GAN) [56] has successfully demonstrated its capability to understand the data distributions by generating realistic samples. The power of GAN primarily comes from its generative nature by jointly training the generator and the discriminator in a tight competitive loop. In a situation like malware outbreaks where a handful of samples are available, adversarial autoencoder is a natural choice so that the scarce number of training samples produce smooth approximated nearby distributions. The core architecture for malware outbreak detection in this chapter is borrowed from the original adversarial autoencoder [58] as shown in Figure 3.3.

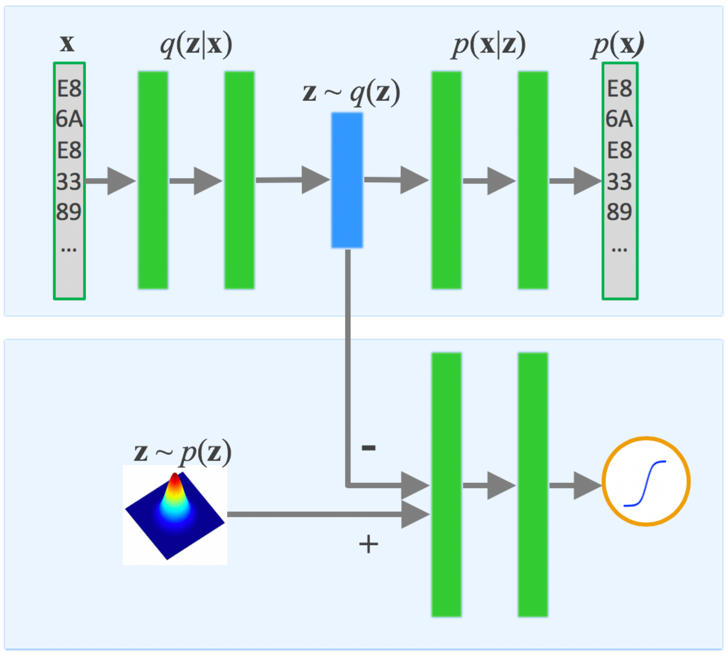


Figure 3.3. Adversarial autoencoder architecture used for malware outbreak detection. The input, **x**, and the reconstructed input, *p*(**x**) are instruction sequence feature.

Adversarial autoencoder essentially combines an arbitrary autoencoder with GAN. The autoencoder part within the model must have two properties:

- The stacked weights are symmetric and shared between encoder and decoder: This is a compulsory requirement to be qualified as an autoencoder.
- Encoder also functions as a generator, hence must have all properties of the generator of GAN: Since encoder has a dual function, it needs to conform to the training techniques used for the generator while maintaining the autoencoder property.

As one of the desired properties in malware outbreak detection is to identify relocated functions, the autoencoder used in the proposed model is a stacked convolutional autoencoder [66] to take advantage of translation invariant property of the architecture. This allows the model to capture the program instruction sequence in the presence of code transposition and integration metamorphism [41].

The input vector consists of sparse discrete symbols, which are difficult to train with stochastic gradient descent. Therefore we create an embedding lookup for the symbols, and let the model find the best representations for them during the training. This embedding layer nicely transforms 1D input vector to a 2D array, which can be fed as an input to this convolutional autoencoder. The reconstruction method based on cosine similarity cross entropy is used to deal with sparse discrete input symbols [70].

As outlined in [58], both the adversarial network and the autoencoder are trained jointly with stochastic gradient descent in two phases – the reconstruction phase and the regularization phase – executed on each mini-batch. Specifically, in reconstruction phase, the model is trained by minimizing the cross entropy loss between the input symbol and the decoder output with sigmoid activation. During regularization phase, binary cross entropy is used for the discriminator loss, which is computed by summing the loss between positive samples from Gaussian normal distribution and negative samples from the encoder output. Binary cross entropy is also used for the generator loss. Let $\boldsymbol{x}$ be the one-hot encoded representation of input data distribution and $\boldsymbol{z}$ be the latent code vector of an autoencoder. Let $\mathrm{p}(\boldsymbol{z})$ be the prior distribution imposed on the codes, $\mathrm{q}(\boldsymbol{z}|\boldsymbol{x})$ be the encoding distribution, $\mathrm{p}(\boldsymbol{x}|\boldsymbol{z})$ be the decoding distribution, and $\mathrm{p}(\boldsymbol{x})$

be the model's reconstructed distribution. Reconstruction loss is described in (1), and discriminator and generator loss are defined by (2).

$$E_{\boldsymbol{x}}\left[E_{q(\boldsymbol{z}|\boldsymbol{x})}[-\log p(\boldsymbol{x}|\boldsymbol{z})]\right] \tag{1}$$

$$\min_{G} \max_{D} \; E_{\boldsymbol{x}\sim p_{data}}[\log D(\boldsymbol{x})] + E_{\boldsymbol{z}\sim p(\boldsymbol{z})}[\log (1 - D(G(\boldsymbol{z}))] \tag{2}$$

In order to mitigate the instability caused by standard adversarial autoencoder training, consensus optimization has been adopted. Mescheder L [62] showed that simultaneous gradient descent used in GAN does not generally converge to Nash equilibrium in non-cooperative minmax game. The solution proposed by Mescheder L is to construct a conservative vector field from the original using consensus optimization [62].

There are a number of hyperparameters that can be tuned in the network architecture such as how much of diverse clusters you want to detect and how much performance you need during prediction. In terms of clustering behavior, the standard deviation of GAN's Gaussian noise input generally affects the total number of clusters detectable with accuracy. The wider the Gaussian normal distribution is, the larger number of clusters the model will spread evenly. The gradients are clipped by a global norm to force Lipschitz constraint in latent variables, which enhances the smooth code generation for a small number of training samples. A large size latent vector increases the accuracy of clustering. On the contrary, a large embedding size for input symbol does not have significant impact on clustering accuracy.

In general, convolutional autoencoder part of the network does not influence much in accuracy, but the increased number of convolutional layers can significantly reduce both training and prediction speed. A large convolutional filter window tends to produce less optimal results.

From the training perspective, batch normalization within the GAN generator is necessary to help generate consistent latent representation. ADAM optimizer [154] was used for both reconstruction and regularization phase while consensus optimization was performed with RMSProp [155]. Some of the key hyperparameters are shown below:

Latent representation dimension: 100
Input gaussian noise standard deviation: 5.0
Embedding dimension: 4
Number of channels for each convolution layer: [1, 20, 20, 1]
Filter sizes: [3, 3, 3, 3]
Strides: [1, 2, 2, 1]
Maximum epochs: 100
Learning rate: 0.0001
Batch size: 20

When training is complete, the encoder output is taken as a latent vector that represents the input sample, which will be used as input to semantic hashing.

3.2.3 Semantic Hashing

The latent representation obtained through adversarial autoencoder needs to be transformed into a class number for prediction. Firstly, the latent vector represented by real valued numbers are binarized using the bitwise mean value of the training samples. Then hamming distance is used to compute the distance for the two given latent vectors [71]. Finally, a test sample is assigned a class of the closest training sample.

3.3 Evaluation

3.3.1 Static Sample Dataset

In total, 3,254 in-the-wild OS X malware samples collected from proprietary source and randomly chosen 9,981 benign OS X Mach-O samples were used for evaluation. These samples are equipped with full malicious functionalities and have once managed to enter into user machines to exercise real damage. A snapshot of family distribution of 3,254 in-the-wild malware samples is shown in Figure 3.4.

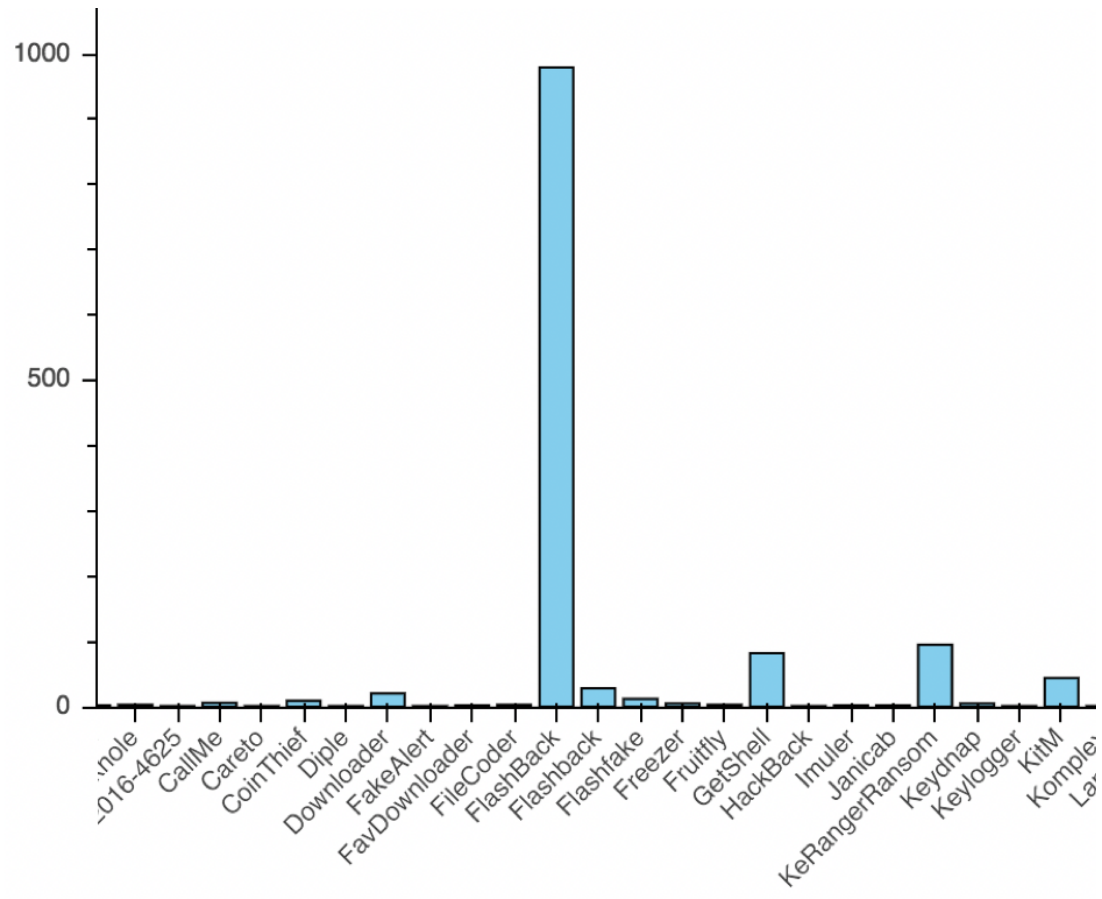


Figure 3.4. Malicious sample distribution by VirusTotal detection names. The biggest bar at the far left hand side indicates the samples with no detection or with generic name.

In order to simulate the outbreak situation, 175 out of 3,254 malicious samples that exhibit unique instruction sequence patterns were manually selected by a human malware expert as core malicious training samples and were assigned a unique label for each sample. Note that no benign samples were included in the training set.

Since there is no generic evaluation metric available to find a core sample of a malware family based on instruction sequence, the instruction sequences of all 3,254 malicious samples were visually explored to obtain the core sample of each family. The properties used to include samples in the same category are summarized as follows.

- Similar instruction distribution: The samples have similar instruction distribution statistics.
- Minor local variations: The modification of a sample's instruction sequence is restricted to one or more local areas.

- Translation invariant: Most part of the sample's code distribution is identical to the rest of them within the same cluster when the code is translated or shuffled.

The properties for unqualified malware family are summarized as follows:

- Mismatched function-wise distribution: Neither similar sample length nor similar statistical distribution qualifies a sample to become a member of a cluster. Samples must also match function-wise statistical distributions.
- Substantial difference in code distribution size: Although a partial match suggests a variant, it is desirable to have the size of similar code distribution significantly larger than the variations. It's because the clusters can drift over repetitive trainings across longer period of time, which can potentially cause false positives due to the mixed distributions.

In order to find out the category for the latent representation, HDBSCAN clustering algorithm was used, which shows the most appealing performance against unknown number of clusters [156]. The samples categorized as a noise by HDBSCAN are classified as benign because it indicates no similar cluster was found for the test sample. Overall flow chart is illustrated in Figure 3.5.

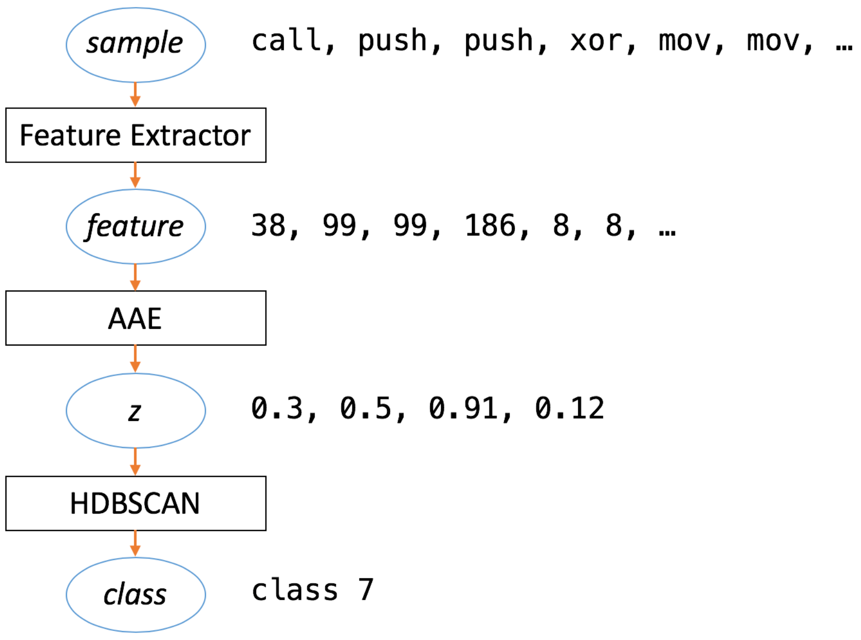


Figure 3.5. Overall pipeline from a sample to its predicted class number. Examples are shown on the right.

3.3.2 Static Detection Result

Using raw instruction sequence for classification models significantly reduces the accuracy. Gradient boosting, Support Vector Machine and Random Forest models were chosen as baseline with the feature implemented using n-gram [150] over the instruction sequence. Clustering models such as KNN were not included in the baseline model since they need decent number of training samples to work, which is different to the problem setting put forward in this research.

As shown in Table 3.1, traditional classifiers perform reasonably well even for a training dataset that consists of a single sample for each class. The proposed model, aae-sh, which is adversarial autoencoder combined with semantic hashing, shows reasonably high detection accuracy against malicious samples. However, all traditional classification models catastrophically fail on benign samples, recording 100% false positives. With the training set of only core malicious samples by which outbreaks are simulated, the traditional classification methods do not work at all. On the contrary, *aae-sh*, records a 91% accuracy over benign samples with this training setting.

Model	Malicious (3,254)	Benign (9,981)
gradient-boosting-1gram	0.935	0.000
gradient-boosting-2gram	0.936	0.000
gradient-boosting-3gram	0.931	0.000
svm-1gram	0.934	0.000
svm-2gram	0.944	0.000
svm-3gram	0.968	0.000
randomforest-1gram	0.983	0.000
randomforest-2gram	0.987	0.000
randomforest-3gram	0.989	0.000
aae-sh	0.959	0.910

Table 3.1 Detection rate against malicious and benign samples for various models.

3.3.3 Static Detection Analysis

Visual analysis of the families detected by aae-sh not only shows similar instruction sequences with variations within the family but also it does not exhibit undesirable properties described in the previous subsection. Figure 3.6 shows aae-sh correctly identifying malware variants whose major feature mass is identical across all samples in the cluster while variations occur in many different ways. These samples are the variants of malware named Blackhole or Freezer. It becomes clear that the names from VirusTotal [157] do not necessarily agree with malware clusters produced by aae-sh because human analysts tag detection names based on analyst-specific heuristics whereas the proposed approach in this chapter derives the detection purely from the instruction sequence pattern.

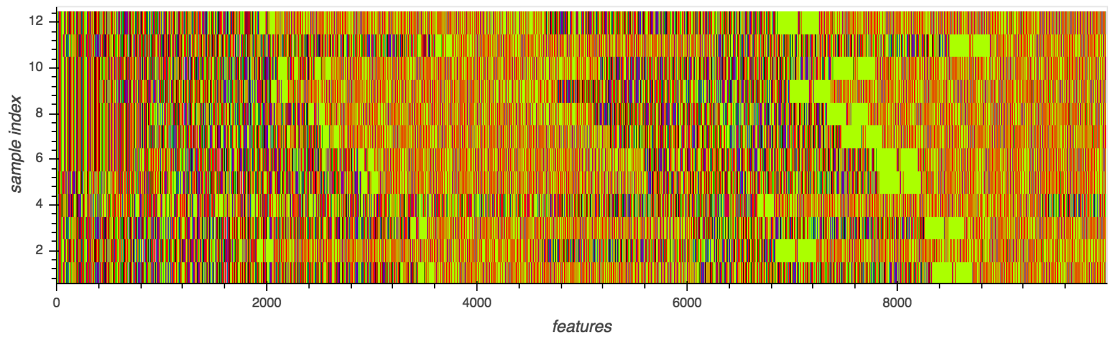


Figure 3.6. Visualisation of instruction sequence of Blackhole or Freezer samples identified by aae-sh. X axis is the feature. Y axis is the sample number.

Figure 3.7 shows the detected cluster 49 that contains many Flashback variants. Note that aae-sh detected the samples of different lengths as long as instruction sequences are similar.

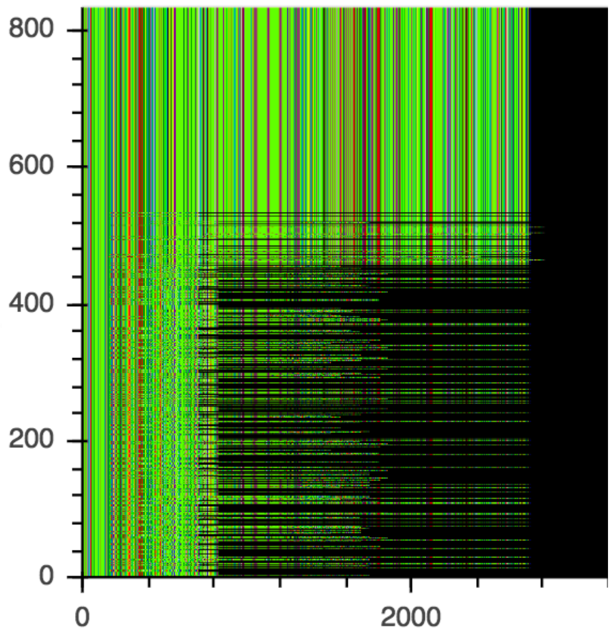


Figure 3.7. Visualisation of instruction sequence of malware samples within cluster 49 in Figure 3.8 identified by aae-sh. X axis is the feature. Y axis is the sample number.

```
Cluster 49 (nsamples=836)
        FlashBack.AF (2 samples)
        FlashBack.L (344 samples)
        FlashBack.M (5 samples)
        FlashBack.Q (2 samples)
        Flashback.E (1 samples)
        Flashback.J (1 samples)
        Flashback.K (1 samples)
        Flashback.L (1 samples)
        Flashback.M (10 samples)
        Flashback.N (7 samples)
        Flashback.O (1 samples)
        Flashback.P (1 samples)
        Flashback.Q (1 samples)
        Trojan-Downloader.Flashfake.ab (12 samples)
        Unknown (447 samples)
```

Figure 3.8. VirusTotal detection names for the samples in cluster 49 that is visualised in Figure 3.7.

Figure 3.8 shows redacted VirusTotal detection names for the samples in Figure 3.7. It is notable that 447 samples either had no names or had generic names.

3.3.4 Theoretical Analysis

A function $f : \boldsymbol{X} \longmapsto \boldsymbol{Y}$ is called *Lipschitz continuous* if there exists a positive constant K for any given x_1 and x_2 in $\boldsymbol{X}$ such that

$$|f(x1) - f(x2)| \leq K(x1 - x2) \tag{1}$$

The encoder function of the adversarial autoencoder, $q : \boldsymbol{x} \longmapsto \boldsymbol{z}$, maps input feature vector $\boldsymbol{x}$ to a latent variable $\boldsymbol{z}$ whose distribution $q(\boldsymbol{z})$ is imposed by the prior distribution, Gaussian distribution, after the training.

$$q(\boldsymbol{z}) \sim \boldsymbol{\mathcal{N}}(0,1) \tag{2}$$

The decoder function of the adversarial autoencoder, $p : \boldsymbol{z} \longmapsto \boldsymbol{x}$, reconstructs the original input $\boldsymbol{x}$ from $\boldsymbol{z}$.

Lemma 1. *The encoder function* $q : \boldsymbol{x} \longmapsto \boldsymbol{z}$ *is Lipschitz continuous.*

Proof. Let the gradient of the encoder function be $\boldsymbol{g}$ and the clipping norm be c. The following gradient penalty is applied if $||\boldsymbol{g}|| > c$ where $||\boldsymbol{g}||$ is Euclidean norm over gradients.

$$\boldsymbol{g} := \frac{c\boldsymbol{g}}{||\boldsymbol{g}||} \tag{3}$$

A differentiable function is K-Lipschitz if and only if it has gradients with norm at most K everywhere [158]. Therefore, the encoder function $q : \boldsymbol{x} \longmapsto \boldsymbol{z}$ is Lipschitz continuous, concluding the proof.

Lemma 2. *The decoder function* $p : \boldsymbol{z} \longmapsto \boldsymbol{x}$ *is Lipschitz continuous.*
Proof. In autoencoder, the gradient of the decoder function calculated during the back propagation is identical to equation (3). Since the decoder function is differentiable and has gradients with norm at most K everywhere, $p : \boldsymbol{z} \longmapsto \boldsymbol{x}$ is *Lipschitz continuous*, concluding the proof.

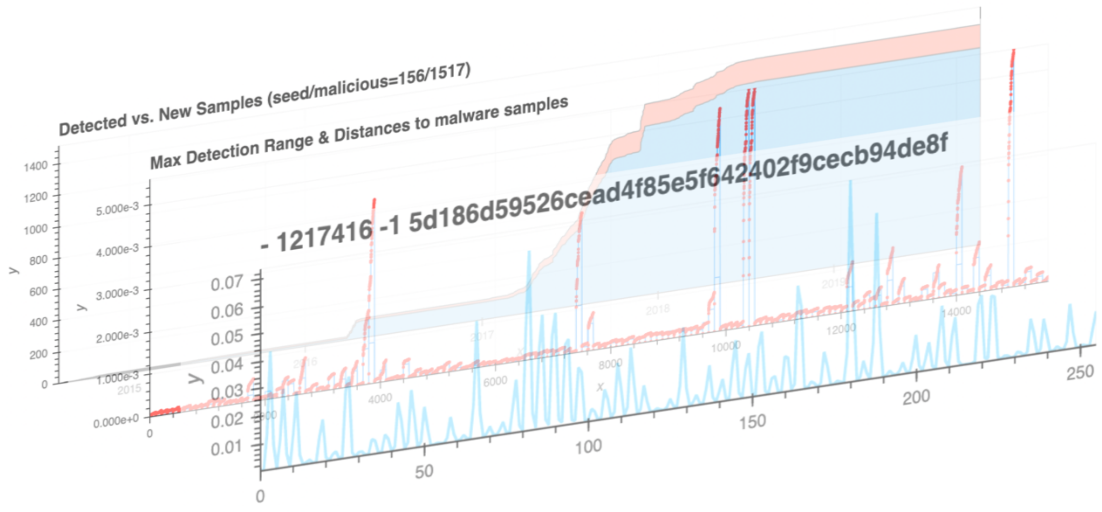
Theorem 1. *If there exist malware variants for a training malware sample that share common instruction features, the latent variables for malware variants are located near the latent variable for a single prototype training sample in latent space.*
Proof. Since the latent distribution $p(\boldsymbol{z})$ has Gaussian distribution after the training, all nearby points for a given z_1 in $\boldsymbol{Z}$ is continuous as shown in **Lemma 1**. **Lemma 2** dictates that the inverse of $\boldsymbol{z}$ is also *Lipschitz continuous* because the decoder function is a mirror implementation of the encoder function. Therefore, being the inverse of $\boldsymbol{z}$, input feature $\boldsymbol{x}$ is continuous if there exist nearby points in $\boldsymbol{z}$. This means, if there are latent $\boldsymbol{z}$ variables near z_1 corresponding to a known prototype malware sample, they have similar features to the known prototype malware sample. This theorem proves that a single training sample can find similar samples in the proposed generative adversarial autoencoder if its variants do exist.

3.4 Conclusion

The research presented in this chapter has shown that the generative power of adversarial autoencoder creates latent representations that can be used to identify similar samples with minimum number of training samples. It turned out that some malware families such as Flashback reuse the same piece of code repeatedly across their variants, which subsequently enables the adversarial autoencoder to identify its family effectively. In addition, the model was found to be effective in discovering multiple variants across heterogeneous malware families that share similar instruction-wise characteristics.

In chapter 3, we looked into static malware detection method by leveraging generative adversarial encoder. In chapter 4, dynamic malware detection is presented by extending the concept presented in this chapter to dynamic execution events.

Chapter 4: Generative Dynamic Malware Detection



The work in this chapter was published in following paper:

Sean Park, Iqbal Gondal, Joarder Kamruzzaman, and Leo Zhang, "One-Shot Malware Outbreak Detection using Spatio-Temporal Isomorphic Dynamic Features" 2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering. Melbourne, 2019.

Previous chapter focused on static malware detection using generative adversarial autoencoder (AAE). In this chapter, we will demonstrate its efficacy in dynamic malware detection where higher dimensional feature is utilised as input.

Fingerprinting the malware by its behavioural signature has been an attractive approach for malware detection due to the homogeneity of dynamic execution patterns across different variants of similar families. Although previous studies show reasonably good performance in dynamic detection using machine learning techniques on a large corpus of training set, decisions must be undertaken based upon a scarce number of observable samples in many practical defence scenarios. This chapter demonstrates the effectiveness of generative adversarial autoencoder for dynamic malware detection under outbreak situations where in most cases a single sample is available for training the machine learning algorithm to detect similar samples that are in the wild.

4.1 Introduction

Every malware has its own purpose, be it banking Trojan, ransomware, information stealer or advanced persistent threat. We name them by their behaviours. All top level malware categories contain multiple families, each of which again contains many variants. Despite this complex hierarchy of malware samples, the invariancy of execution offers a great opportunity to detect not only a wide range of malware variants of a family but also multiple families that share similar behaviour patterns. This chapter proposes a detection method that leverages this common nature of the malware.

Many individuals and organisations have been breached and have suffered financial losses since early 21st century due to ransomware [159]. Although ransomware has a large number of different families, its execution fundamentally involves behaviourally similar patterns:

- Downloads a custom encryption key from a remote command and control server
- Enumerates the file system for the targeted files in the victim's machine
- Encrypts the target files using the key
- Displays the ransom note

- Extorts the victim for payment.

Likewise, various banking Trojans that have impacted the world over the past decade such as Zeus, Dyre, and Trickbot have kept their core behaviour unchanged although they have gone through numerous binary structure changes to bypass static detection. After many years, the fact that banking Trojans hijack web traffic, inject a custom Java script within a target online banking page and initiate or modify a covert transaction to underground channel, has not still changed. Although there are minor variations in the behaviours of variants of a family or a group of families, the substantial body of the execution remains relatively invariant. As a result there appears a phenomenon as shown in Figure 4.1, commonly seen among different variants and quite often cross-family malware samples that are behaviourally isomorphic in temporal behaviour space. The method proposed in this chapter will exploit the presence of this pattern exhibited by behaviourally similar malware samples.

The intrinsic behaviour of a sample is best defined by its dynamic execution log. As with many previous works, this chapter will leverage API call sequence as a feature to our model while maintaining its spatio-temporal characteristics. The API call logs are obtained via a custom sandbox [160] that captures a selected set of user mode API calls along with their parameters. Dynamic detection faces several challenges of its own. Multi-threaded operating environment can result in the API call events shuffled in threaded code blocks. A sample can exhibit several different behaviours depending on how it gets executed in a sandbox. Anti-analysis and non-deterministic programs can lead the sample to different execution paths. Resilience over mutation of dynamic execution is a long-term challenge as well. All of these perturbations in the event log interferes with accurate dynamic detection. The global feature that captures local characteristics is deemed optimal in this chapter for dynamic detection models.

From practical defence point of view, time-to-detection is a critical factor when there is a new malware outbreak and several thousand variants of its kind could soon follow. The task at hand is to build a quick knowledge base, out of a handful of samples which are available from the outbreak, in the manner that the detection is resilient to minor changes in the malware behaviour. This is very different to the traditional problem setting where a large corpus of training samples are available and used to predict maliciousness of test samples. Due to the availability of limited training samples, the problem in this chapter is posed as similarity-based detection instead of traditional binary classification.

The order of API call sequence within a block is the key to the semantics of the behaviour. Unfortunately traditional machine learning models such as SVM, Random Forest and Gradient Boosting generally use N-gram [149] which is computed from the original sample in order to come up with a fixed size feature vector. The N-gram approach takes away the important context information present in the global scope. This chapter explores a deep learning model that fully utilises the order of API call sequence information. Given that similarity-based detection is desired, a generative deep learning model is used and evaluated against several traditional machine learning approaches.

This chapter presents a novel method that detects malware outbreaks using dynamic execution features when very limited training samples are available. The proposed model, generative adversarial autoencoder, automatically extracts perturbation-resilient and context-aware features from the raw API call sequences to proactively detect malware outbreaks in practical threat response environments.

4.2 Method

This section discusses features, model, and metric of the proposed method. The proposed method takes advantage of the generative power of adversarial neural network against spatio-temporal features extracted from API call events, which shows remarkable generalisation of a single sample of its kind used in the training.

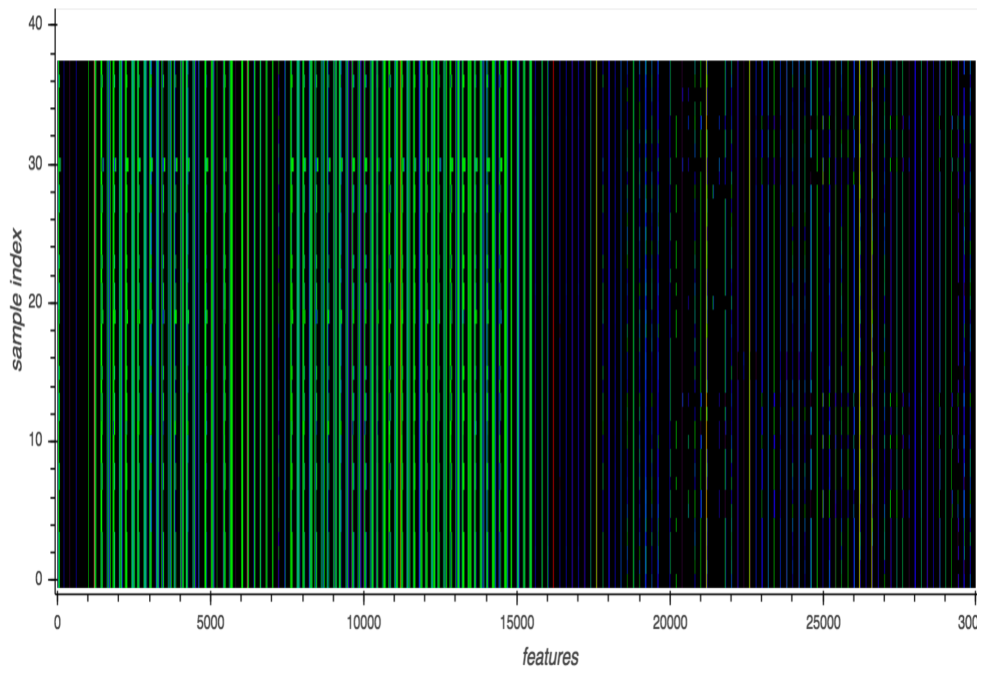


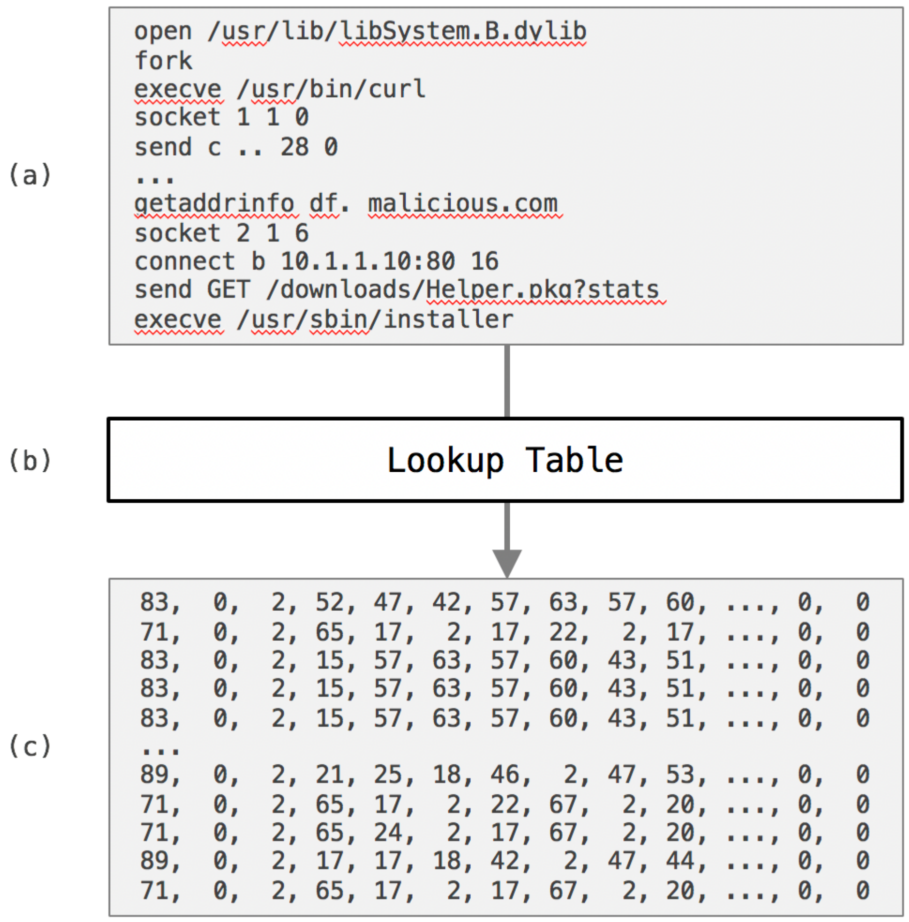
Figure 4.1. Each row represents a per-sample feature, which is a sequence of API call events made issued by a malware sample. Each normalised API call event is rendered as pixel with a unique color assigned via a lookup table. X axis is the feature. Y axis is the sample number. This cluster contains 38 samples (HO_WINPLYER.MSMIU18: 15 samples, OSX_Agent.PFL: 3 samples, OSX_Generic.PFL: 1 sample, OSX_SearchPage.PFM: 18 samples, OSX_WINPLYER. RSMSMIU18: 1 sample)

4.2.1 Dynamic Features

Figure 4.1 illustrates the API call events of the variants of a malware family, which are identified by the model proposed in this chapter. Although there are several variations in the family name, the visual representation of the API call events for this family remains very much isomorphic.

The steps to construct a feature are as follows and an example is shown in Figure 4.2:

1) Map API call name to a unique ID
2) Assign a unique ID for each character found in the API call arguments
3) Pad each API call event with zeros to a predefined fixed size.



```
open /usr/lib/libSystem.B.dylib
fork
execve /usr/bin/curl
socket 1 1 0
send c .. 28 0
...
getaddrinfo df. malicious.com
socket 2 1 6
connect b 10.1.1.10:80 16
send GET /downloads/Helper.pkg?stats
execve /usr/sbin/installer
```

```
83,  0,  2, 52, 47, 42, 57, 63, 57, 60, ..., 0,  0
71,  0,  2, 65, 17,  2, 17, 22,  2, 17, ..., 0,  0
83,  0,  2, 15, 57, 63, 57, 60, 43, 51, ..., 0,  0
83,  0,  2, 15, 57, 63, 57, 60, 43, 51, ..., 0,  0
83,  0,  2, 15, 57, 63, 57, 60, 43, 51, ..., 0,  0
...
89,  0,  2, 21, 25, 18, 46,  2, 47, 53, ..., 0,  0
71,  0,  2, 65, 17,  2, 22, 67,  2, 20, ..., 0,  0
71,  0,  2, 65, 24,  2, 17, 67,  2, 20, ..., 0,  0
89,  0,  2, 17, 17, 18, 42,  2, 47, 44, ..., 0,  0
71,  0,  2, 65, 17,  2, 17, 67,  2, 20, ..., 0,  0
```

Figure 4.2. An example of dynamic execution log transformation. (a) A dynamic execution log. (b) Symbol to ID lookup table constructed by creating a map of symbols from all training samples. (c) A two dimensional array constructed from (a) with padded zeros at the end of each API call event.

A dynamic execution log consists of a sequence of API call events, each of which is broken down into an API identifier and its corresponding API arguments. A sample dynamic execution log is shown in Figure 4.2(a). The symbol lookup table shown in Figure 4.2(b) is constructed by creating a map of all possible call argument characters. This symbol lookup is performed on a per-character basis instead of per-word.

This design decision was made to allow arbitrary number of arguments and values. Figure 4.2(c) shows the final feature transformed from the dynamic execution log, which will be fed to the deep learning model.

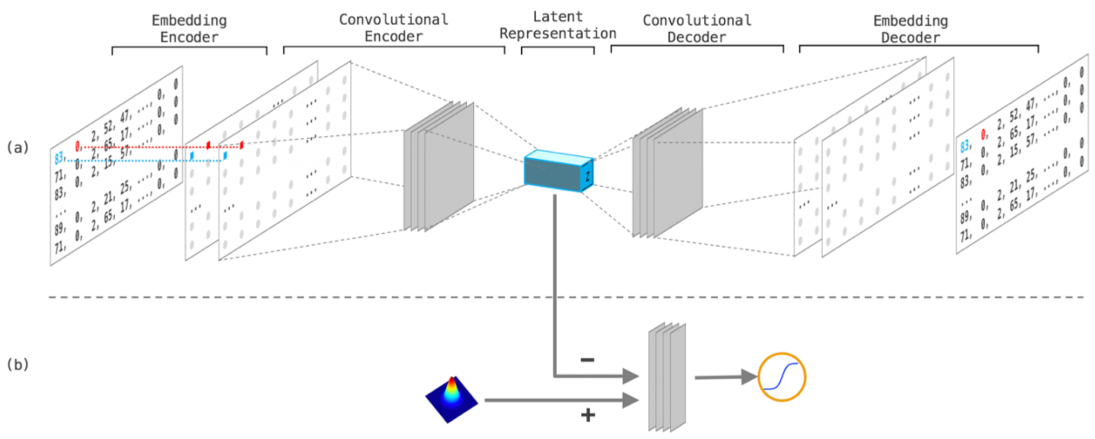


Figure 4.3. Generative adversarial autoencoder with 2D API call event feature and embedding. (a) Autoencoder with each value in 2D feature mapped to an embedding. Note that the value in blue gets mapped to an embedding of size two at its corresponding position marked in dotted blue line. The embedding value is chosen from the embedding lookup table for each symbol at the input feature. The same applies to the value in red. (b) Discriminator with positive samples from Gaussian normal distribution and negative samples from the latent representation, z, obtained from the input feature.

4.2.2 Adversarial Autoencoder with Multi-Dimensional Input

Adversarial Autoencoder [58] forms the core of the model, which consists of an autoencoder and generator-discriminator pair. In addition, DCGAN [35] is used as an autoencoder in order to cope with various perturbations that occur in API call events. The proposed model is jointly trained with stochastic gradient descent by minimising the reconstruction loss on spatio-temporal input x over latent representation z in Equation (1) and by performing min-max adversarial game in Equation (2). Consensus optimisation [62] is also applied on top of Adam optimisation [154] to aid stable GAN [56] training. Batch normalisation [161] is used within the GAN generator to help generate stable latent representations.

$$E_{\boldsymbol{x}}\left[E_{q(\boldsymbol{z}|\boldsymbol{x})}[-\log p(\boldsymbol{x}|\boldsymbol{z})]\right] \tag{1}$$

$$\min_G \max_D \begin{matrix} E_{\boldsymbol{x}\sim p_{data}}[\log D(\boldsymbol{x})] + \\ E_{\boldsymbol{z}\sim p(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z}))] \end{matrix} \tag{2}$$

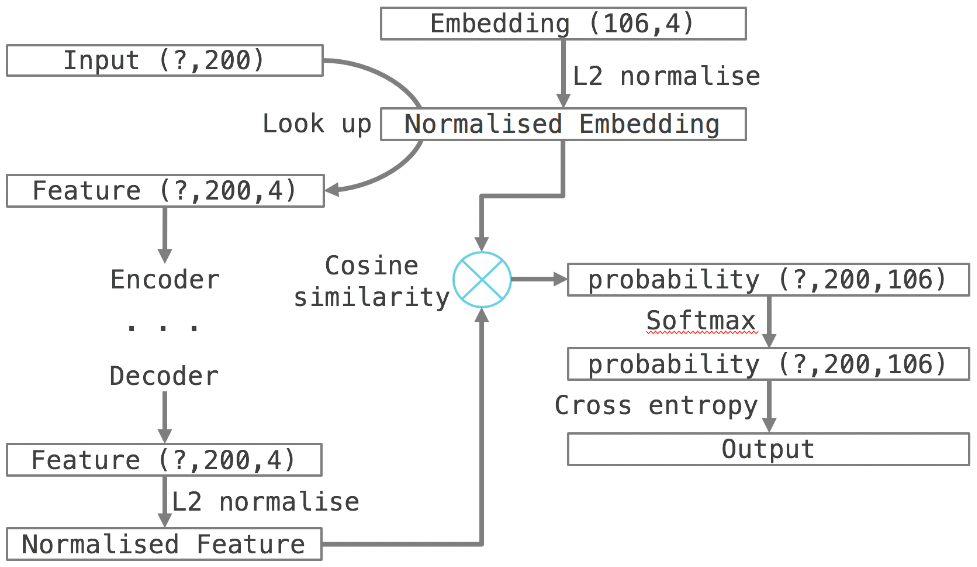


Figure 4.4. Autoencoder loss calculation diagram.

However, several key adjustments are made on top of the base adversarial autoencoder model to fit the two-dimensional symbol inputs into the model and to generate consistent latent representation over variable length inputs.

The input feature is a two-dimensional array with its width fixed. As shown in Figure 4.3(a), each symbol at the input is mapped to an embedding vector using a embedding lookup table, which transforms the 2D symbol inputs to 3D array required by the convolutional encoder. The autoencoder cost calculation process is shown in Figure 4.4 for each API call event where feature length is limited to 200, embedding vector size is 4, and symbol lookup table size is 106. The variable length API call event is denoted as '?'. The reconstructed symbol probability is calculated via cosine similarity obtained as an inner product of L2 normalised reconstructed input and L2 normalised embedding [70]. The output in Figure 4.4 contains the cross entropy between the one hot encoded input symbol and the softmax'ed reconstructed symbol probability. The autoencoder cost is calculated by taking the mean of the sum of the negative log likelihood value at each symbol position in the input.

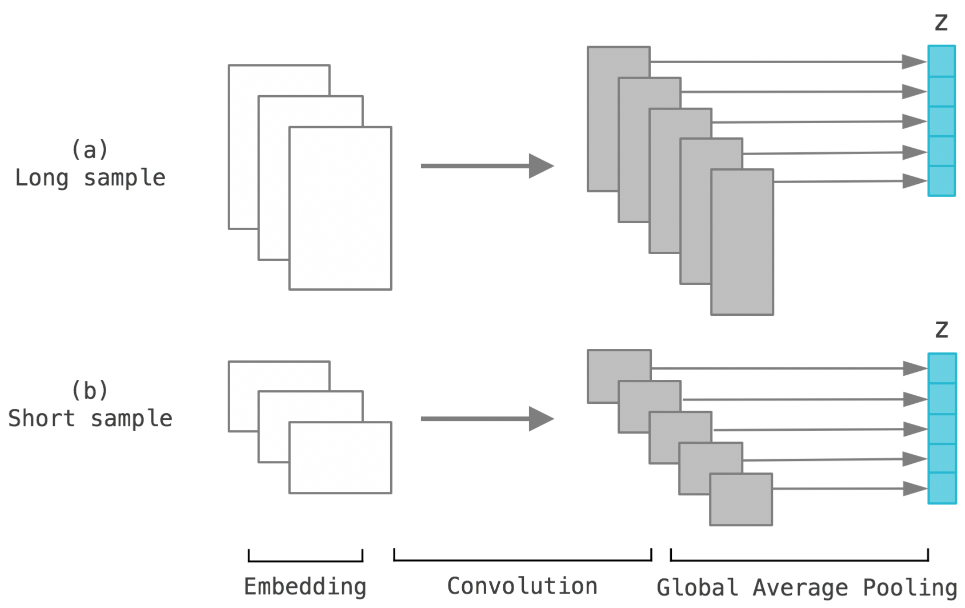


Figure 4.5. A fixed size *z* is required for similarity comparison. (a) shows a sample with larger number of API call events causing larger kernel volume. (b) shows a sample with relatively smaller number of API call events leading to smaller kernel volume. Both (a) and (b) must generate a *z* with identical size for similarity comparison.

Further, variable length inputs pose a challenge in producing consistent latent representations. Although the convolution operation can be performed against arbitrary length inputs, the size of the latent representation, z, can vary depending on the input length, which is not a desirable property when a fixed size *z* is needed for similarity comparison during detection. Figure 4.5 illustrates the need for pooling the variable size convolutional filters produced by variable length inputs. While global max pooling works well when the features are extracted for classification problems, it performs poorly for autoencoders because each *z* bit gets biased to a relatively larger value, saturates to a distinct bit range, generates poor reconstruction, and therefore negatively impacts the accurate clustering of similar samples. Our experiments show that global average pooling performs better for autoencoders. Notably Gaussian normal distribution that drives adversarial training is nicely imposed on z bits with global average pooling (See Figure 4.6).

As shown in Figure 4.7, global average pooling is performed on the last convolutional layer of the encoder by producing the *z* bits from each filter's mean value (blue part of Figure 4.7). Then *z* is unpooled by setting bit values to the decoder filters at the argmax index locations saved from the encoder filters (red part of Figure 4.7).

4.2.3 Mean Squared Error Metric

The latent representations obtained through adversarial autoencoder contain real numbers which can be directly compared with each other by calculating mean squared error (MSE).

If binary bits are desired for efficient comparison, z needs to be binarized using the bitwise mean value of the training samples, which splits the bit distributions into halves and subsequently helps better to distinguish between samples. Adversarial training driven by Gaussian normal distribution significantly helps in spreading the bits evenly. Hamming distance is then used to compute the distance for the given two latent vectors. However, our experiment indicates that MSE metric over real valued z performs slightly better than hamming distance over binary z bits.

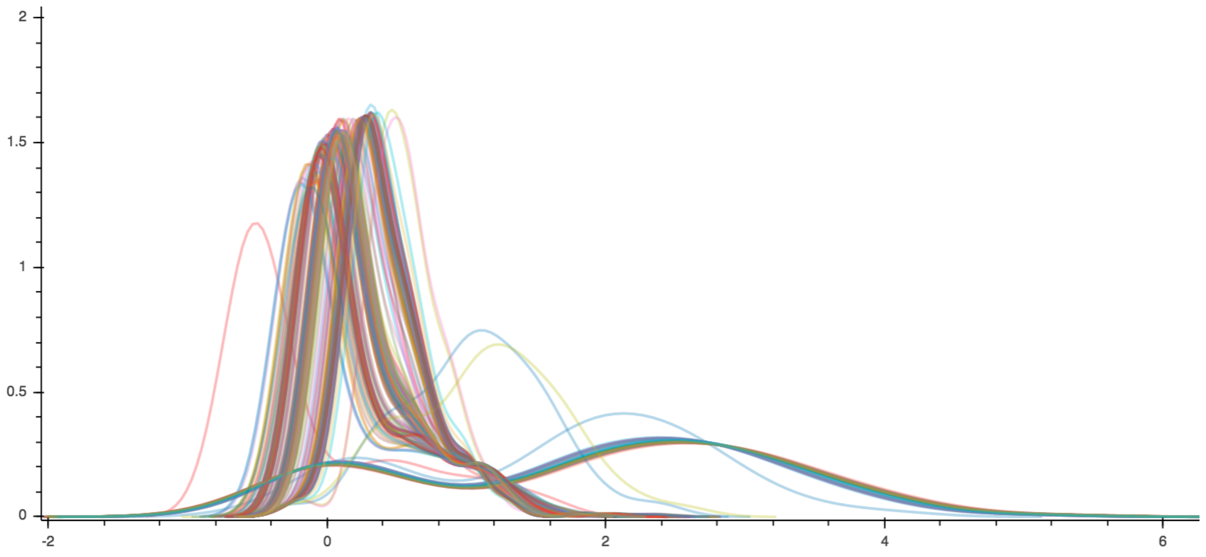


Figure 4.6. Distribution of each bit in z.

Each bit has Gaussian normal distribution as instructed by adversarial training, which helps distinguishing between different samples. Global average pooling allows this adversarial property as well as sound reconstruction required for autoencoder.

4.3 Evaluation

4.3.1 Dynamic Sample Dataset

Dynamic execution logs of 2,855 in-the-wild OS X malware samples collected from a proprietary commercial sandbox and 7,541 benign OS X samples were used for evaluation. A snapshot of family distribution for a portion of 2,855 in-the-wild malware samples is shown in Figure 4.8.

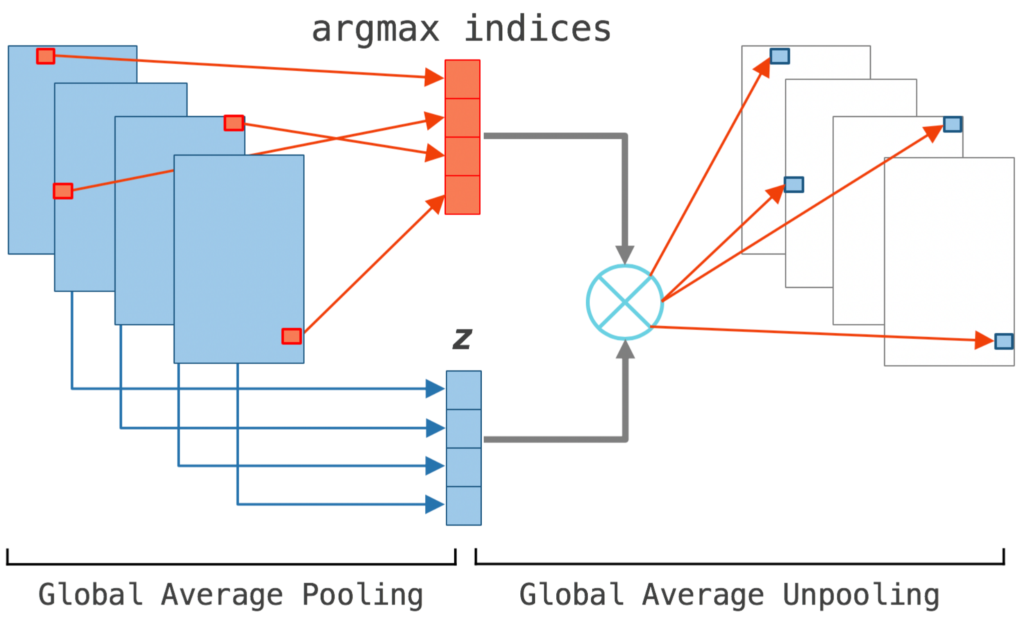


Figure 4.7. Global average pooling, latent representation (z) generation, and global average un-pooling. Argmax of each encoder filter is marked in red. The mean value of each encoder filter is marked in blue.

Dynamic execution logs were analysed and labelled by human experts with the focus on the similarity of API call event sequences. Out of 2,855 malicious samples, 353 unique API call sequence patterns have been identified and were used for training. Note that the benign samples were not included in the training set in order to test the outbreak detection capability as outlined at the introduction of this chapter.

Gradient boosting, Support Vector Machine and Random Forest models were chosen as baseline that the proposed model is evaluated against. The two dimensional variable length input feature vector used for the proposed model collapses to a one dimensional array in order to produce n-gram feature [149] for the baseline models. Note that clustering models were not included in the baseline since they need decent number of training samples, which is different to the problem setting put forward in this research.

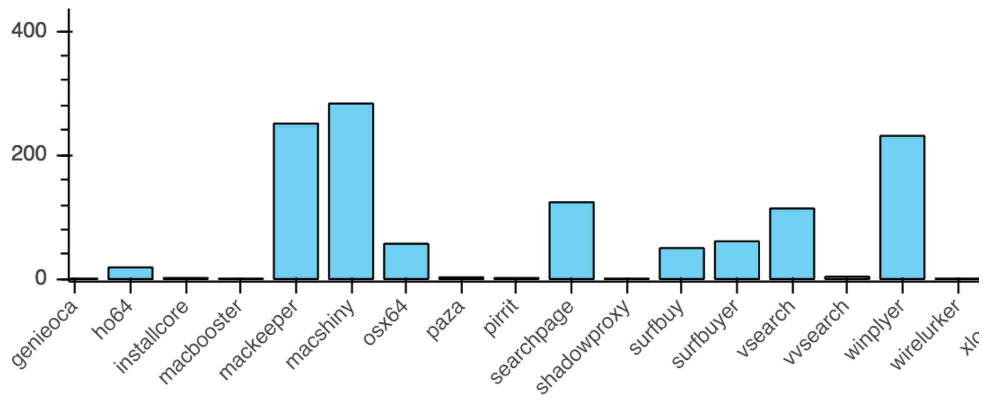


Figure 4.8. A snapshot of the family distribution for a portion of 2,855 in-the-wild malware samples used in the evaluation.

Since the baseline classifiers perform differently depending on whether the benign samples are included in the training set or not, two separate experiments were conducted in this research The samples were split for the two experiments and are shown in Table 4.1. Note that the benign samples in training set and those in test set are mutually exclusive in experiment 2.

	Experiment 1		**Experiment 2**	
	malicious	**benign**	**malicious**	**benign**
train	353	**0**	353	**3770**
test	2855	**7541**	2855	**3771**

Table 4.1. Dynamic detection sample splits

4.3.2 Dynamic Detection Result

To simulate the outbreaks, we have assigned a unique label for each of these 353 unique malicious training samples and have evaluated how accurately the label matches with the prediction. Therefore, True Positive (TP) ratio in Table 4.2 shows the label-wise matches of all test malicious samples against malicious training samples.

Baseline models did not produce a good TP ratio against malicious samples scoring less than 50% detection rate regardless of the sample splits in both experiments. They also catastrophically failed the False Positive (FP) test in experiment 1 where no benign samples were used for training. The definitions of sensitivity and specificity including TP ratio and FP ratio can be found in [162]:

Model	Experiment 1		Experiment 2	
	TP	**FP**	**TP**	**FP**
gradient-boosting-1gram	0.167	1.000	0.194	0.026
gradient-boosting-2gram	0.126	1.000	0.108	0.027
svm-1gram	0.367	1.000	0.056	0.000
svm-2gram	0.394	1.000	0.056	0.000
randomforest-1gram	0.452	1.000	0.460	0.000
randomforest-2gram	0.356	1.000	0.385	0.000
aae-mse	**0.991**	**0.001**		

Table 4.2. Dynamic detection evaluation result

Threshold	TP	FP
0.000500	1.000	0.306
0.000100	1.000	0.057
0.000075	1.000	0.033
0.000050	1.000	0.010
0.000025	**0.991**	**0.001**
0.000010	0.860	0.000

Table 4.3. Effect of thresholds in dynamic detection

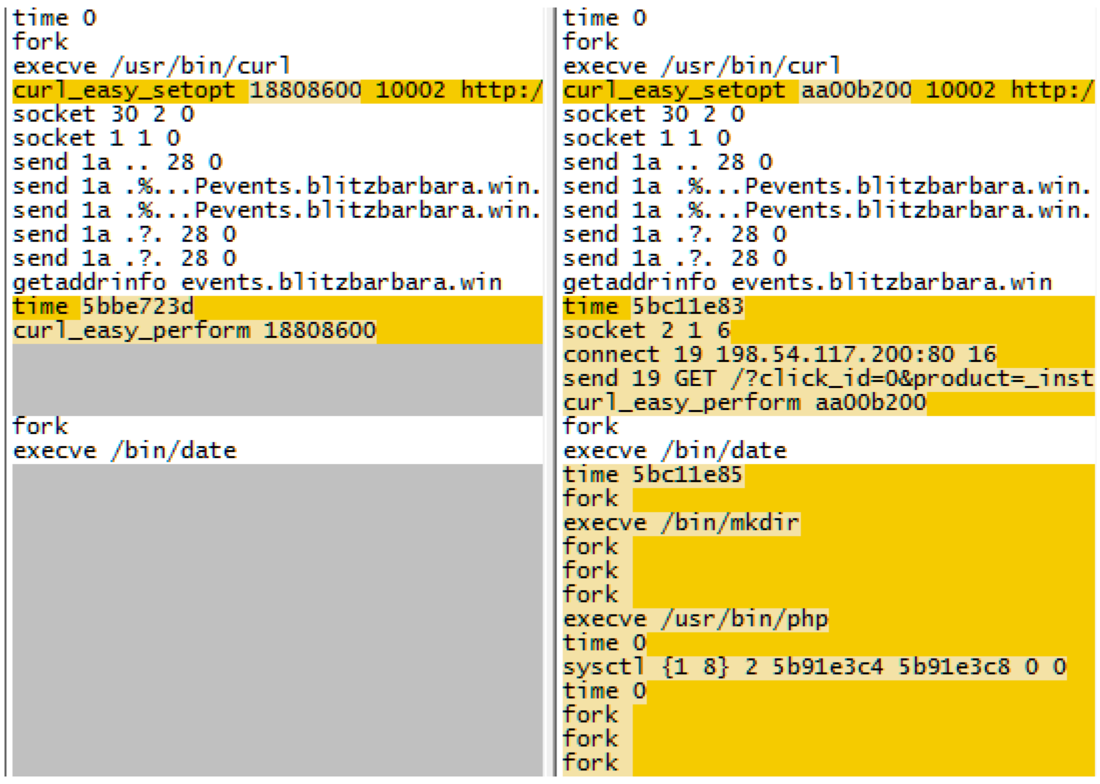
However, they recorded near perfect FP ratio for experiment 2 where large benign samples were included in the training set, which meets the expectation for a good classifier. On the contrary, aae-mse*,* the proposed generative adversarial autoencoder using mean squared error (MSE) as autoencoder loss, shows 99.1% true positives along with 0.1% false positives with MSE decision threshold set to 0.000025. This threshold influences how much the model generalises the detection.

The higher the threshold is set, the more reliable the detection is whereas the more likely the model misses mutated samples. The effect of various threshold values is shown in Table 4.3, and the highlighted threshold shows the performance described in Table 4.2. One can run the inference with a sufficiently low threshold for accurate detection while running a separate inference with a higher threshold to grab potentially malicious samples for further analysis.

In summary, the experiments show that the proposed model is capable of reliably generalising the sample variations while keeping a sufficiently fair distance from previously unseen benign samples. In contrast, traditional classifiers are much less likely to work to detect under the malware outbreak situations.

4.3.3 Dynamic Detection Analysis

In this section, we analyse several key aspects of how the latent representation produced by generative adversarial autoencoder is correlated to the semantics of API call events. Figure 4.9 shows an example of two similar samples detected by the proposed model. The majority of the API call events remain identical (marked in white background). A sequence of API calls appears (marked in yellow background), initiated by a successful network connection establishment in the sample on the right hand. Those API calls are absent (marked in grey background) in the sample on the left hand. This demonstrates the robust detection capability of the model even when samples execute different code paths in different operational environments.

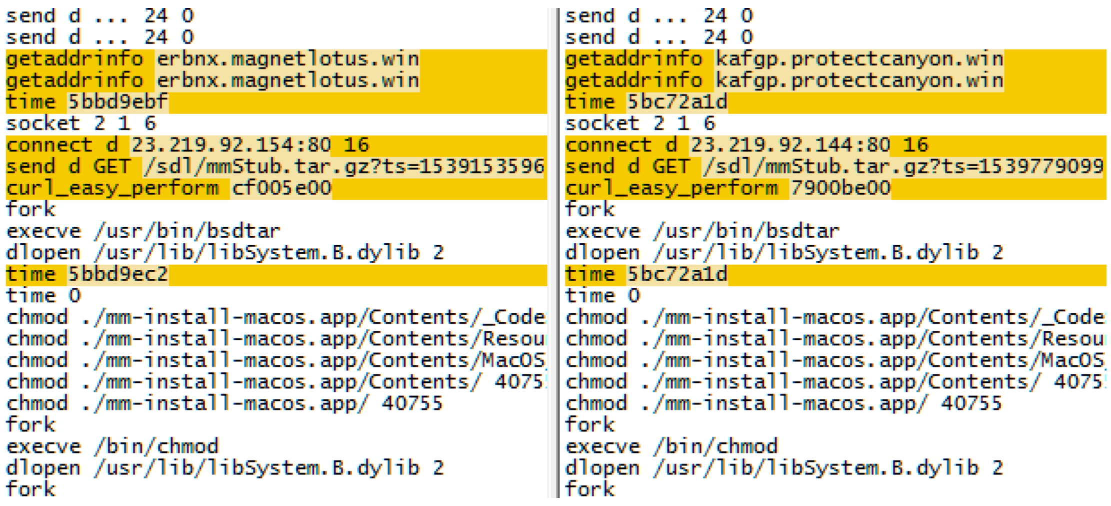


```
time 0
fork
execve /usr/bin/curl
curl_easy_setopt 18808600 10002 http:/
socket 30 2 0
socket 1 1 0
send 1a .. 28 0
send 1a .%...Pevents.blitzbarbara.win.
send 1a .%...Pevents.blitzbarbara.win.
send 1a .?. 28 0
send 1a .?. 28 0
getaddrinfo events.blitzbarbara.win
time 5bbe723d
curl_easy_perform 18808600



fork
execve /bin/date
```

```
time 0
fork
execve /usr/bin/curl
curl_easy_setopt aa00b200 10002 http:/
socket 30 2 0
socket 1 1 0
send 1a .. 28 0
send 1a .%...Pevents.blitzbarbara.win.
send 1a .%...Pevents.blitzbarbara.win.
send 1a .?. 28 0
send 1a .?. 28 0
getaddrinfo events.blitzbarbara.win
time 5bc11e83
socket 2 1 6
connect 19 198.54.117.200:80 16
send 19 GET /?click_id=0&product=_inst
curl_easy_perform aa00b200
fork
execve /bin/date
time 5bc11e85
fork
execve /bin/mkdir
fork
fork
fork
execve /usr/bin/php
time 0
sysctl {1 8} 2 5b91e3c4 5b91e3c8 0 0
time 0
fork
fork
fork
```

Figure 4.9. An example of API call snippets of two similar samples detected by the proposed model. White lines indicate identical events. Yellow lines indicate non-identical events due to a difference in the event. Gray lines indicate absent events relative to the other event.

Figure 4.10 shows API call snippets of OSX_Bundlore.ESS2 (on the left hand) and OSX_Bundlore.PFL (on the right hand side) which shows that aae-mse found that these samples are similar to each other. Although the URL and the IP address of the remote command and control server have changed, aae-mse successfully identified this variation.



```
send d ... 24 0
send d ... 24 0
getaddrinfo erbnx.magnetlotus.win
getaddrinfo erbnx.magnetlotus.win
time 5bbd9ebf
socket 2 1 6
connect d 23.219.92.154:80 16
send d GET /sdl/mmStub.tar.gz?ts=1539153596
curl_easy_perform cf005e00
fork
execve /usr/bin/bsdtar
dlopen /usr/lib/libSystem.B.dylib 2
time 5bbd9ec2
time 0
chmod ./mm-install-macos.app/Contents/_Code
chmod ./mm-install-macos.app/Contents/Resou
chmod ./mm-install-macos.app/Contents/MacOS
chmod ./mm-install-macos.app/Contents/ 4075
chmod ./mm-install-macos.app/ 40755
fork
execve /bin/chmod
dlopen /usr/lib/libSystem.B.dylib 2
fork
```

```
send d ... 24 0
send d ... 24 0
getaddrinfo kafgp.protectcanyon.win
getaddrinfo kafgp.protectcanyon.win
time 5bc72a1d
socket 2 1 6
connect d 23.219.92.144:80 16
send d GET /sdl/mmStub.tar.gz?ts=1539779099
curl_easy_perform 7900be00
fork
execve /usr/bin/bsdtar
dlopen /usr/lib/libSystem.B.dylib 2
time 5bc72a1d
time 0
chmod ./mm-install-macos.app/Contents/_Code
chmod ./mm-install-macos.app/Contents/Resou
chmod ./mm-install-macos.app/Contents/MacOS
chmod ./mm-install-macos.app/Contents/ 4075
chmod ./mm-install-macos.app/ 40755
fork
execve /bin/chmod
dlopen /usr/lib/libSystem.B.dylib 2
fork
```

Figure 4.10. An example of API call snippets of two similar samples detected by the proposed model. Both samples are distinguishable by the access to different remote network nodes.

The cluster detected by aae-mse shown in Figure 4.11 contains 209 different samples of 6 different variants of a family. It also detected similar samples across multiple different malware families as shown in Figure 4.12.

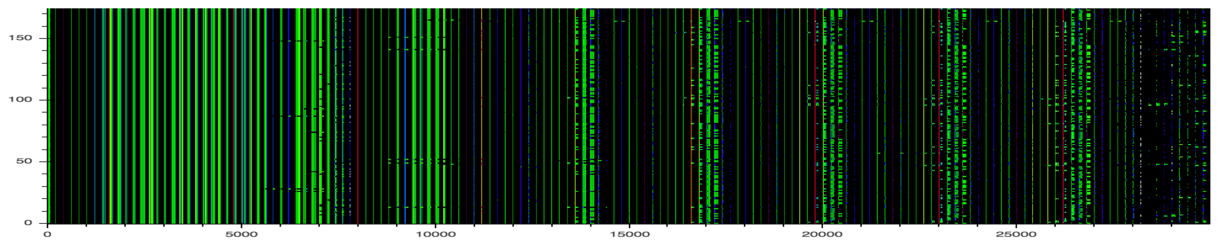


Figure 4.11. Cluster 2 (nsamples=209) consists of OSX_Bnodlero.PFL (2 samples), OSX_Bundlore.PFL (88 samples), OSX_Bundlore.PFM (109 samples), OSX_BundloreCA.PFL (1 samples), OSX_BundloreCA.PFM (4 samples), and OSX_SurfBuy.ESS2 (5 samples).

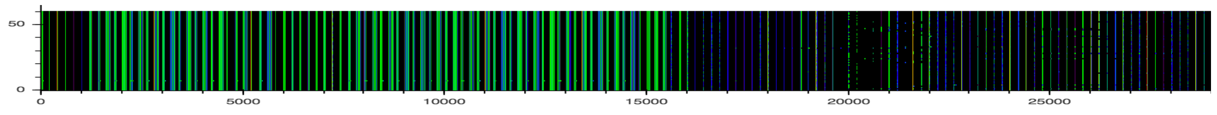


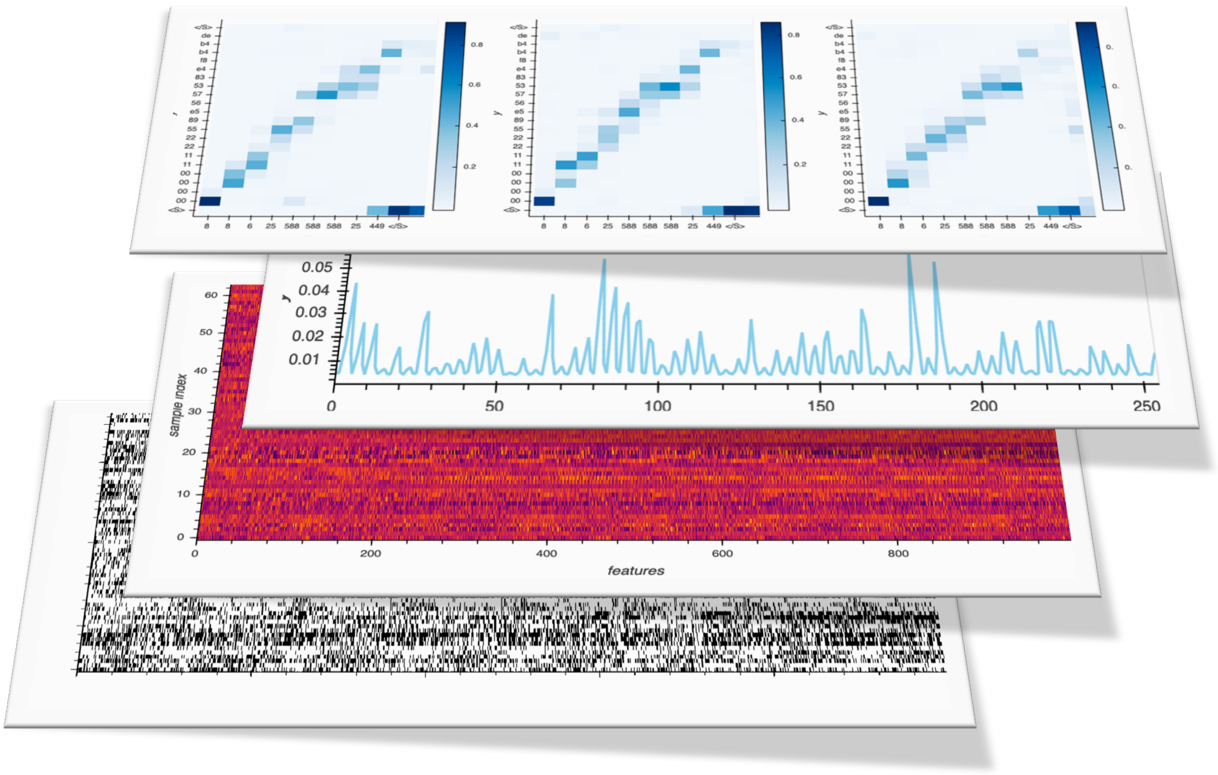
Figure 4.12. Cluster 4 (nsamples=61) consists of HO_WINPLYER.MSMIU18 (31 samples), OSX_Agent.PFL (1 samples), OSX_SearchPage.PFM (3 samples), and OSX_WINPLYER.RSMSMIU18 (26 samples).

4.4 Conclusion

Proposed generative adversarial autoencoder with the use of API call event as features can produce good latent representations with small training set that can be used for distance-based similarity between samples. The training set restriction seriously tests the generalisation capability of the model. The result shows that the proposed model provides effective detection for gradually diverging mutation of malware species in behaviours. The model was also found to be effective in discovering multiple heterogeneous malware families that share similar dynamic execution events.

This chapter demonstrated how generative adversarial autoencoder can be leveraged for dynamic malware detection. In chapter 5, more semantic malware detection method using instruction cognitive signal will be presented.

Chapter 5: Instruction Cognitive Malware Detection



The work in this chapter was published in following paper:

Sean Park, Iqbal Gondal, Joarder Kamruzzaman, Jon Oliver, " Instruction Cognitive One-Shot Malware Outbreak Detection" 2019 26th International Conference on Neural Information Processing of the Asia-Pacific Neural Network Society. Sydney, 2019.

Previous two chapters presented an efficient malware detection method using deep learning over instruction feature. This chapter will show a method that extracts highly accurate instruction signals from raw binary executables and that leverages them for malware detection.

New malware outbreaks cannot provide thousands of training samples which are required to counter malware campaigns. In some cases, there could be just one sample. So, the defence system at the firing line must be able to quickly detect many automatically generated variants using a single malware instance observed from the initial outbreak by statically inspecting the binary executables. As previous research works show, statistical features such as term frequency–inverse document frequency and n-gram are significantly vulnerable to attacks by mutation through reinforcement learning. Recent studies focus on raw binary executable as a base feature which contains instructions describing the core logic of the sample. However, many approaches using image-matching neural networks are insufficient due to the malware mutation technique that generates a large number of samples with high entropy data. Deriving instruction cognitive representation that disambiguates legitimate instructions from the context is necessary for accurate detection over raw binary executables. In this chapter, we present a novel method of detecting semantically similar malware variants within a campaign using a single raw binary malware executable. We utilize Discrete Fourier Transform of instruction cognitive representation extracted from self-attention transformer network. The experiments were conducted with in-the-wild malware samples from ransomware and banking Trojan campaigns. The proposed method outperforms several state-of-the-art binary classification models.

5.1 Introduction

Modern malware authors write a script that automatically generates an arbitrarily large number of diverse samples that share similar characteristics in program logic, which is a very cost-effective way to evade detection with minimum effort. A series of malware outbreaks that stems from the same automated script constitutes a malware campaign. The majority of traditional approaches make an assumption that a large number of training samples are available, searching the model space for a presumably perfect parameters that perfectly fits the training samples. These approaches tend to overfit the lab samples while not generalising well against real world samples, let alone the training sample availability assumption is far from practical. Given that it is crucial to detect these seemingly diverse but identically rooted variants by utilising a single malware instance observed from the initial outbreak, this chapter will focus on one-shot training optimized for one-to-many malware detection. To the best of our knowledge, no work has been conducted on one-shot training over raw binary executables.

Saxe at al. [116] and Vinayakumar et al. [163] implemented a deep feed forward network using statistical features derived from the executable file metadata. Anderson et al. [27] showed that it is easy to defeat these statistical features largely from the executable header metadata such as import table entries, sections, entropy, and other relevant metadata. Anderson's experiment shows a significant evasion rate even without intensive fine-tuning of the random mutation performed at each iteration of reinforcement learning, which essentially demonstrates the vulnerability of statistics-based detection approaches.

Byte n-gram has been considered an attractive approach when dealing with highly structured data such as raw executable files. However, Zak et al. [148] showed that byte n-gram learns little information from code sections contrary to common hypotheses in machine learning. Then Raff et al. [133] discovered a potential that neural network models can learn useful representation from uninterpreted sequence of executable bytes that helps classification. In addressing the problems faced by the above approaches and devising a technique capable of learning from a single instance of the initial outbreak, this chapter makes following contributions:

1) Developments of a method that learns a representation directly correlated with the legitimate instructions embedded within the binary executable file.
2) Development of method to use the model to detect malware variants that possess instruction-wise similarity by performing one-shot training.

5.2 Methodology

5.2.1 Transformer Network

Figuring out valid instructions from raw sequence of bytes in executables requires a model that understands the semantic relationship between the elements of the input sequence. A plethora of deep learning techniques have been produced in early 21st century. Notably CNN and RNN along with Generative Adversarial Network (GAN) [56] have been the base platform for language modelling and machine translation. Despite its success, correctly identifying the instructions purely based on the context remained challenging until Transformer network [79] was proposed. For our purpose, Transformer network can be trained by providing raw sequence of bytes from an executable as an input and by setting the desired instructions as a target (See Figure 5.1).

Line	Input (raw bytes)	Instruction disassembly	Output (opcodes)
1	**89** e1	mov ecx, esp	**89** 64
2	**b8** 32 00 00 00	mov eax, 0x32	**b8** 64 64 64 64
3	**83** ec 3c	sub esp, 0x3c	**83** 64 64
4	**8b** 35 03 6c 40 00	mov esi, dword ptr [0x406c03]	**8b** 64 64 64 64 64
5	**56**	push esi	**56**
6	**ff 15** c4 50 40 00	call dword ptr [0x4050c4]	**ff 15** 64 64 64 64
7	**2e** ba c2 37 40 00	mov edx, 0x4037c2	**2e** 64 **c2** 64 64 64
8	ff e2	jmp edx	**64 64**
9	00 00	add byte ptr [eax], al	64 64
10	00 8b 3d ff 6b 40	add byte ptr [ebx + 0x406bff3d], cl	64 64 64 64 64 64
11	00 57 ff	add byte ptr [edi - 1], dl	64 64 64
12	15 8c 50 40 00	adc eax, 0x40508c	64 64 64 64 64

Figure 5.1. The first column is line number. The second column shows input raw bytes to Transformer model. The third column is the disassembly for the input. The last column shows the output of the Transformer. All numbers are in hexadecimal while the line numbers in the first column is in decimal. Legitimate instructions are shown until line 8, and the following bytes are data bytes.

The goal of the model is to produce correct opcodes at its output while padding the rest of the bytes with 64, which indicates INVALID. As highlighted in blue, the model correctly identifies opcodes until it starts outputting 64 (INVALID) from line 9. The model correctly disregards invalid instructions as highlighted in the disassembly in red from line 9, by filling output bytes with 64 (INVALID). Although the model's output (last column) is not correct at line 8 when transitioning from the end of the code block to the beginning of data block, the model mostly produces accurate outputs. In short, self-attention enables Transformer to find out the relationship between different positions of the input sequence. With LSTM model [124], the experiment shows that output instruction sequence is far from accurate in the presence of packed data, which suggests self-attention plays a key role in predicting the opcodes.

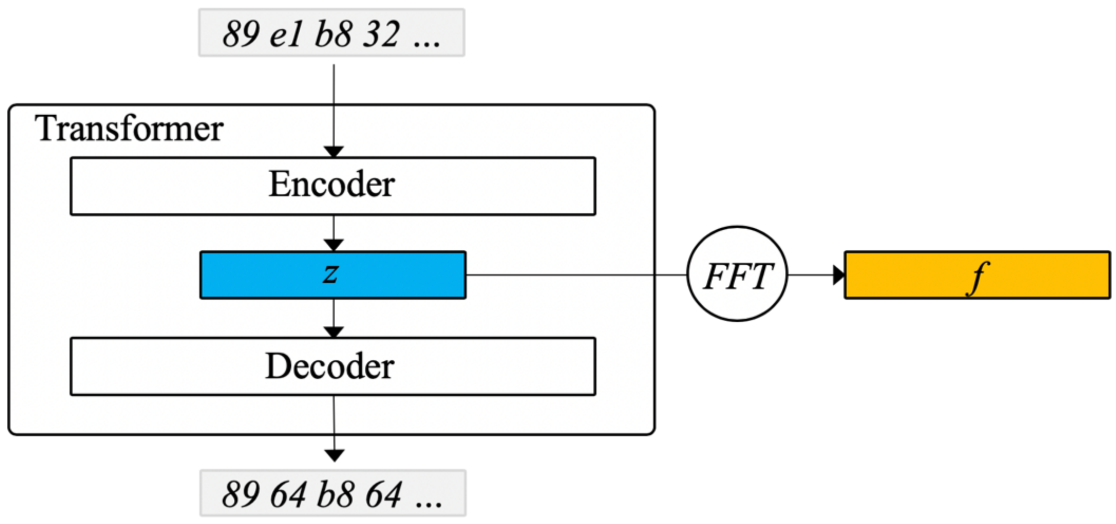


Figure 5.2. Model architecture using Transformer. Frequency spectrum of approximated encoded latent representation is used as the feature for malware detection.

5.2.2 Model Architecture

Transformer network [79] is used as a base model to produce instruction cognitive signals for the raw input sequence. The model is trained using hold-out dataset that consists of both malicious and benign samples. Let $\boldsymbol{x} = (x_1, \dots, x_n) \in \mathbb{R}^{\boldsymbol{n}}$ be an input sequence of symbols, $\boldsymbol{z} = (z_1, \dots, z_n) \in \mathbb{R}^{\boldsymbol{d \times n}}$ be latent representation of dimension $\boldsymbol{d}$ retrieved from the encoder output, and $\boldsymbol{y} = (y_1, \dots, y_n) \in \mathbb{R}^{\boldsymbol{n}}$ be the desired output sequence with the opcode placed at the beginning of each valid instruction and INVALID symbol in the rest of the positions. $\boldsymbol{z}$ is learned by minimizing the softmax cross entropy loss of the decoder output, $\hat{\boldsymbol{y}}$, against the label $\boldsymbol{y}$, using adam optimizer [154]. The model architecture is shown in Figure 5.2. The trained $\boldsymbol{z}$ has instruction cognitive signals that can directly transform the raw byte sequences into a sequence of legitimate instructions.

Adversarial attack by manipulating the bytes is a threat to the success of the model as described in Section 2.4. Figure 5.3 illustrates an example of $\boldsymbol{z}$ for a malware sample with the majority of the executable occupied by high entropy packed data. This demonstrates that the model is resilient to adversarial perturbations modifying the binary executable by inserting arbitrary bytes without caring about the legitimacy of instructions.

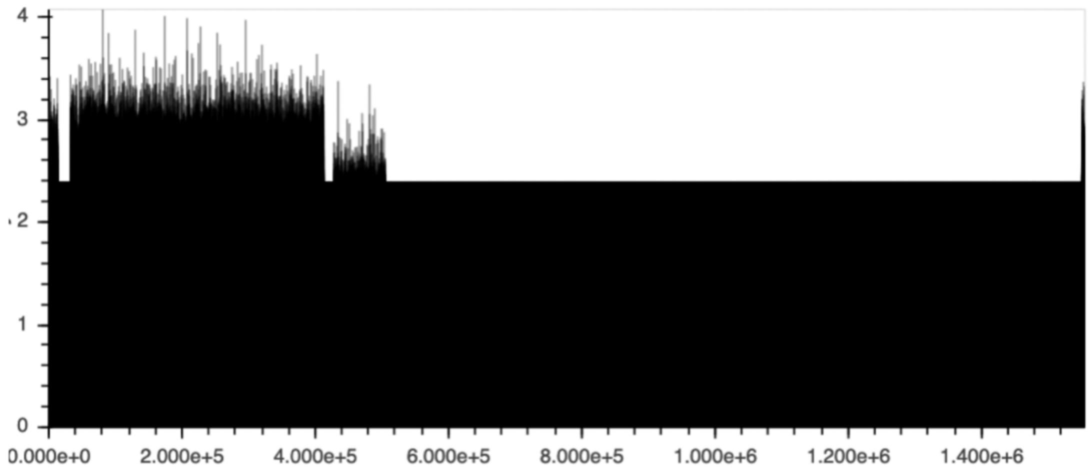


Figure 5.3. Maximum activation for backdoor TORFSEE.SMF. There are several intervals where the activation strength is flat where no valid instruction was found.

As described in Section 2.3, it is critical to detect diverse malware variants deploying metamorphism. Given that frequency spectrum exhibits a coherent view of the features correlated to instructions while staying resilient to minor variations, we perform discrete Fourier transform, $\boldsymbol{z} \rightarrow \boldsymbol{f}$ [164]. However, the dimension of the learned latent representation $\boldsymbol{z}$ is reasonably large which prevents us from performing multi-dimensional Fourier transform due to high computational complexity. Therefore, we approximate $\boldsymbol{z}$ by taking the most active neuron for each $\boldsymbol{z_t} \in \mathbb{R}^{\boldsymbol{d}}$ across dimension $\boldsymbol{d}$.

$$\boldsymbol{a}_t = argmax\ \boldsymbol{z_t} \qquad (1)$$

$$\boldsymbol{f}_k = \sum_{t=0}^{N-1} a_t \cdot e^{-\frac{2\pi i}{N}kt} \qquad (2)$$

where $\boldsymbol{f}_k \in \mathbb{R}^{\mathbf{1}}$ and $k = (0, \ldots, N-1)$. We use FFT [30] in order to compute the frequency coefficients faster.

We discovered that the samples sharing similar instruction-wise characteristics exhibit similar spectrum distributions. Therefore, we use Pearson Correlation Coefficient [165] against spectral density as a distance metric between samples, which is defined in equation (3).

$$\boldsymbol{\rho}_{a,b} = \frac{E[(a-\mu_a)(b-\mu_b)]}{\sigma_a \sigma_b} \qquad (3)$$

where $\boldsymbol{\sigma}_a$ is the standard deviation of a, $\boldsymbol{\sigma}_b$ is the standard deviation of b, $\boldsymbol{\mu}_a$ is the mean of a, $\boldsymbol{\mu}_b$ is the mean of b, and E is the expectation.

A sample is detected as malicious if the correlation, defined by the equation (3), to a known malware instance in a malware campaign is within the threshold, which needs to be empirically decided depending on the dataset.

5.2.3 Detection Analysis

Figure 5.4 shows the FFT of two separate malware campaigns captured in the wild. Each graph contains two variants exhibiting their frequency spectra overlapped to each other. Variants from the same campaign show similar spectral characteristics while the difference in spectra from different campaigns is distinct enough to distinguish them.

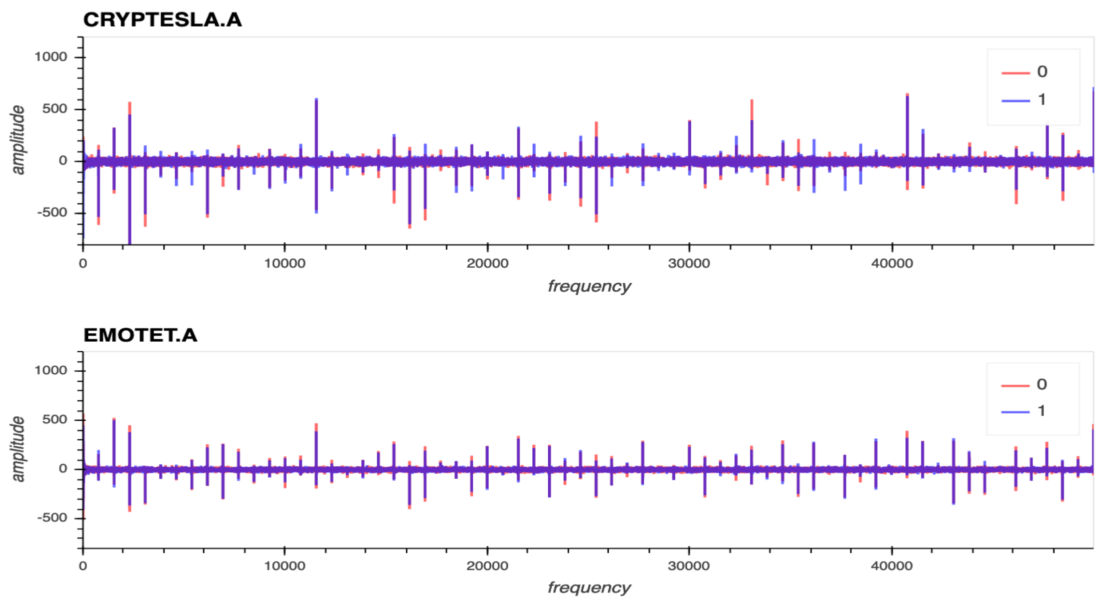


Figure 5.4. The top graph shows FFT of CRYPTESLA variants whereas the bottom graph displays that of EMOTET variants. The difference between families is clearly distinctive while the samples within the family are kept close in frequency distribution.

5.3 Evaluation

5.3.1 Instruction Cognitive Dataset

As stated at the introduction, one-shot training is used to evaluate the model's performance on one-to-many detection capability. There is no publicly available dataset for this problem setting. Repurposing public datasets for our problem setting is not optimal because some datasets come without binary samples, and others contain imbalanced samples with no campaign information, which makes it difficult to derive an accurate evaluation of the model's capability to detect malware variants originated from the same campaign. Besides most datasets are old and are not annotated with first-seen timestamp.

For these reasons, we use a proprietary dataset provisioned by a commercial vendor that contains major ransomware and banking Trojans campaigns of 2017 and 2018. Each individual malware outbreak has been recorded along with its time and the binary sample. The largest campaigns are shown in Figure 5.5.

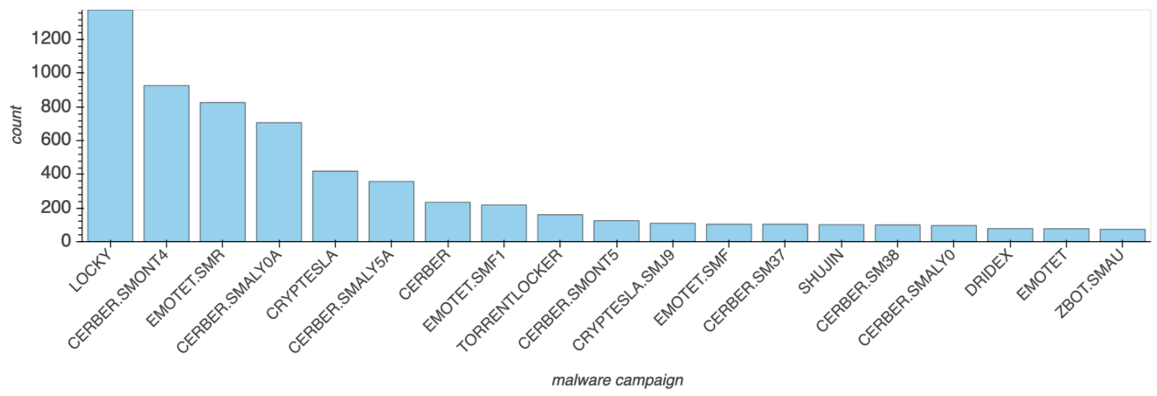


Figure 5.5. A snapshot of the malware campaign distribution of the dataset used in the evaluation. X-axis is the name of the malware campaign and Y-axis is the number of samples within each malware campaign. Shown from the largest campaign (left) to the smaller ones (right).

Gradient Boosting, Support Vector Machine, and Random Forest were selected as baseline models to evaluate our proposed model against. Benchmarking against clustering methods is deemed irrelevant to our problem setting since only a single sample from each campaign is available for training and therefore, cluster around each campaign cannot be formed. We used a single malware sample first seen in each campaign for training, which counts to 488. 20% of total benign samples were used for training while the rest of them were used for validation (see Table 5.1). Training and validation sets are mutually exclusive. Note that our proposed model did not use benign training samples. In addition, our model does not need extra training for malicious training samples once Transformer has been trained to recognize instructions using off-the-dataset samples. We use malicious training samples for distance computation only.

	malicious	benign	
		baselines	our model
Train	488	1365	0
Validation	3085	5461	5461

Table 5.1. Instruction cognitive detection: train and validation dataset split

5.3.2 Model Performance

As shown in Table 5.2, our proposed model (transformer+fft) outperforms all models in True Positive despite the fact that no benign sample was used for training. Our model marginally comes in the second place for False Positive following SVM. However, SVM records a poor TP, which is sub-optimal to be used as a production model.

Model	TP	FP
Gradientbooster-unigram	0.967	0.1518
Gradientbooster-bigram	0.986	0.0957
SVM-unigram	0.656	0.0016
SVM -bigram	0.853	**0.0016**
RandomForest-unigram	0.981	0.1648
RandomForest -bigram	0.982	0.1168
Transformer+FFT	**0.997**	**0.0190**

Table 5.2. Instruction cognitive model performance comparison

ROC of the decision threshold for the equation (3) is shown in Figure 5.6.

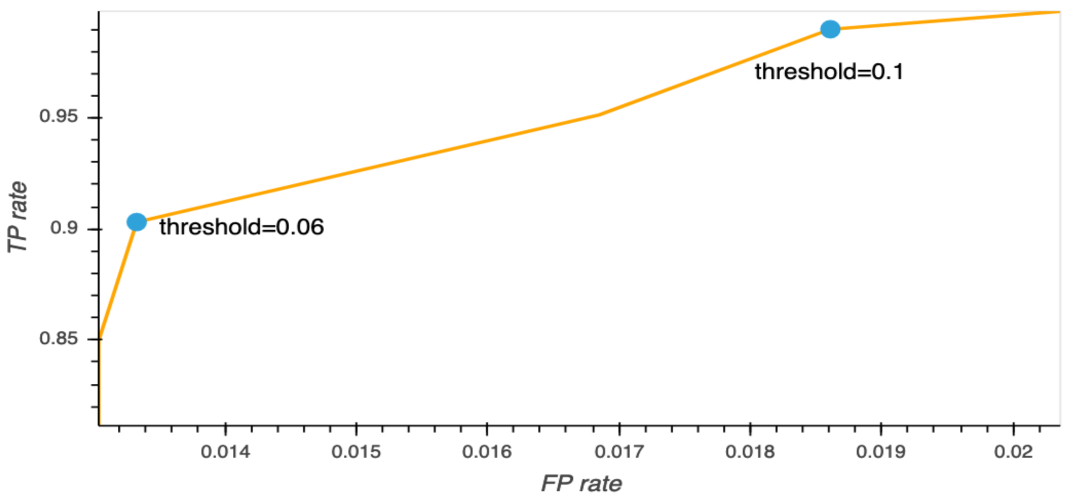


Figure 5.6. ROC of decision threshold of the model as defined by equation (3).

5.3.3 False Positive Complexity

It is important to understand how scalable a given model's FP is especially when a tiny percentage of false positives can cause catastrophic failure for real world traffic since the volume is amplified by several magnitudes. We measured the FP by changing the benign training set size. Figure 5.7 illustrates that the FP complexity of the baseline models is close to O(n), showing strong dependency on the amount of the benign training samples whereas the FP complexity of transformer+fft is O(1) because it does not use any benign training samples for detection, which shows a great resiliency of our model against false positives.

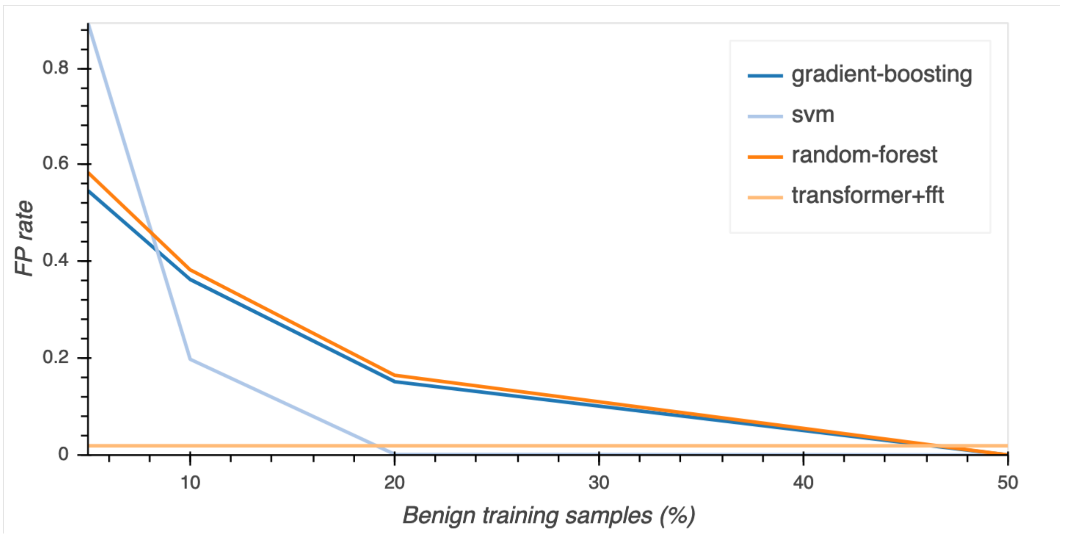


Figure 5.7. FP rate of the baseline models when the amount of benign samples varies. transformer+fft model that does not use benign training samples is also shown for comparison.

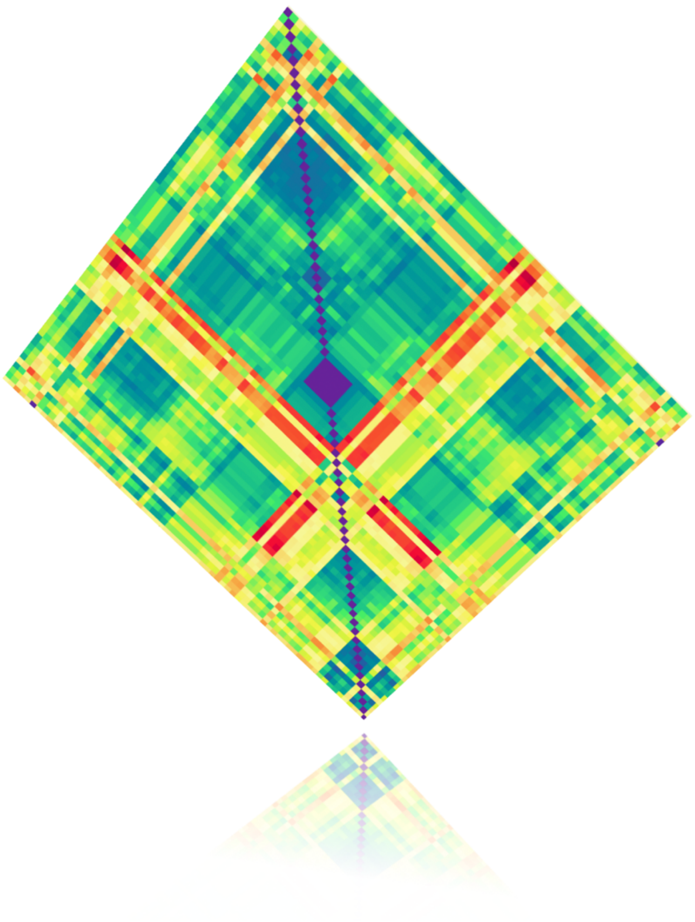
5.4 Conclusion

In this chapter, we presented a novel method that extracts instruction cognitive representation from uninterpreted raw binary executables. This method can be used for one-to-many malware detection via one-shot training against frequency spectrum of the Transformer's encoded latent representation. The method works regardless of the presence of diverse malware variations while remaining resilient to adversarial attacks that mostly use random perturbation against raw binaries.

One significant advantage of the method is that no computationally expensive training is required each time a new malware sample is added once Transformer is fully trained to produce the representation sufficient to recognize instructions within the binary sequence.

In this chapter, we demonstrated a method that extracts semantic features from the raw binary executables, which is critical in accurate malware detection in the presence of adversarial attacks that target the vulnerabilities of modern machine learning models. Next, we will look into an effective hunting approach to triage suspicious samples through a comprehensive study on metamorphic threat detection.

Chapter 6: Threat Hunting against Metamorphic Threats



The work in this chapter was accepted in following book chapter:

Sean Park, Iqbal Gondal, Joarder Kamruzzaman, Jon Oliver, "Comparative Study on Machine Learning Methods to Detect Metamorphic Threats" Malware Analysis using Artificial Intelligence and Deep Learning" Springer, 2020.

Many effective malware detection methods using deep learning has been demonstrated in the last three chapters. In this chapter, we will show the significance of clustering against metamorphic threats and provide a benchmarking result over several representative models including deep learning method presented in this thesis.

As the sheer volume of incoming samples exponentially increases every year, one of the critical tasks in defence pipeline is to identify probabilistically suspicious samples that require further investigation. The task essentially involves filtering out a large volume of unknown samples and producing a manageable set of potentially malicious samples with sufficiently high confidence. Today the majority of malware samples possess metamorphic property where various mutations over the original set of code blocks occur before they are released. This arbitrary custom-designed mutation algorithm applied at each outbreak constitutes the crux of the constant battle between the attackers and the defenders. Therefore it is crucial to capture this non-linear metamorphic pattern unique to the mutation in order to detect the variants. This chapter compares the performance of various clustering methods against metamorphic malware samples to identify the model that best suits in practical threat hunting. Results have shown that Adversarial autoencoder performs better than well-known techniques such as. HDBSCAN, KNN, and SDHASH.

6.1 Introduction

Modern threats are created using a script that automatically generates an arbitrarily large number of diverse samples that share similar characteristics in program logic, which is a very cost-effective way to evade detection with minimum effort. For instance, a series of malware outbreaks originated from the same automated script constitutes a malware campaign that distributes seemingly diverse but identically rooted variants. This diversity comes from metamorphism that forms the basis of modern threats [41].

As the scale of incoming samples rises, many machine learning methods have been attempted to detect those threats. However, the survey by Ye et al. [166] essentially shows that it is very difficult to accurately classify malware samples. With the lack of machine learning models with the desired accuracy over real world samples, triaging unknown samples has been increasingly important. In particular, it is required to produce a smaller number of potentially malicious samples which human analysts can manually inspect or other tools with finer-grained detection capability can analyse.

Although several triaging methods have been previously published with respectable results [145][146], it has always been challenging to gain certain degree of confidence in the correctness of triage. The triage result is more reliable when it is interpretable. Particularly the extensive use of black box dataset with unknown characteristics used in most studies makes it difficult to validate the triage result for unlabelled samples. Considering this limitation, given that the vast majority of modern malware samples exhibit similar patterns in the automatically generated instruction sequences, we argue that triage result is less likely to be incorrect if instruction-wise characteristics are similar. For instance, let us consider Figure 6.1 that shows the variants of a Cerber malware family captured in the wild that spanned across several weeks. Similar instruction-wise characteristics are present in basic block (BBL) level, function level, or somewhere in between. A reasonable degree of newly introduced or removed code fragments are observed, which can be considered as a minor noise. Despite various metamorphic techniques in place, the instruction patterns for this malware campaign are clearly similar to each other to humans. Even though a significant portion of the code is related to unpacking the hidden code, this surface level instructions still carry the semantics of the metamorphic code combined with the original unpacking code.

This chapter focuses on static instructions as a base feature, which possesses semantic representation of a potentially metamorphic samples. Statistical features are not considered in this chapter due to its inherent vulnerability demonstrated by the evasion method based on reinforcement learning [27]. In this chapter, the instruction IDs are used as a base feature instead of raw binary instructions [34] in order to avoid instruction differences caused by register transposition metamorphism [41] and language compiler differences.

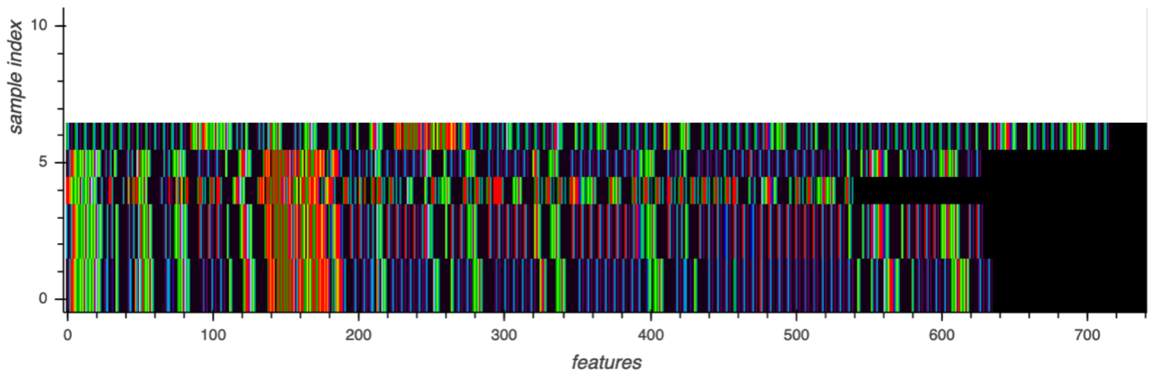


Figure 6.1. Visualisation of 7 Ransomware Cerber variants. Each row represents a per-sample feature, which is a sequence of instructions of a malware sample. Each instruction is rendered as horizontal stripes with a unique colour assigned to different instructions. X axis is the feature. Y axis is the sample number.

To the best of our knowledge, no comparative study has been conducted on the efficacy of semi-supervised learning methods against metamorphic samples. The main contribution of this chapter is the comparison of various machine learning methods that best detects malware variants with similar campaign-wise metamorphic similarity. Our study shows that Adversarial Autoencoder has shown better performance as compared with other techniques.

6.2 Malware Triage Methods

There are a plethora of machine learning methods that can be used for malware triage. We selected four different methods for comparison that meet the following requirements:

- The model must be widely deployed in similarity detection field.
- The model must provide a method to find the closest training sample for a given test sample since the similarity of a test sample is evaluated by checking if its closest training sample and itself belong to the same cluster.
- The model must be able to predict in the absence of total cluster count information. Since most threats evolve over time with minor changes, no explicit decision boundary between adjacent clusters exist in practice. In fact, many clusters exhibit similar metamorphism with variations. Therefore, the evaluation based on strict cluster matching can interfere with correct detection of the threats.

We briefly describe the four methods in the light of the above selection requirements.

6.2.1 K-Nearest Neighbour

K-Nearest Neighbour (KNN) [167] is one of the most widely used machine learning methods in similarity search. Despite its long history, it still shows reasonably competitive results in many research papers and provides non-parametric interface for ease of use. The hyper parameters include search algorithm and distance metric. The implementation can leverage either KDTree [168] or BallTree [169] for efficient pairwise distance computation. Since KNN requires fixed length inputs, it uses n-gram of instruction feature as input, as discussed in the next section. We use BallTree to cope with high dimensional n-gram variables with 'minkowski' having the parameter equal to 2 for distance metric [170]

6.2.2 HDBSCAN (Hierarchical Density-Based Spatial Clustering of Applications with Noise)

HDBSCAN [171] is widely used in unsupervised clustering method that creates clusters with varying densities. As with KNN, n-gram is used as input due to fixed input restriction of the model. 'euclidean', which is identical 'minkowski' with parameter of 2, is used for distance metric of HDBSCAN. The primary hyper parameters include the following:

```
min_cluster_size=5,   min_samples=None,   metric='euclidean',
alpha=1.0
```

Different ranges of these hyper parameters have been experimented during the evaluation and the best result is compared against the rest of the models.

6.2.3 SDHASH (Similarity Digest Hash)

SDHASH, also known as Fuzzy hashing, has been widely used when determining the similarities of documents by maximising the probability of local hash collision. Popular algorithms include SDHASH [172], SSDEEP [173], and TLSH [174]. SSDEEP calculates edit distance to recognise identical blocks, TLSH uses n-gram frequency distribution for similarity measure, and SDHASH uses normalized entropy measure to find statistically improbable features. Studies show SDHASH and TLSH outperforms SSDEEP [175][176]. In this chapter, we show the results for SDHASH because TLSH shows similar results. Since fuzzy hashing can deal with variable length input, raw instruction feature is used as input to the model. The distance is calculated by normalising the inverse of the similarity score returned by SDHASH.

6.2.4 Adversarial Auto-Encoder

Adversarial Auto-Encoder (AAE) is a dimensionality reduction method based on deep learning that compresses an arbitrary length input to a fixed length latent representation. The model implemented by Park et al. [34] is used for the evaluation, which combines the power of generative adversarial network [58] and location-resilient convolutional neural network [35]. Raw instruction feature is fed to the model as input. The distance is computed by taking the mean squared error of the difference of *z* values, which are the latent representations at the bottleneck.

6.3 Evaluation

6.3.1 Metamorphic Malware Dataset

In this chapter, we use interpretable family-wise metamorphic malware samples that possess similar instruction-wise characteristics instead of arbitrarily chosen malware dataset from the black box. To deal with the biased ground truth cluster problem [144], we carefully created the dataset for the experiment in such a way that each cluster holds the core semantics of campaign's metamorphism. A commercial vendor provided 3,390 unique malware samples from multiple isolated in-the-wild campaigns, which is a representative dataset from a larger sample pool. The 3,390 samples are further reduced to 1,012 samples after removing duplicate samples with identical instruction sequence are removed. Using various machine learning methods combined with manual analysis, these 1,012 instruction-unique samples have been identified to form 211 clusters. We used half of each cluster for training and use the rest for testing. In addition, we reduced a large pool of benign testing samples to 5,854 by taking representative samples with similar TLSH digest values. The dataset for the experiment is summarised in Table 6.1.

	Malicious	Benign
Train	485	0
Test	527	5854

Table 6.1. Metamorphic clustering: train and test dataset split

The malicious samples contain major ransomware, backdoor, and banking Trojans in Windows operating system. The colour-coded instructions of randomly chosen families are depicted in Figure 6.2.

6.3.2 Instruction Feature

As described in [34], instruction sequences are extracted from each binary sample from which instruction IDs are calculated using capstone [177]. Input feature is essentially a sequence of variable length integers representing instruction characteristics. Models can choose different types of random variables for the input. KNN and HDBSCAN require fixed length inputs whereas SDHASH and AAE accept variable length inputs. To utilise the maximum potential of the model, SDHASH and AAE use the original instruction feature with minimum pre-processing. For KNN and HDBSCAN, we deploy n-gram, which is one of the most widely used tools that generate fixed length inputs. We use n=2, which generates 7,302 features for the instruction feature.

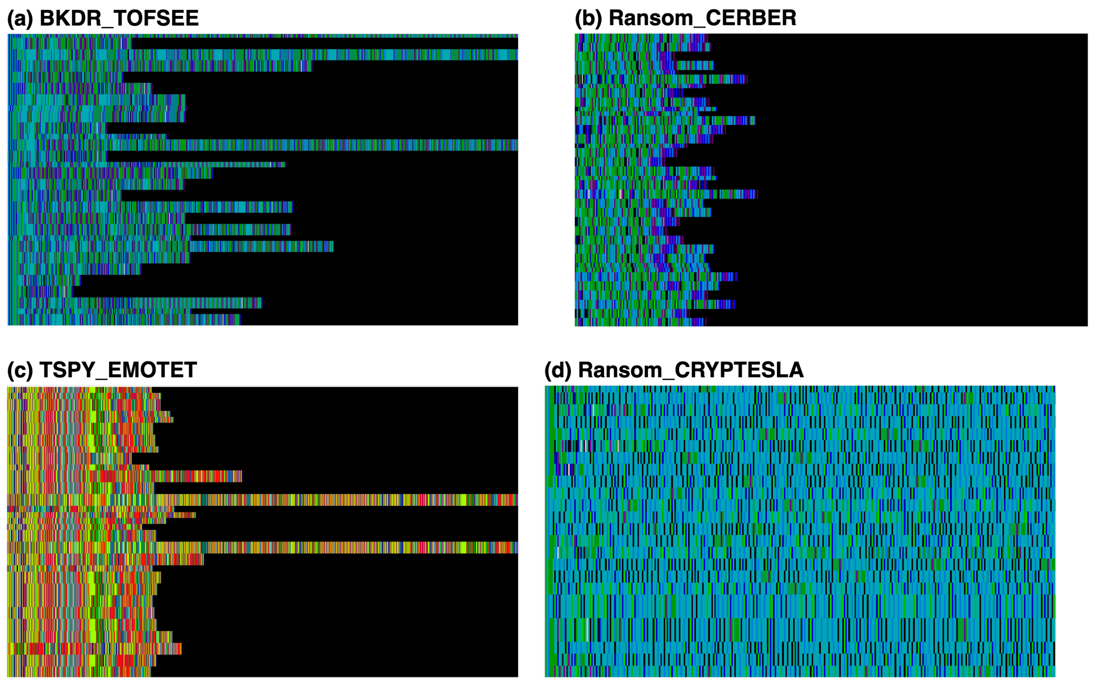


Figure 6.2. Horizontal lines represent samples showing each family can contain variable length malware variants exhibiting different instances of metamorphism.

6.3.3 Evaluation Criteria

Let *ground_truth* be a mapping $f: x \rightarrow cluster$ where x is the input feature prepared for each model as explained in the previous section and $cluster$ is a label obtained through the process described above. Let *is_similar* be a function defined by each model that calculates the distance between the first argument and the second argument and returns the closest sample within the second argument, and *threshold* be the hyper parameter for each model. The process for strict evaluation is described by the algorithm in Figure 6.3. The algorithm sets the decision metrics (*tp*, *tn*, *fp, fn*) if the predicted cluster number strictly matches the true cluster number. In addition to this strict matching criteria, relaxed evaluation is also performed to see if models are capable of separating malicious samples from benign samples given that our ultimate goal is malware detection. In relaxed evaluation, an exception is made to line 16 of the algorithm in which *tp* is incremented if both *ytrue* and *ypred* have a valid malicious cluster number.

Algorithm: Process for strict evaluation

```
1:  train_xs = x for all x ∈ train_malicious
2:  test_xs = x for all x ∈ {test_malicious, test_clean}
3:  tp = 0, tn = 0, fp = 0, fn = 0
4:  for test_x in test_xs:
5:      ytrue = ground_truth[test_x]
6:      Distance, similar_train_x = is_similar(test_x, train_xs)
7:      if distance < threshold:
8:          ypred = ground_truth[similar_train_x]
9:      else:
10:         ypred = 0
11:     if ytrue == ypred:
12:         if ytrue == 0: tn += 1
13:         else: tp += 1
14:     else:
15:         if ytrue == 0: fp += 1
```

```
16          else: $fn$ += 1
```

Figure 6.3. The algorithm used to compute the performance metrics of the models. *tp*, *tn*, *fp*, *fn* describe the count of True Positive, True Negative, False Positive, and False Negative, respectively.

6.3.4 Malware Triage Benchmarking Results

The model performances are depicted as ROC curves in Figure 6.4 and Figure 6.5. The top performing model is AAE, followed by KNN with a decent gap. SDHASH finds the correct malware clusters approximately for 50% of the test samples. On the contrary, HDBSCAN performed extremely poorly in strict setting while its accuracy remains around 50% in relaxed setting.

First of all, the AAE model that utilises convolutional autoencoder performs global max pooling at the last encoder layer, which captures translation invariant features insensitive to locational variations of the instructions. Furthermore, generative adversarial network finetunes the latent representation in a non-cooperative minmax game, making it more resilient to artificially generated similar samples. The result also shows locality sensitive hashing does not adapt well in identifying the similarities between metamorphic malware samples in the same cluster. Above all the fact that metamorphism involves frequent code transpositions renders location dependent SDHASH simply ineffective in clustering similar malware samples. In the meantime, KNN performs reasonably well although insufficient to compete with AAE. A straightforward Euclidean distance comparison over location-independent histogram of n-gram appears to support KNN's decent performance. Finally, HDBSCAN shows poor performance with the result unpredictable both in strict and relaxed settings. Especially HDBSCAN in strict setting shows worse results than random decisions.

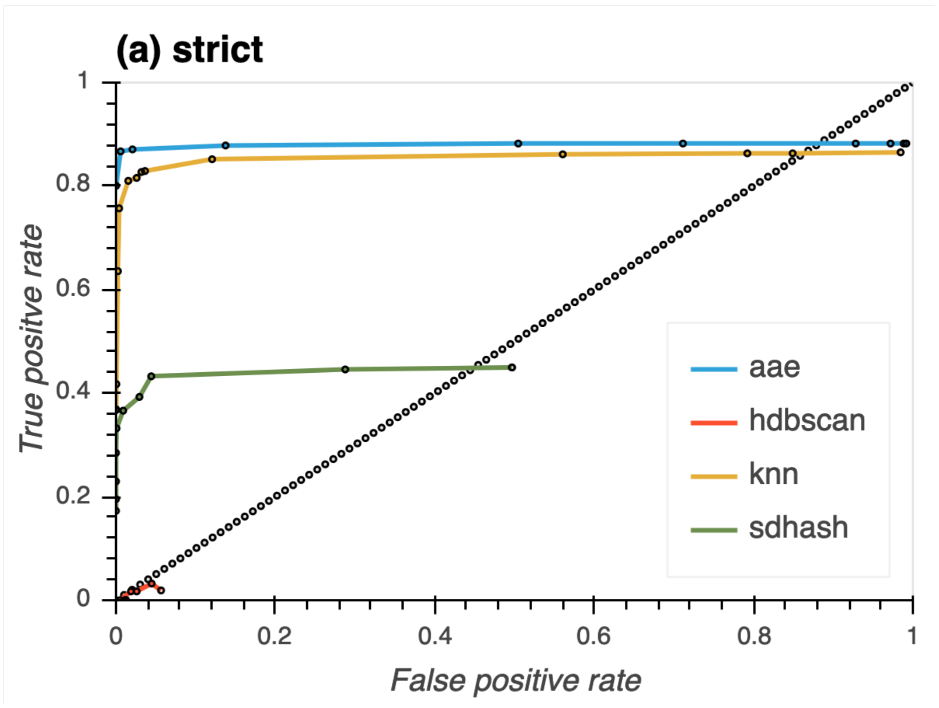


Figure 6.4. (a) ROC of model performances with strict matching.

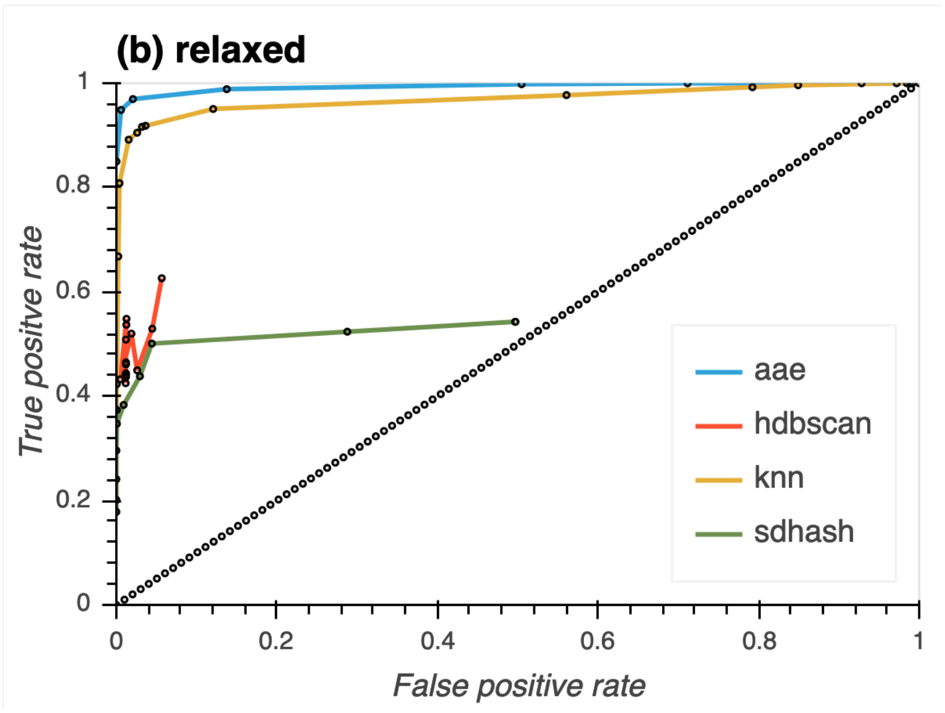


Figure 6.5. (b) ROC of model performances with relaxed matching criteria.

6.4 Conclusion

In this work, we presented a comprehensive comparison of various machine learning methods on their capabilities in identifying the variants of malware families that show similar characteristics in their core instructions. The result shows adversarial network combined with convolutional network with global max pooling outperforms the rest of the machine learning techniques in this task. Since it is critical to generalise the distribution under metamorphic threats, this result suggests that models with translation invariant property is more resilient to metamorphic threats than the ones that leverages traditional distance metrics.

In chapter 7 of the thesis, conclusions are presented based on contribution chapters 3, 4, 5 and 6.

Chapter 7: Conclusion

In previous chapters, it has been demonstrated that deep learning is capable of tackling various difficult challenges posed by modern malware threats. Neurally identifying complex non-linear patterns is increasingly important especially when malware diversity is accelerated with a high volume of potentially suspicious samples arriving every day. This chapter summarises the primary contributions of the thesis and sets out future works.

7.1 Summary of Contributions

The contributions of the deep learning based malware detection in this thesis is summarised below.

1. **Generative static malware outbreak detection**: Using the data model of original instruction sequence with order retained, the proposed model overcomes imperfect labelling problem and provides maximum information possible required for deep learning to create accurate latent representation. The generative power of GAN by use of AAE makes one-shot training possible, which is a critical requirement in production environment. The translation invariant property achieved through stacked CNN enables the detection of many metamorphic characteristics present in the instruction sequence. The unsupervised learning through autoencoder in combination with semantic hashing allows a lot of freedom to finetune detection in practical defence systems using adjustable distance threshold. Performance evaluation against representative traditional supervised models confirms the superiority of the proposed model.

2. **Generative dynamic malware outbreak detection**: The static outbreak detection model has also demonstrated its efficacy for dynamic detection. DCGAN is used to cope with the increased number of dimensions introduced by the use of dynamic API call event sequence at the input. For the variable length inputs, the upgraded model produces consistent size of latent

representation from the autoencoder by using global average pooling that allows distance calculation for similarity comparison. The experimental results again confirm the efficacy of the model, outperforming many state-of-the-art traditional machine learning approaches by a significant margin. Despite the training set size restriction that seriously tests the generalisation capability, the model effectively discovers not only intra-family variants but also heterogeneous malware families that exhibit similar dynamic execution behaviour.

3. **Instruction cognitive malware detection**: The proposed approach performs an important task of extracting instruction-aware signals from the raw binary executables whose structure is completely unknown. Equipped with self-attention capability, the Transformer model learns a representation directly correlated with the legitimate instructions embedded within the binary, which helps identifying semantically similar malware variants. Pearson Correlation Coefficient against spectral density computed by Discrete Fourier Transform is used for similarity comparison. The efficacy of the instruction cognitive representation was demonstrated with the evaluation against traditional supervised models. False Positive complexity of the model shows O(1) dependency on the training set size, which shows great resiliency against false positives. In contract, False Positive complexity of all traditional models is close to O(n), which shows unacceptable sensitivity to real world samples whose volume is several magnitudes higher.

4. **Comparative study on metamorphic threats**: One of the critical tasks in defence pipeline is threat hunting that filters out a large volume of unknown samples, producing a manageable set of potentially malicious samples with sufficiently high confidence. A comparative study has been conducted on the efficacy of semi-supervised learning methods malware variants with similar campaign-wise metamorphic similarity. A set of widely used methods from each field were selected for benchmarking, which include a similarity search algorithm – KNN, an unsupervised clustering method – HDBSCAN, a fuzzy hashing method – SDHASH, a deep learning based dimensionality reduction method – AAE. It has been found that adversarial network with CNN and global

max pooling performs the best in both multi-class classification and binary classification tasks.

7.2 Future Works

Recently malware are also appearing in the forms other than executable binary such as Microsoft's PowerShell, JavaScript, and VB Script. Hendler et al. [178] implemented CNN-based classifier that detects malicious PowerShell scripts. The generative models with variable length inputs proposed in this thesis are expected to outperform Hendler's naïve CNN over truncated one-hot encoded input features let alone the requirement of outbreak detection leaves the generative models as the sole option. Further research can be carried out for script malware by taking advantage of generative outbreak detection approaches presented in this thesis.

The instruction cognitive Transformer model described in Chapter 5 can provide a highly accurate approximated instruction sequence. Creating an express pipeline from raw binary executable straight to the detection would be a good extension to this thesis by feeding the semantic instruction signal from Transformer model to the generative outbreak detection model described in Chapter 3.

Despite its capability to capture highly complex non-linear metamorphism present in the instructions, the proposed models in this thesis still confined to the surface level features. For instance, detailed program analysis such as discovering Command and Control URL and decryption keys is beyond the scope of this thesis. Obtaining high level semantics by capturing the logical relationships in the sequence requires further research as the development in deep learning progresses.

Overall, the developed techniques can be applied to provide solutions in other areas as well.

References

[1] Wiki, "Cyber-attack," *http://en.wikipedia.org/wiki/Cyber-attack*. http://en.wikipedia.org/wiki/Cyber-attack.

[2] Wiki, "Malware," *Malware*. http://en.wikipedia.org/wiki/Malware.

[3] F. Guo, P. Ferrie, and T. C. Chiueh, "A study of the packer problem and its solutions," 2008, doi: 10.1007/978-3-540-87403-4_6.

[4] K. A. Roundy and B. P. Miller, "Binary-code obfuscations in prevalent packer tools," *ACM Computing Surveys*. 2013, doi: 10.1145/2522968.2522972.

[5] J.-M. Borello and L. Mé, "Code obfuscation techniques for metamorphic viruses," *J. Comput. Virol.*, vol. 4, no. 3, pp. 211–220, 2008.

[6] Symantec, "Internet Security Threat Report Volume 24," *https://www.symantec.com/content/dam/symantec/docs/reports/istr-24-2019-en.pdf*, 2019. .

[7] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surv.*, vol. 44, no. 2, p. 6, 2012.

[8] B. K. Mishra and N. Jha, "Fixed period of temporary immunity after run of anti-malicious software on computer nodes," *Appl. Math. Comput.*, 2007, doi: 10.1016/j.amc.2007.02.004.

[9] M. Antonakakis *et al.*, "From throw-away traffic to bots: detecting the rise of DGA-based malware," in *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, 2012, pp. 491–506.

[10] Wiki, "2013 South Korea Attack," *https://en.wikipedia.org/wiki/2013_South_Korea_cyberattack*, 2013.

[11] A. Dinaburg, P. Royal, M. Shari, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," 2008, doi: 10.1145/1455770.1455779.

[12] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," 2010, doi: 10.1109/SP.2010.26.

[13] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," 2007, doi: 10.1109/SP.2007.17.

[14] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation for Exploits on Commodity Software," 2005.

[15] J. Clause, W. Li, and A. Orso, “Dytan: A generic dynamic taint analysis framework,” *2007 ACM Int. Symp. Softw. Test. Anal. ISSTA’07*, 2007, doi: 10.1145/1273463.1273490.

[16] J. C. King, “Symbolic Execution and Program Testing,” *Commun. ACM*, 1976, doi: 10.1145/360248.360252.

[17] M. Hom, “Symbolic Execution Over Native X86,” NAVAL POSTGRADUATE SCHOOL MONTEREY CA, 2012.

[18] L. De Moura and N. Bjørner, “Z3: An efficient SMT Solver,” 2008, doi: 10.1007/978-3-540-78800-3_24.

[19] H. Yin and D. Song, *Automatic Malware Analysis An Emulator Based Approach*. 2013.

[20] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, “Automatically identifying trigger-based behavior in malware,” *Adv. Inf. Secur.*, 2008, doi: 10.1007/978-0-387-68768-1_4.

[21] M. G. Kang, P. Poosankam, and H. Yin, “Renovo: A hidden code extractor for packed executables,” 2007, doi: 10.1145/1314389.1314399.

[22] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, “Panorama: Capturing system-wide information flow for malware detection and analysis,” 2007, doi: 10.1145/1315245.1315261.

[23] Z. Liang, H. Yin, and D. Song, “HookFinder: Identifying and understanding malware hooking behaviors,” 2008.

[24] H. Ma, X. Ma, W. Liu, Z. Huang, D. Gao, and C. Jia, “Control flow obfuscation using neural network to fight concolic testing,” in *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST*, 2015.

[25] Z. Ding, Y. Guo, L. Zhang, and Y. Fu, “One-shot face recognition via generative learning,” 2018, doi: 10.1109/FG.2018.00011.

[26] C. Eagle, *The IDA pro book*. No Starch Press, 2011.

[27] 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 Prepr. arXiv1801.08917*, 2018.

[28] B. Kolosnjaji *et al.*, “Adversarial malware binaries: Evading deep learning for malware detection in executables,” in *2018 26th European Signal Processing Conference (EUSIPCO)*, 2018, pp. 533–537.

[29] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, “Adversarial examples for malware detection,” in *European Symposium on Research in Computer Security*, 2017, pp. 62–79.

[30] N. Papernot, P. Mcdaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, “The limitations of deep learning in adversarial settings,” 2016, doi: 10.1109/EuroSP.2016.36.

[31] W. Hu and Y. Tan, “Generating adversarial malware examples for black-box attacks based on GAN,” *arXiv Prepr. arXiv1702.05983*, 2017.

[32] B. Biggio *et al.*, “Evasion attacks against machine learning at test time,” 2013, doi: 10.1007/978-3-642-40994-3_25.

[33] P. Fogla, M. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee, “Polymorphic Blending Attacks,” *15th USENIX Secur. Symp.*, 2006.

[34] S. Park, I. Gondal, J. Kamruzzaman, and J. Oliver, “Generative malware outbreak detection,” in *Proceedings of the IEEE International Conference on Industrial Technology*, 2019, vol. 2019-Febru, pp. 1149–1154, doi: 10.1109/ICIT.2019.8754939.

[35] A. Radford, L. Metz, and S. Chintala, “Unsupervised representation learning with deep convolutional generative adversarial networks,” *arXiv Prepr. arXiv1511.06434*, 2015.

[36] I. Gondal, J. Kamruzzaman, and L. Zhang, “One-shot malware outbreak detection using spatio-temporal isomorphic dynamic features,” 2019, doi: 10.1109/TrustCom/BigDataSE.2019.00108.

[37] S. Park, I. Gondal, J. Kamruzzaman, and J. Oliver, “Instruction Cognitive One-Shot Malware Outbreak Detection,” 2019.

[38] S. Park, I. Gondal, J. Kamruzzaman, and J. Oliver, *Comparative Study on Machine Learning Methods to Detect Metamorphic Threats*, Malware An. Melbourne, Australia: Springer, 2020.

[39] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, “Impeding Malware Analysis Using Conditional Code Obfuscation,” *Proc. 15th Netw. Distrib. Syst. Secur. Symp. - NDSS ’08*, 2008.

[40] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, “PolyUnpack: Automating the hidden-code extraction of unpack-executing malware,” 2006, doi: 10.1109/ACSAC.2006.38.

[41] I. You and K. Yim, “Malware obfuscation techniques: A brief survey,” 2010,

doi: 10.1109/BWCCA.2010.85.

[42] B. Bashari Rad, M. Masrom, and S. Ibrahim, “Camouflage In Malware: From Encryption To Metamorphism,” *Int. J. Comput. Sci. Netw. Secur.*, 2012.

[43] W. Wong and M. Stamp, “Hunting for metamorphic engines,” *J. Comput. Virol.*, 2006, doi: 10.1007/s11416-006-0028-7.

[44] D. Lin and M. Stamp, “Hunting for undetectable metamorphic viruses,” *J. Comput. Virol.*, 2011, doi: 10.1007/s11416-010-0148-y.

[45] N. Runwal, R. M. Low, and M. Stamp, “Opcode graph similarity and metamorphic detection,” *J. Comput. Virol.*, 2012, doi: 10.1007/s11416-012-0160-5.

[46] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[47] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, 1986, doi: 10.1038/323533a0.

[48] L. Bottou, “Large-Scale Machine Learning with Stochastic Gradient Descent,” in *Proceedings of COMPSTAT’2010*, 2010.

[49] G. E. Hinton, S. Osindero, and Y.-W. Teh, “A fast learning algorithm for deep belief nets,” *Neural Comput.*, vol. 18, no. 7, pp. 1527–1554, 2006.

[50] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[51] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Gated feedback recurrent neural networks,” in *International Conference on Machine Learning*, 2015, pp. 2067–2075.

[52] A. Graves, “Supervised Sequence Labelling,” 2012.

[53] S. Hochreiter, “The vanishing gradient problem during learning recurrent neural nets and problem solutions,” *Int. J. Uncertainty, Fuzziness Knowledge-Based Syst.*, vol. 6, no. 02, pp. 107–116, 1998.

[54] D. Tran, H. Mac, V. Tong, H. A. Tran, and L. G. Nguyen, “A LSTM based framework for handling multiclass imbalance in DGA botnet detection,” *Neurocomputing*, 2018, doi: 10.1016/j.neucom.2017.11.018.

[55] Y. LeCun and Y. Bengio, “Convolutional networks for images, speech, and time series,” *Handb. brain theory neural networks*, vol. 3361, no. 10, p. 1995, 1995.

[56] I. J. Goodfellow *et al.*, “Generative adversarial nets,” 2014, doi: 10.3156/jsoft.29.5_177_2.

[57] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, “Improved techniques for training GANs,” 2016.

[58] A. Makhzani, J. Shlens, N. Jaitly, I. Goodfellow, and B. Frey, “Adversarial autoencoders,” *arXiv Prepr. arXiv1511.05644*, 2015.

[59] S. Nowozin, B. Cseke, and R. Tomioka, “f-GAN: Training generative neural samplers using variational divergence minimization,” 2016.

[60] C. K. Sønderby, J. Caballero, L. Theis, W. Shi, and F. Huszár, “Amortised map inference for image super-resolution,” 2019.

[61] M. Arjovsky, S. Chintala, and L. Bottou, “Wasserstein gan,” *arXiv Prepr. arXiv1701.07875*, 2017.

[62] L. Mescheder, S. Nowozin, and A. Geiger, “The numerics of GANs,” 2017.

[63] P. Baldi, “Autoencoders, Unsupervised Learning, and Deep Architectures,” *ICML Unsupervised Transf. Learn.*, 2012, doi: 10.1561/2200000006.

[64] P. Vincent, H. Larochelle, Y. Bengio, and P. A. Manzagol, “Extracting and composing robust features with denoising autoencoders,” 2008, doi: 10.1145/1390156.1390294.

[65] Y. Gal and Z. Ghahramani, “Dropout as a Bayesian approximation: Representing model uncertainty in deep learning,” 2016.

[66] J. Masci, U. Meier, D. Cireşan, and J. Schmidhuber, “Stacked convolutional auto-encoders for hierarchical feature extraction,” 2011, doi: 10.1007/978-3-642-21735-7_7.

[67] D. P. Kingma and M. Welling, “Auto-encoding variational bayes,” 2014.

[68] D. Harris and S. Harris, *Digital Design and Computer Architecture: Second Edition*. 2012.

[69] C. Guo and F. Berkhahn, “Entity embeddings of categorical variables,” *arXiv Prepr. arXiv1604.06737*, 2016.

[70] Y. Zhang, D. Shen, G. Wang, Z. Gan, R. Henao, and L. Carin, “Deconvolutional paragraph representation learning,” 2017.

[71] R. Salakhutdinov and G. Hinton, “Semantic hashing,” *Int. J. Approx. Reason.*, vol. 50, no. 7, pp. 969–978, 2009.

[72] D. J. S. Robinson, *An introduction to abstract algebra*. Walter de Gruyter, 2008.

[73] P. Isola, J. Y. Zhu, T. Zhou, and A. A. Efros, “Image-to-image translation with conditional adversarial networks,” 2017, doi: 10.1109/CVPR.2017.632.

[74] H. Zhang *et al.*, “StackGAN: Text to Photo-Realistic Image Synthesis with

Stacked Generative Adversarial Networks," 2017, doi: 10.1109/ICCV.2017.629.

[75] J. Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks," 2017, doi: 10.1109/ICCV.2017.244.

[76] C. Ledig *et al.*, "Photo-realistic single image super-resolution using a generative adversarial network," 2017, doi: 10.1109/CVPR.2017.19.

[77] D. Pathak, P. Krahenbuhl, J. Donahue, T. Darrell, and A. A. Efros, "Context Encoders: Feature Learning by Inpainting," 2016, doi: 10.1109/CVPR.2016.278.

[78] M. Y. Liu, T. Breuel, and J. Kautz, "Unsupervised image-to-image translation networks," 2017.

[79] A. Vaswani *et al.*, "Attention is all you need," 2017.

[80] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv Prepr. arXiv1810.04805*, 2018.

[81] Ł. Kaiser and I. Sutskever, "Neural GPUs learn algorithms," 2016.

[82] E. Price, W. Zaremba, and I. Sutskever, "Extensions and Limitations of the Neural GPU," *arXiv Prepr. arXiv1611.00736*, 2016.

[83] E. Gandotra, D. Bansal, and S. Sofat, "Malware analysis and classification: A survey," *J. Inf. Secur.*, vol. 5, no. 02, p. 56, 2014.

[84] Z. Bazrafshan, H. Hashemi, S. M. H. Fard, and A. Hamzeh, "A survey on heuristic malware detection techniques," 2013, doi: 10.1109/IKT.2013.6620049.

[85] F. A. Narudin, A. Feizollah, N. B. Anuar, and A. Gani, "Evaluation of machine learning classifiers for mobile malware detection," *Soft Comput.*, 2016, doi: 10.1007/s00500-014-1511-6.

[86] H. V. Nath and B. M. Mehtre, "Static Malware Analysis Using Machine Learning Methods," 2014, doi: 10.1007/978-3-642-54525-2_39.

[87] H. Rathore, S. Agarwal, S. K. Sahay, and M. Sewak, "Malware Detection Using Machine Learning and Deep Learning," in *International Conference on Big Data Analytics*, 2018, pp. 402–411.

[88] K. Raman, "Selecting Features to Classify Malware," *InfoSec Southwest 2012*, 2012.

[89] I. Santos *et al.*, "Idea: Opcode-sequence-based malware detection," 2010, doi: 10.1007/978-3-642-11747-3_3.

[90] I. Santos, F. Brezo, X. Ugarte-Pedrero, and P. G. Bringas, "Opcode sequences

as representation of executables for data-mining-based unknown malware detection," *Inf. Sci. (Ny).*, 2013, doi: 10.1016/j.ins.2011.08.020.

[91] T. M. Cover and J. A. Thomas, *Elements of Information Theory*. 2005.

[92] I. Santos, J. Nieves, and P. G. Bringas, "Semi-supervised learning for unknown malware detection," 2011, doi: 10.1007/978-3-642-19934-9_53.

[93] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic analysis of malware behavior using machine learning," *J. Comput. Secur.*, vol. 19, no. 4, pp. 639–668, 2011.

[94] S. Y. Yerima, S. Sezer, and G. McWilliams, "Analysis of Bayesian classification-based approaches for Android malware detection," *IET Inf. Secur.*, 2014, doi: 10.1049/iet-ifs.2013.0095.

[95] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane, "Graph-based malware detection using dynamic analysis," *J. Comput. Virol.*, 2011, doi: 10.1007/s11416-011-0152-x.

[96] Z. Xu, S. Ray, P. Subramanyan, and S. Malik, "Malware detection using machine learning based analysis of virtual memory access patterns," 2017, doi: 10.23919/DATE.2017.7926977.

[97] G. Jacob, H. Debar, and E. Filiol, "Behavioral detection of malware: from a survey towards an established taxonomy," *J. Comput. Virol.*, vol. 4, no. 3, pp. 251–266, 2008.

[98] R. Islam, R. Tian, L. M. Batten, and S. Versteeg, "Classification of malware based on integrated static and dynamic features," *Journal of Network and Computer Applications*. 2013, doi: 10.1016/j.jnca.2012.10.004.

[99] P. V. Shijo and A. Salim, "Integrated static and dynamic analysis for malware detection," 2015, doi: 10.1016/j.procs.2015.02.149.

[100] D. Kong and G. Yan, "Discriminant malware distance learning on structural information for automated malware classification," 2013, doi: 10.1145/2487575.2488219.

[101] J. Sahs and L. Khan, "A machine learning approach to android malware detection," 2012, doi: 10.1109/EISIC.2012.34.

[102] P. M. Comar, L. Liu, S. Saha, P. N. Tan, and A. Nucci, "Combining supervised and unsupervised learning for zero-day malware detection," 2013, doi: 10.1109/INFCOM.2013.6567003.

[103] Y. Ki, E. Kim, and H. K. Kim, "A novel approach to detect malware based on

API call sequence analysis," *Int. J. Distrib. Sens. Networks*, vol. 11, no. 6, p. 659101, 2015.

[104] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "A quantitative study of accuracy in system call-based malware detection," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 122–132.

[105] I. Firdausi, C. Lim, A. Erwin, and A. S. Nugroho, "Analysis of machine learning techniques used in behavior-based malware detection," 2010, doi: 10.1109/ACT.2010.33.

[106] P. Faruki, V. Laxmi, M. S. Gaur, and P. Vinod, "Behavioural detection with API call-grams to identify malicious PE files," 2012, doi: 10.1145/2490428.2490440.

[107] C. Guarnieri, A. Tanasi, J. Bremer, and M. Schloesser, "The cuckoo sandbox," *Accessed Dec*, vol. 16, p. 2018, 2012.

[108] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software," *ACM SIGKDD Explor. Newsl.*, 2009, doi: 10.1145/1656274.1656278.

[109] R. Tian, R. Islam, L. Batten, and S. Versteeg, "Differentiating malware from cleanware using behavioural analysis," 2010, doi: 10.1109/MALWARE.2010.5665796.

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

[111] F. Ahmed, H. Hameed, M. Z. Shafiq, and M. Farooq, "Using spatio-temporal information in API calls with machine learning algorithms for malware detection," in *Proceedings of the 2nd ACM workshop on Security and artificial intelligence*, 2009, pp. 55–62.

[112] Y. Li, R. Ma, and R. Jiao, "A hybrid malicious code detection method based on deep learning," *Int. J. Secur. Its Appl.*, vol. 9, no. 5, pp. 205–216, 2015.

[113] P. Smolensky, "Information Processing in Dynamical Systems : Foundations of Harmony Theory ; CU-CS-321-86," *Comput. Sci. Tech. Reports*, 1986.

[114] "KDD Cup 1999 Dataset," *http://kdd.ics.uci.edu/databases/ kddcup99/*. .

[115] O. E. David and N. S. Netanyahu, "DeepSign: Deep learning for automatic malware signature generation and classification," 2015, doi: 10.1109/IJCNN.2015.7280815.

[116] J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features," in *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, 2015, pp. 11–20.

[117] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu, "Large-scale malware classification using random projections and neural networks," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013, pp. 3422–3426.

[118] P. Li, T. J. Hastie, and K. W. Church, "Very sparse random projections," 2006, doi: 10.1145/1150402.1150436.

[119] A. Davis and M. Wolff, "Deep learning on disassembly data," *BlackHat USA*, 2015.

[120] S. Park, "Fighting Metamorphism using Deep Neural Network with Fourier.," 2016, [Online]. Available: https://ruxcon.org.au/assets/2016/slides/Fighting Metamorphism using Deep Learning with Fourier v1.4.pdf.

[121] B. Athiwaratkun and J. W. Stokes, "Malware classification with LSTM and GRU language models and a character-level CNN," 2017, doi: 10.1109/ICASSP.2017.7952603.

[122] W. Huang and J. W. Stokes, "MtNet: a multi-task neural network for dynamic malware classification," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2016, pp. 399–418.

[123] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, "Deep learning for classification of malware system call sequences," in *Australasian Joint Conference on Artificial Intelligence*, 2016, pp. 137–149.

[124] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing Functions in Binaries with Neural Networks," *24th USENIX Secur. Symp. (USENIX Secur. 15)*, 2015, doi: 10.1109/PPIC.2011.5982880.

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

[126] Z. Kan, H. Wang, G. Xu, Y. Guo, and X. Chen, "Towards light-weight deep learning based malware detection," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, 2018, vol. 1, pp. 600–609.

[127] H. HaddadPajouh, A. Dehghantanha, R. Khayami, and K.-K. R. Choo, "A deep

Recurrent Neural Network based approach for Internet of Things malware threat hunting," *Futur. Gener. Comput. Syst.*, vol. 85, pp. 88–96, 2018.

[128] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A Neural Probabilistic Language Model," 2003, doi: 10.1162/153244303322533223.

[129] Q. Le, O. Boydell, B. Mac Namee, and M. Scanlon, "Deep learning at the shallow end: Malware classification for non-domain experts," *Digit. Investig.*, vol. 26, pp. S118–S126, 2018.

[130] M. Kalash, M. Rochan, N. Mohammed, N. D. B. Bruce, Y. Wang, and F. Iqbal, "Malware classification with deep convolutional neural networks," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2018, pp. 1–5.

[131] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015.

[132] Y. Zhao, C. Xu, B. Bo, and Y. Feng, "MalDeep: A Deep Learning Classification Framework against Malware Variants Based on Texture Visualization," *Secur. Commun. Networks*, vol. 2019, 2019.

[133] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas, "Malware detection by eating a whole exe," 2018.

[134] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, "Learning Deep Features for Discriminative Localization," 2016, doi: 10.1109/CVPR.2016.319.

[135] E. Raff *et al.*, "An investigation of byte n-gram features for malware classification," *J. Comput. Virol. Hacking Tech.*, vol. 14, no. 1, pp. 1–20, 2018.

[136] M. Krčál, O. Švec, M. Bálek, and O. Jašek, "Deep convolutional malware classifiers can learn from raw executables and labels only," 2018.

[137] L. Chen, "Deep transfer learning for static malware classification," *arXiv Prepr. arXiv1812.07606*, 2018.

[138] J. Deng, W. Dong, R. Socher, L.-J. Li, Kai Li, and Li Fei-Fei, "ImageNet: A large-scale hierarchical image database," 2010, doi: 10.1109/cvpr.2009.5206848.

[139] M. T. Ribeiro, S. Singh, and C. Guestrin, "'Why should i trust you?' Explaining the predictions of any classifier," 2016, doi: 10.1145/2939672.2939778.

[140] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, "Distillation as a Defense to Adversarial Perturbations Against Deep Neural Networks," 2016, doi: 10.1109/SP.2016.41.

[141] C. Szegedy *et al.*, "Intriguing properties of neural networks," 2014.

[142] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "Avclass: A tool for massive malware labeling," in *International Symposium on Research in Attacks, Intrusions, and Defenses*, 2016, pp. 230–253.

[143] A. Shalaginov, S. Banin, A. Dehghantanha, and K. Franke, "Machine learning aided static malware analysis: A survey and tutorial," in *Cyber Threat Intelligence*, Springer, 2018, pp. 7–45.

[144] U. Bayer *et al.*, "On challenges in evaluating malware clustering," in *International Workshop on Recent Advances in Intrusion Detection*, 2010, vol. 9, pp. 238–255.

[145] X. Hu, K. G. Shin, S. Bhatkar, and K. Griffin, "Mutantx-s: Scalable malware clustering based on static features," in *Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, 2013, pp. 187–198.

[146] S. Pai, F. Di Troia, C. A. Visaggio, T. H. Austin, and M. Stamp, "Clustering for malware classification," *J. Comput. Virol. Hacking Tech.*, vol. 13, no. 2, pp. 95–107, 2017.

[147] N. Dilokthanakul *et al.*, "Deep unsupervised clustering with gaussian mixture variational autoencoders," *arXiv Prepr. arXiv1611.02648*, 2016.

[148] R. Zak, E. Raff, and C. Nicholas, "What can N-grams learn for malware detection?," in *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*, 2017, pp. 109–118.

[149] I. Santos, Y. K. Penya, J. Devesa, and P. G. Bringas, "N-grams-based file signatures for malware detection," 2009, doi: 10.5220/0001863603170320.

[150] A. Pektaş, M. Eriş, and T. Acarman, "Proposal of n-gram based algorithm for malware classification," 2011.

[151] S. Jain and Y. K. Meena, "Byte level n-Gram analysis for malware detection," 2011, doi: 10.1007/978-3-642-22786-8_6.

[152] Wiki, "Histogram Matching," *https://en.wikipedia.org/wiki/Histogram_matching*. .

[153] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, 2007, pp. 421–430.

[154] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," 2015.

[155] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA Neural networks Mach.*

Learn., vol. 4, no. 2, pp. 26–31, 2012.

[156] L. McInnes and J. Healy, "Accelerated Hierarchical Density Based Clustering," 2017, doi: 10.1109/ICDMW.2017.12.

[157] "VirusTotal," *https://www.virustotal.com/.* .

[158] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville, "Improved training of wasserstein gans," in *Advances in neural information processing systems*, 2017, pp. 5767–5777.

[159] J. R. Youngblood, "Ransomware," in *Business Theft and Fraud*, 2016.

[160] W. Wright, D. Schroh, P. Proulx, A. Skaburskis, and B. Cort, "The sandbox for analysis - Concepts and methods," 2006.

[161] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," 2015.

[162] "Sensitivity and specificity," *https://en.wikipedia.org/wiki/Sensitivity_and_specificity.* .

[163] R. Vinayakumar and K. P. Soman, "DeepMalNet: Evaluating shallow and deep networks for static PE malware detection," *ICT Express*, 2018, doi: 10.1016/j.icte.2018.10.006.

[164] J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Math. Comput.*, 1965, doi: 10.2307/2003354.

[165] "Pearson Correlation Coefficient," *https://en.wikipedia.org/wiki/Pearson_correlation_coefficient, last accessed 2019/06/21.* .

[166] Y. Ye, T. Li, D. Adjeroh, and S. S. Iyengar, "A survey on malware detection using data mining techniques," *ACM Comput. Surv.*, vol. 50, no. 3, p. 41, 2017.

[167] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *Am. Stat.*, vol. 46, no. 3, pp. 175–185, 1992.

[168] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[169] S. M. Omohundro, *Five balltree construction algorithms*. International Computer Science Institute Berkeley, 1989.

[170] Wiki, "Minkowski Distance," *https://en.wikipedia.org/wiki/Minkowski_distance.* .

[171] L. McInnes, J. Healy, and S. Astels, "hdbscan: Hierarchical density based clustering.," *J. Open Source Softw.*, vol. 2, no. 11, p. 205, 2017.

[172] V. Roussev, “Data fingerprinting with similarity digests,” in *IFIP International Conference on Digital Forensics*, 2010, pp. 207–226.

[173] J. Kornblum, “Identifying almost identical files using context triggered piecewise hashing,” *Digit. Investig.*, vol. 3, pp. 91–97, 2006.

[174] J. Oliver, C. Cheng, and Y. Chen, “TLSH--a locality sensitive hash,” in *2013 Fourth Cybercrime and Trustworthy Computing Workshop*, 2013, pp. 7–13.

[175] V. Roussev, “An evaluation of forensic similarity hashes,” *Digit. Investig.*, vol. 8, pp. S34–S41, 2011.

[176] F. Pagani, M. Dell’Amico, and D. Balzarotti, “Beyond precision and recall: understanding uses (and misuses) of similarity hashes in binary analysis,” in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, 2018, pp. 354–365.

[177] N. A. 24. Quynh, “Capstone: Next-gen disassembly framework,” 2014.

[178] D. Hendler, S. Kels, and A. Rubin, “Detecting malicious powershell commands using deep neural networks,” 2018, doi: 10.1145/3196494.3196511.

Appendix

Generative Static Malware Detection: Detailed Results

Generative static malware detection presented in chapter 3 has been evaluated by executing a python script developed using tensorflow, generative-static-detection.py.

Evaluation on benign test set

```
python generative-static-detection.py eval --name test. benign --threshold 10
Evaluation result for 9981 samples saved to eval.csv
test.benign accuracy: 0.659954 (6587/9981)
python generative-static-detection.py eval --name test.benign --threshold 2
Evaluation result for 9981 samples saved to eval.csv
test.benign accuracy: 0.912734 (9110/9981)
```

Evaluation on malicious test set

```
python generative-static-detection.py eval --name test.malicious --threshold 5
Evaluation result for 3254 samples saved to eval.csv
test.malicious accuracy: 0.977873 (3182/3254)
python generative-static-detection.py eval --name test.malicious --threshold 2
Evaluation result for 3254 samples saved to eval.csv
test.malicious accuracy: 0.957591 (3116/3254)
python generative-static-detection.py eval --name test.malicious --threshold 1
Evaluation result for 3254 samples saved to eval.csv
test.malicious accuracy: 0.948064 (3085/3254)
```

Baseline evaluation was conducted using a python script developed with scikit-learn. The execution log is shown below.

```
trainset=175 testset=3254 batchsize=10
1-gram: total 453 features extracted
        top 10 freqs: [2752, 2445, 2393, 2387, 2074, 1472, 1441, 1085, 901, 870]
        number of 1's: 77
2-gram: total 9449 features extracted
        top 10 freqs: [2386, 2049, 1814, 1574, 1525, 1344, 1337, 1224, 1172, 1166]
        number of 1's: 3900
3-gram: total 54539 features extracted
        top 10 freqs: [6237, 2677, 2361, 2104, 1916, 1754, 1527, 1349, 1209, 1200]
        number of 1's: 27065
>osx-base-outbreak|osx-base-malicious|gradientbooster|1gram: 0.935 (3044/3254)
>osx-base-outbreak|osx-base-malicious|gradientbooster|2gram: 0.936 (3045/3254)
>osx-base-outbreak|osx-base-malicious|gradientbooster|3gram: 0.931 (3028/3254)
>osx-base-outbreak|osx-base-malicious|svm|1gram: 0.934 (3038/3254)
>osx-base-outbreak|osx-base-malicious|svm|2gram: 0.944 (3071/3254)
>osx-base-outbreak|osx-base-malicious|svm|3gram: 0.968 (3151/3254)
>osx-base-outbreak|osx-base-malicious|randomforest|1gram: 0.983 (3200/3254)
>osx-base-outbreak|osx-base-malicious|randomforest|2gram: 0.987 (3213/3254)
```

```
>osx-base-outbreak|osx-base-malicious|randomforest|3gram: 0.989 (3217/3254)

1-gram: total 453 features extracted
        top 10 freqs: [2752, 2445, 2393, 2387, 2074, 1472, 1441, 1085, 901, 870]
        number of 1's: 77
2-gram: total 9449 features extracted
        top 10 freqs: [2386, 2049, 1814, 1574, 1525, 1344, 1337, 1224, 1172, 1166]
        number of 1's: 3900
3-gram: total 54539 features extracted
        top 10 freqs: [6237, 2677, 2361, 2104, 1916, 1754, 1527, 1349, 1209, 1200]
        number of 1's: 27065

train.x.npz!train.y.npz|test.benign.x.npz!test.benign.y.npz|gradientbooster|1gram
>train.x.npz!train.y.npz|test.benign.x.npz!test.benign.y.npz|gradientbooster|1gram: 0.000 (0/7857)
train.x.npz!train.y.npz|test.benign.x.npz!test.benign.y.npz|gradientbooster|2gram
>train.x.npz!train.y.npz|test.benign.x.npz!test.benign.y.npz|gradientbooster|2gram: 0.000 (0/7857)
train.x.npz!train.y.npz|test.benign.x.npz!test.benign.y.npz|gradientbooster|3gram
>train.x.npz!train.y.npz|test.benign.x.npz!test.benign.y.npz|gradientbooster|3gram: 0.000 (0/7857)
train.x.npz!train.y.npz|test.benign.x.npz!test.benign.y.npz|svm|1gram
>train.x.npz!train.y.npz|test.benign.x.npz!test.benign.y.npz|svm|1gram: 0.000 (0/7857)
train.x.npz!train.y.npz|test.benign.x.npz!test.benign.y.npz|svm|2gram
>train.x.npz!train.y.npz|test.benign.x.npz!test.benign.y.npz|svm|2gram: 0.000 (0/7857)
train.x.npz!train.y.npz|test.benign.x.npz!test.benign.y.npz|svm|3gram
>train.x.npz!train.y.npz|test.benign.x.npz!test.benign.y.npz|svm|3gram: 0.000 (0/7857)
train.x.npz!train.y.npz|test.benign.x.npz!test.benign.y.npz|randomforest|1gram
>train.x.npz!train.y.npz|test.benign.x.npz!test.benign.y.npz|randomforest|1gram: 0.000 (0/7857)
train.x.npz!train.y.npz|test.benign.x.npz!test.benign.y.npz|randomforest|2gram
>train.x.npz!train.y.npz|test.benign.x.npz!test.benign.y.npz|randomforest|2gram: 0.000 (0/7857)
train.x.npz!train.y.npz|test.benign.x.npz!test.benign.y.npz|randomforest|3gram
>train.x.npz!train.y.npz|test.benign.x.npz!test.benign.y.npz|randomforest|3gram: 0.000 (0/7857)
train.x.npz!train.y.npz|test.malicious.x.npz!test.malicious.y.npz|gradientbooster|1gram
>train.x.npz!train.y.npz|test.malicious.x.npz!test.malicious.y.npz|gradientbooster|1gram:    0.956
(3112/3254)
train.x.npz!train.y.npz|test.malicious.x.npz!test.malicious.y.npz|gradientbooster|2gram
>train.x.npz!train.y.npz|test.malicious.x.npz!test.malicious.y.npz|gradientbooster|2gram:    0.938
(3051/3254)
train.x.npz!train.y.npz|test.malicious.x.npz!test.malicious.y.npz|gradientbooster|3gram
>train.x.npz!train.y.npz|test.malicious.x.npz!test.malicious.y.npz|gradientbooster|3gram:    0.934
(3038/3254)
train.x.npz!train.y.npz|test.malicious.x.npz!test.malicious.y.npz|svm|1gram
>train.x.npz!train.y.npz|test.malicious.x.npz!test.malicious.y.npz|svm|1gram: 0.934 (3038/3254)
train.x.npz!train.y.npz|test.malicious.x.npz!test.malicious.y.npz|svm|2gram
>train.x.npz!train.y.npz|test.malicious.x.npz!test.malicious.y.npz|svm|2gram: 0.944 (3071/3254)
train.x.npz!train.y.npz|test.malicious.x.npz!test.malicious.y.npz|svm|3gram
>train.x.npz!train.y.npz|test.malicious.x.npz!test.malicious.y.npz|svm|3gram: 0.968 (3151/3254)
train.x.npz!train.y.npz|test.malicious.x.npz!test.malicious.y.npz|randomforest|1gram
>train.x.npz!train.y.npz|test.malicious.x.npz!test.malicious.y.npz|randomforest|1gram:       0.983
(3200/3254)
train.x.npz!train.y.npz|test.malicious.x.npz!test.malicious.y.npz|randomforest|2gram
>train.x.npz!train.y.npz|test.malicious.x.npz!test.malicious.y.npz|randomforest|2gram:       0.987
(3213/3254)
train.x.npz!train.y.npz|test.malicious.x.npz!test.malicious.y.npz|randomforest|3gram
>train.x.npz!train.y.npz|test.malicious.x.npz!test.malicious.y.npz|randomforest|3gram:       0.989
(3217/3254)
```

Generative Dynamic Malware Detection: Detailed Results

Generative dynamic malware detection presented in chapter 4 has been evaluated by executing a python script developed using tensorflow, generative-dynamic-detection.py.

```
python generative-dynamic-detection.py predict --sampledir program-datasets --datasetname osx-
dynamic-malware --metric mse --threshold 0.000025
Total detection: 99.159% (2831/2855)
python generative-dynamic-detection.py predict -- sampledir program-datasets --datasetname osx-
dynamic-benign --metric mse --threshold 0.000025
Total detection: 0.106% (8/7541)
```

The result of baseline models trained with no benign training samples is shown below.

```
python classifier_benchmark.py
trainset=(train.x.npz,train.y.npz)
testset=(test.benign.x.npz,test.benign.y.npz) batchsize=10
train.x.npz,train.y.npz|test.benign.x.npz,test.benign.y.npz|gradientbooster|1gram
1-gram: total 103 features extracted
        top 10 freqs: [2527, 1948, 1736, 1633, 1578, 1402, 1246, 1022, 954, 933]
        number of single appearance ngrams: 11
train.x.npz,train.y.npz|test.benign.x.npz,test.benign.y.npz|gradientbooster|2gram
2-gram: total 5123 features extracted
        top 10 freqs: [723, 551, 545, 499, 498, 497, 497, 497, 497, 497]
        number of single appearance ngrams: 3178
>train.x.npz,train.y.npz|test.benign.x.npz,test.benign.y.npz|gradientbooster|1gram: 0.000 (0/7541)
>train.x.npz,train.y.npz|test.benign.x.npz,test.benign.y.npz|gradientbooster|2gram: 0.000 (0/7541)
>train.x.npz,train.y.npz|test.benign.x.npz,test.benign.y.npz|svm|1gram: 0.000 (0/7541)
>train.x.npz,train.y.npz|test.benign.x.npz,test.benign.y.npz|svm|2gram: 0.000 (0/7541)
>train.x.npz,train.y.npz|test.benign.x.npz,test.benign.y.npz|randomforest|1gram: 0.000 (0/7541)
>train.x.npz,train.y.npz|test.benign.x.npz,test.benign.y.npz|randomforest|2gram: 0.000 (0/7541)
>train.x.npz,train.y.npz|test.malicious.x.npz,test.malicious.y.npz|gradientbooster|1gram:    0.167
(478/2855)
>train.x.npz,train.y.npz|test.malicious.x.npz,test.malicious.y.npz|gradientbooster|2gram:    0.126
(361/2855)
>train.x.npz,train.y.npz|test.malicious.x.npz,test.malicious.y.npz|svm|1gram: 0.367 (1047/2855)
>train.x.npz,train.y.npz|test.malicious.x.npz,test.malicious.y.npz|svm|2gram: 0.394 (1125/2855)
>train.x.npz,train.y.npz|test.malicious.x.npz,test.malicious.y.npz|randomforest|1gram:       0.452
(1291/2855)
>train.x.npz,train.y.npz|test.malicious.x.npz,test.malicious.y.npz|randomforest|2gram:       0.356
(1017/2855)
```

The result of baseline models trained with 50% of benign samples in the training set is shown below.

```
python classifier_benchmark.py
loading data(train.x.npz,train.y.npz)
loading data(test.benign.x.npz,test.benign.y.npz)
loading data(test.malicious.x.npz,test.malicious.y.npz)
3770 benign test samples addded to train set
train=4113 test.benign=3771 test.malicious=2855
1-gram: total 104 features extracted
        top 10 freqs: [1075, 869, 761, 585, 569, 556, 523, 447, 398, 350]
        number of single appearance ngrams: 19
2-gram: total 5690 features extracted
        top 10 freqs: [425, 369, 296, 296, 296, 296, 296, 296, 296, 296]
        number of single appearance ngrams: 3675
>gradientbooster|train.x.npz|test.benign.x.npz|1gram: 0.974 (3674/3771)
>gradientbooster|train.x.npz|test.benign.x.npz|2gram: 0.973 (3671/3771)
>svm|train.x.npz|test.benign.x.npz|1gram: 1.000 (3771/3771)
>svm|train.x.npz|test.benign.x.npz|2gram: 1.000 (3771/3771)
>randomforest|train.x.npz|test.benign.x.npz|1gram: 1.000 (3771/3771)
>randomforest|train.x.npz|test.benign.x.npz|2gram: 1.000 (3771/3771)
>gradientbooster|train.x.npz|test.malicious.x.npz|1gram: 0.194 (555/2855)
>gradientbooster|train.x.npz|test.malicious.x.npz|2gram: 0.108 (308/2855)
>svm|train.x.npz|test.malicious.x.npz|1gram: 0.056 (161/2855)
>svm|train.x.npz|test.malicious.x.npz|2gram: 0.056 (161/2855)
>randomforest|train.x.npz|test.malicious.x.npz|1gram: 0.460 (1314/2855)
>randomforest|train.x.npz|test.malicious.x.npz|2gram: 0.385 (1099/2855)
```

The result for varying threshold values is shown below.

Threshold	TP	FP
0.000500	2855/2855	2315/7541
0.000100	2855/2855	433/7541
0.000075	2855/2855	246/7541
0.000050	2855/2855	83/7541
0.000025	**2831/2855**	**8/7541**
0.000010	2456/2855	0/7541

Instruction Cognitive Malware Detection: Detailed Results

Instruction cognitive static detection presented in chapter 5 has been evaluated by executing a python script. 20% of total benign samples were used for training. Training and test sets are mutually exclusive.

```
Training set size=488: 3085 campaign samples
threshold=0.110000 tp=3078/3085 (99.773%) fp=130/6826 (1.904%)

train benign: 6826-(2730+2731) = 1365 (20%)
```

Baseline evaluation was conducted using a python script developed with scikit-learn. The execution log is shown below.

```
loading data(train.x.npz,train.y.npz)
loading data(test.benign1.x.npz,test.benign1.y.npz)
loading data(test.benign2.x.npz,test.benign2.y.npz)
loading data(test.malicious.x.npz,test.malicious.y.npz)
1-gram: total 256 features extracted
        top 10 freqs: [3600, 2227, 925, 739, 701, 683, 561, 490, 436, 432]
        number of single occurrence ngrams: 0
>gradientbooster|train.x.npz|test.benign1.x.npz|1gram: 0.874 (2387/2730)
>svm|train.x.npz|test.benign1.x.npz|1gram: 0.998 (2724/2730)
>randomforest|train.x.npz|test.benign1.x.npz|1gram: 0.856 (2336/2730)

>gradientbooster|train.x.npz|test.benign2.x.npz|1gram: 0.822 (2245/2731)
>svm|train.x.npz|test.benign2.x.npz|1gram: 0.999 (2728/2731)
>randomforest|train.x.npz|test.benign2.x.npz|1gram: 0.815 (2225/2731)

>gradientbooster|train.x.npz|test.malicious.x.npz|1gram: 0.967 (2982/3085)
>svm|train.x.npz|test.malicious.x.npz|1gram: 0.656 (2023/3085)
>randomforest|train.x.npz|test.malicious.x.npz|1gram: 0.981 (3025/3085)

2-gram: total 65536 features extracted
        top 10 freqs: [2948, 1668, 1370, 1248, 1131, 927, 826, 669, 556, 549]
        number of single occurrence ngrams: 0
>gradientbooster|train.x.npz|test.benign1.x.npz|2gram: 0.942 (2572/2730)
>svm|train.x.npz|test.benign1.x.npz|2gram: 0.997 (2722/2730)
>randomforest|train.x.npz|test.benign1.x.npz|2gram: 0.919 (2509/2730)

>gradientbooster|train.x.npz|test.benign2.x.npz|2gram: 0.866 (2366/2731)
>svm|train.x.npz|test.benign2.x.npz|2gram: 0.999 (2728/2731)
>randomforest|train.x.npz|test.benign2.x.npz|2gram: 0.847 (2314/2731)

>gradientbooster|train.x.npz|test.malicious.x.npz|2gram: 0.986 (3043/3085)
>svm|train.x.npz|test.malicious.x.npz|2gram: 0.853 (2630/3085)
>randomforest|train.x.npz|test.malicious.x.npz|2gram: 0.982 (3030/3085)
```

The TP (True Positive) and FP (False Positive) are calculated as follows.

1-gram

```
gradientbooster : 1 - (2387+2245)/(2730+2731) = 0.1518
svm : 1 - (2724+2728)/(2730+2731) = 0.0016
randomforest : 1 - (2336+2225)/(2730+2731) = 0.1648
```

2-gram

```
gradientbooster : 1 - (2572+2366)/(2730+2731) = 0.0957
svm : 1 - (2722+2728)/(2728+2731) = 0.0016
randomforest : 1 - (2509+2314)/(2730+2731) = 0.1168
```

FP complexity analysis has been conducted as follows by varying the number of training samples indicated by *train percent*, followed by a table showing the final result.

train percent=**0.050000**

```
>gradientbooster|train.x.npz|test.benign1.x.npz|1gram: 0.456 (1478/3242)
>svm|train.x.npz|test.benign1.x.npz|1gram: 0.102 (332/3242)
>randomforest|train.x.npz|test.benign1.x.npz|1gram: 0.418 (1355/3242)
>gradientbooster|train.x.npz|test.benign2.x.npz|1gram: 0.451 (1463/3243)
>svm|train.x.npz|test.benign2.x.npz|1gram: 0.111 (361/3243)
>randomforest|train.x.npz|test.benign2.x.npz|1gram: 0.414 (1342/3243)
gradientbooster: 1. - (1478+1463)/(3242+3243) = 0.5464
svm: 1. - (332+361)/(3242+3243) = 0.8931
randomforest: 1. - (1355+1342)/(3242+3243) = 0.5841
```

train percent=**0.100000**

```
>gradientbooster|train.x.npz|test.benign1.x.npz|1gram: 0.665 (2044/3072)
>svm|train.x.npz|test.benign1.x.npz|1gram: 0.993 (3052/3072)
>randomforest|train.x.npz|test.benign1.x.npz|1gram: 0.635 (1951/3072)
>gradientbooster|train.x.npz|test.benign2.x.npz|1gram: 0.609 (1870/3072)
>svm|train.x.npz|test.benign2.x.npz|1gram: 0.998 (3065/3072)
>randomforest|train.x.npz|test.benign2.x.npz|1gram: 0.599 (1841/3072)
gradientbooster: 1. - (2044+1870)/(3072+3072) = 0.3629
svm: 1. - (3052+1870)/( 3065+3072) = 0.1979
randomforest: 1. - (1951+1841)/(3072+3072) = 0.3828
```

train percent=**0.200000**

```
>gradientbooster|train.x.npz|test.benign1.x.npz|1gram: 0.874 (2387/2730)
>svm|train.x.npz|test.benign1.x.npz|1gram: 0.998 (2724/2730)
>randomforest|train.x.npz|test.benign1.x.npz|1gram: 0.856 (2336/2730)
>gradientbooster|train.x.npz|test.benign2.x.npz|1gram: 0.822 (2245/2731)
>svm|train.x.npz|test.benign2.x.npz|1gram: 0.999 (2728/2731)
>randomforest|train.x.npz|test.benign2.x.npz|1gram: 0.815 (2225/2731)
>gradientbooster|train.x.npz|test.malicious.x.npz|1gram: 0.967 (2982/3085)
>svm|train.x.npz|test.malicious.x.npz|1gram: 0.656 (2023/3085)
>randomforest|train.x.npz|test.malicious.x.npz|1gram: 0.981 (3025/3085)
gradientbooster : 1 - (2387+2245)/(2730+2731) = 0.15180
svm : 1 - (2724+2728)/(2730+2731) = 0.00164
randomforest : 1 - (2336+2225)/(2730+2731) = 0.16480
```

Model	FP with different benign train set			
	5%	10%	20%	50%
gradientbooster-unigram	54.649	36.295	15.180	0.005
svm-unigram	89.313	19.797	0.164	0.000
randomforest-unigram	58.411	38.281	16.480	0.005

Comparative Study on Metamorphic Threats: Detailed Results

Comparative study presented in chapter 6 has been evaluated using a custom python script developed with scikit-learn and tensorflow.

Bi-gram for the relevant baseline models

```
2-gram: total 4457 features extracted
        top 10 freqs: [3166, 683, 682, 390, 387, 368, 363, 361, 360, 358]
        number of single occurrence ngrams: 0
```

Evaluation in Strict mode

```
Loaded tokens from: model-eval/tokens.pkl
233 tokens in model_eval
>>sdhash
threshold=0.010: precision=1.000000 recall=0.172676 f1=0.294498 (tp=91 tn=5854 fp=0 fn=436)
threshold=0.050: precision=1.000000 recall=0.195446 f1=0.326984 (tp=103 tn=5854 fp=0 fn=424)
threshold=0.100: precision=1.000000 recall=0.229602 f1=0.373457 (tp=121 tn=5854 fp=0 fn=406)
threshold=0.200: precision=1.000000 recall=0.284630 f1=0.443131 (tp=150 tn=5854 fp=0 fn=377)
threshold=0.300: precision=0.977654 recall=0.332068 f1=0.495751 (tp=175 tn=5850 fp=4 fn=352)
threshold=0.400: precision=0.784553 recall=0.366224 f1=0.499353 (tp=193 tn=5801 fp=53 fn=334)
threshold=0.500: precision=0.546174 recall=0.392789 f1=0.456954 (tp=207 tn=5682 fp=172 fn=320)
threshold=0.700: precision=0.468172 recall=0.432638 f1=0.449704 (tp=228 tn=5595 fp=259 fn=299)
threshold=0.900: precision=0.122460 recall=0.445920 f1=0.192150 (tp=235 tn=4170 fp=1684 fn=292)
>>knn
threshold=1.000: precision=0.986547 recall=0.417457 f1=0.586667 (tp=220 tn=5851 fp=3 fn=307)
threshold=5.000: precision=0.959885 recall=0.635674 f1=0.764840 (tp=335 tn=5840 fp=14 fn=192)
threshold=10.000: precision=0.945498 recall=0.757116 f1=0.840885 (tp=399 tn=5831 fp=23 fn=128)
threshold=20.000: precision=0.866530 recall=0.800759 f1=0.832347 (tp=422 tn=5789 fp=65 fn=105)
threshold=30.000: precision=0.825919 recall=0.810247 f1=0.818008 (tp=427 tn=5764 fp=90 fn=100)
threshold=40.000: precision=0.792208 recall=0.810247 f1=0.801126 (tp=427 tn=5742 fp=112 fn=100)
threshold=50.000: precision=0.738832 recall=0.815939 f1=0.775473 (tp=430 tn=5702 fp=152 fn=97)
threshold=80.000: precision=0.699839 recall=0.827324 f1=0.758261 (tp=436 tn=5667 fp=187 fn=91)
threshold=100.000: precision=0.672308 recall=0.829222 f1=0.742566 (tp=437 tn=5641 fp=213 fn=90)
threshold=200.000: precision=0.607438 recall=0.836812 f1=0.703911 (tp=441 tn=5569 fp=285 fn=86)
threshold=300.000: precision=0.488474 recall=0.844402 f1=0.618915 (tp=445 tn=5388 fp=466 fn=82)
>>hdbscan
threshold=0.000: precision=0.031111 recall=0.013283 f1=0.018617 (tp=7 tn=5636 fp=218 fn=520)
threshold=0.000: precision=0.064286 recall=0.034156 f1=0.044610 (tp=18 tn=5592 fp=262 fn=509)
threshold=0.000: precision=0.055901 recall=0.017078 f1=0.026163 (tp=9 tn=5702 fp=152 fn=518)
threshold=0.000: precision=0.075630 recall=0.017078 f1=0.027864 (tp=9 tn=5744 fp=110 fn=518)
threshold=0.000: precision=0.000000 recall=0.000000 f1=0.000000 (tp=0 tn=5825 fp=29 fn=527)
threshold=0.000: precision=0.000000 recall=0.000000 f1=0.000000 (tp=0 tn=5780 fp=74 fn=527)
threshold=0.000: precision=0.000000 recall=0.000000 f1=0.000000 (tp=0 tn=5781 fp=73 fn=527)
threshold=0.000: precision=0.000000 recall=0.000000 f1=0.000000 (tp=0 tn=5784 fp=70 fn=527)
threshold=0.000: precision=0.000000 recall=0.000000 f1=0.000000 (tp=0 tn=5784 fp=70 fn=527)
threshold=0.000: precision=0.000000 recall=0.000000 f1=0.000000 (tp=0 tn=5784 fp=70 fn=527)
threshold=0.000: precision=0.000000 recall=0.000000 f1=0.000000 (tp=0 tn=5784 fp=70 fn=527)
threshold=0.000: precision=0.000000 recall=0.000000 f1=0.000000 (tp=0 tn=5785 fp=69 fn=527)
threshold=0.000: precision=0.000000 recall=0.000000 f1=0.000000 (tp=0 tn=5785 fp=69 fn=527)
threshold=0.000: precision=0.000000 recall=0.000000 f1=0.000000 (tp=0 tn=5785 fp=69 fn=527)
```

```
threshold=0.000: precision=0.000000 recall=0.000000 f1=0.000000 (tp=0 tn=5786 fp=68 fn=527)
>>aae
Ground Truth: tp=527 tn=5854
threshold=0.010: precision=0.997636 recall=0.800759 f1=0.888421 (tp=422 tn=5853 fp=1 fn=105)
threshold=0.020: precision=0.930754 recall=0.867173 f1=0.897839 (tp=457 tn=5820 fp=34 fn=70)
threshold=0.025: precision=0.791379 recall=0.870968 f1=0.829268 (tp=459 tn=5733 fp=121 fn=68)
threshold=0.030: precision=0.365142 recall=0.878558 f1=0.515877 (tp=463 tn=5049 fp=805 fn=64)
threshold=0.040: precision=0.135965 recall=0.882353 f1=0.235622 (tp=465 tn=2899 fp=2955 fn=62)
threshold=0.050: precision=0.100432 recall=0.882353 f1=0.180337 (tp=465 tn=1689 fp=4165 fn=62)
threshold=0.080: precision=0.078827 recall=0.882353 f1=0.144725 (tp=465 tn=420 fp=5434 fn=62)
threshold=0.100: precision=0.075548 recall=0.882353 f1=0.139180 (tp=465 tn=164 fp=5690 fn=62)
threshold=0.150: precision=0.074376 recall=0.882353 f1=0.137188 (tp=465 tn=67 fp=5787 fn=62)
threshold=0.200: precision=0.074151 recall=0.882353 f1=0.136805 (tp=465 tn=48 fp=5806 fn=62)
```

Evaluation in Relaxed mode

```
Loaded tokens from: model-eval/tokens.pkl
233 tokens in model_eval
>>sdhash
threshold=0.010: precision=1.000000 recall=0.178368 f1=0.302738 (tp=94 tn=5854 fp=0 fn=433)
threshold=0.050: precision=1.000000 recall=0.201139 f1=0.334913 (tp=106 tn=5854 fp=0 fn=421)
threshold=0.100: precision=1.000000 recall=0.240987 f1=0.388379 (tp=127 tn=5854 fp=0 fn=400)
threshold=0.200: precision=1.000000 recall=0.296015 f1=0.456808 (tp=156 tn=5854 fp=0 fn=371)
threshold=0.300: precision=0.978610 recall=0.347249 f1=0.512605 (tp=183 tn=5850 fp=4 fn=344)
threshold=0.400: precision=0.792157 recall=0.383302 f1=0.516624 (tp=202 tn=5801 fp=53 fn=325)
threshold=0.500: precision=0.573201 recall=0.438330 f1=0.496774 (tp=231 tn=5682 fp=172 fn=296)
threshold=0.700: precision=0.504780 recall=0.500949 f1=0.502857 (tp=264 tn=5595 fp=259 fn=263)
threshold=0.900: precision=0.140816 recall=0.523719 f1=0.221954 (tp=276 tn=4170 fp=1684 fn=251)
>>knn
threshold=1.000: precision=0.986726 recall=0.423150 f1=0.592297 (tp=223 tn=5851 fp=3 fn=304)
threshold=5.000: precision=0.961749 recall=0.667932 f1=0.788354 (tp=352 tn=5840 fp=14 fn=175)
threshold=10.000: precision=0.948775 recall=0.808349 f1=0.872951 (tp=426 tn=5831 fp=23 fn=101)
threshold=20.000: precision=0.875954 recall=0.870968 f1=0.873454 (tp=459 tn=5789 fp=65 fn=68)
threshold=30.000: precision=0.839286 recall=0.891841 f1=0.864765 (tp=470 tn=5764 fp=90 fn=57)
threshold=40.000: precision=0.807890 recall=0.893738 f1=0.848649 (tp=471 tn=5742 fp=112 fn=56)
threshold=50.000: precision=0.758347 recall=0.905123 f1=0.825260 (tp=477 tn=5702 fp=152 fn=50)
threshold=80.000: precision=0.720896 recall=0.916509 f1=0.807018 (tp=483 tn=5667 fp=187 fn=44)
threshold=100.000: precision=0.694405 recall=0.918406 f1=0.790850 (tp=484 tn=5641 fp=213 fn=43)
threshold=200.000: precision=0.632732 recall=0.931689 f1=0.753645 (tp=491 tn=5569 fp=285 fn=36)
threshold=300.000: precision=0.515593 recall=0.941176 f1=0.666219 (tp=496 tn=5388 fp=466 fn=31)
>>hdbscan
threshold=0.000: precision=0.495370 recall=0.406072 f1=0.446298 (tp=214 tn=5636 fp=218 fn=313)
threshold=0.000: precision=0.498084 recall=0.493359 f1=0.495710 (tp=260 tn=5592 fp=262 fn=267)
threshold=0.000: precision=0.628362 recall=0.487666 f1=0.549145 (tp=257 tn=5702 fp=152 fn=270)
threshold=0.000: precision=0.711286 recall=0.514231 f1=0.596916 (tp=271 tn=5744 fp=110 fn=256)
threshold=0.000: precision=0.887160 recall=0.432638 f1=0.581633 (tp=228 tn=5825 fp=29 fn=299)
threshold=0.000: precision=0.796143 recall=0.548387 f1=0.649438 (tp=289 tn=5780 fp=74 fn=238)
threshold=0.000: precision=0.794944 recall=0.537002 f1=0.640997 (tp=283 tn=5781 fp=73 fn=244)
threshold=0.000: precision=0.792899 recall=0.508539 f1=0.619653 (tp=268 tn=5784 fp=70 fn=259)
threshold=0.000: precision=0.777778 recall=0.464896 f1=0.581948 (tp=245 tn=5784 fp=70 fn=282)
threshold=0.000: precision=0.776358 recall=0.461101 f1=0.578571 (tp=243 tn=5784 fp=70 fn=284)
threshold=0.000: precision=0.769737 recall=0.444023 f1=0.563177 (tp=234 tn=5784 fp=70 fn=293)
threshold=0.000: precision=0.771523 recall=0.442125 f1=0.562123 (tp=233 tn=5785 fp=69 fn=294)
threshold=0.000: precision=0.770000 recall=0.438330 f1=0.558646 (tp=231 tn=5785 fp=69 fn=296)
threshold=0.000: precision=0.769231 recall=0.436433 f1=0.556901 (tp=230 tn=5785 fp=69 fn=297)
```

```
threshold=0.000: precision=0.767123 recall=0.425047 f1=0.547009 (tp=224 tn=5786 fp=68 fn=303)
>>aae
Ground Truth: tp=527 tn=5854
threshold=0.010: precision=0.997773 recall=0.850095 f1=0.918033 (tp=448 tn=5853 fp=1 fn=79)
threshold=0.020: precision=0.936330 recall=0.948767 f1=0.942507 (tp=500 tn=5820 fp=34 fn=27)
threshold=0.025: precision=0.808544 recall=0.969639 f1=0.881795 (tp=511 tn=5733 fp=121 fn=16)
threshold=0.030: precision=0.392911 recall=0.988615 f1=0.562331 (tp=521 tn=5049 fp=805 fn=6)
threshold=0.040: precision=0.151106 recall=0.998102 f1=0.262475 (tp=526 tn=2899 fp=2955 fn=1)
threshold=0.050: precision=0.112319 recall=1.000000 f1=0.201954 (tp=527 tn=1689 fp=4165 fn=0)
threshold=0.080: precision=0.088408 recall=1.000000 f1=0.162454 (tp=527 tn=420 fp=5434 fn=0)
threshold=0.100: precision=0.084768 recall=1.000000 f1=0.156287 (tp=527 tn=164 fp=5690 fn=0)
threshold=0.150: precision=0.083465 recall=1.000000 f1=0.154071 (tp=527 tn=67 fp=5787 fn=0)
threshold=0.200: precision=0.083215 recall=1.000000 f1=0.153644 (tp=527 tn=48 fp=5806 fn=0)
```