# Clustering for hybrid malware analysis and multi-path execution

*A thesis submitted in partial fulfilment of the requirements*

*for the degree of Master of Technology*

*by*

Vineet Purswani


DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

July 2017

# Certificate

It is certified that the work contained in this thesis entitled "Clustering for hybrid malware analysis & multi-path execution" by "Vineet Purswani" has been carried out under my supervision and that it has not been submitted elsewhere for a degree.

4.7.17

Dr. Sandeep Shukla

Professor

Department of Computer Science and Engineering

Indian Institute of Technology Kanpur

*July 2017*

# *Abstract*

---

Name of the student: **Vineet Purswani** Roll No: **12807813**

Degree: **M.Tech.**

Department: **Computer Science and Engineering**

Thesis title: **Clustering for hybrid malware analysis and multi-path execution**

Thesis supervisor: **Dr. Sandeep Shukla**

Month and year of thesis submission: **July 2017**

---

These days, the Internet faces a mounting threat from malicious software developers. Almost everyday new malware get into headline news, demanding ransoms and disrupting crucial services like power supply, healthcare etc. There is an urgent need to protect against them and defend our critical cyber infrastructure from such attacks. Manual malware analysis is not sufficient due to the huge number of such cases. As a result automated yet efficient malware analysis is much needed. Malware classification powered by a multi-path execution engine will be a great weapon for computer security industry. In this thesis, we present various feature engineering techniques that has lead to great improvements in classification of executables into malware families. Furthermore, we present a GDB extension tool to facilitate symbolic execution of binaries, primarily for multiple path execution purpose.

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

# *Acknowledgements*

I would extend my sincere gratitude to **Dr. Sandeep Shukla** for guiding me in this project. I would also like to thank **Pranjul Ahuja** and **Ajay Singh Chaudhary** for their help and cooperation during various phases in the project. I am also thankful to my parents and sister for the love they have given me.

I would like to take this opportunity to thank all my friends at the cyber security lab, especially, **Avik Dayal, Abhay Kumar, Saptarshi Gan, Neha Ajmani, Debleena Das and Saurabh Kumar** for making my stay memorable.

Finally, I am grateful to my friends at the hostel and outside, especially **Ludhiya Johnson, Ritesh Giri, Kishan Kumar and Ayushman Sisodiya** for being my greatest support.

I am grateful to VirusTotal for being generous and supportive towards my research and providing their private API to help me build the dataset. Finally, I would like to thank the open-source community for providing me with enormous support.

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| **PC** | **Program Counter** |
| **PCA** | **Principal Component Analysis** |
| **DLL** | **Dynamic-link Library** |
| **DBA** | **Dynamic Binary Analysis** |
| **API** | **Application Programming Interface** |
| **IoT** | **Internet of Things** |
| **GDB** | **GNU Debugger** |
| **GUI** | **Graphical User Interface** |
| **VM** | **Virtual Machine** |
| **VMM** | **Virtual Machine Manager** |
| **AMI** | **Adjusted Mutual Information** |
| **ARI** | **Adjusted Random Index** |
| **CTF** | **Capture the Flag Contest** |
| **SMT** | **Satisfiability Modulo Theories** |

*This thesis is dedicated to my parents*

*Shri Rajkumar Purswani*

*&*

*Smt Sneha Purswani*

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

# Chapter 1

# Introduction

Due to an outburst of malware hitting the cyber systems worldwide, computer security community has realized its responsibility and is fighting hard against the culprits. Malware analysis paradigm is undergoing a huge shift. Signature matching is now assisted by heuristic analysis techniques in almost all the good anti-malware software now a days. Furthermore, machine learning techniques are being employed to detect malicious activities automatically. There are several commercial products available in the market that are equipped with all the three pillars of malware analysis to guard against the offenders. Although, machine learning is already in use, both as in-cloud malware analysis service as well as intrusion detection systems mounted over firewalls etc., there is still a lot of scope left in improving the malware classification.

The greatest barrier against the problem at hand is the presence of huge variety of malware classes. Moreover, these classes are not even mutually exclusive. This is an active battle, therefore malware developers are also evolving alongside the security community, making their produce even more deceitful and crafty. To build a robust malware classification system, work has to be done on several fronts, such as constructing a substantially sized and balanced dataset, generating the most effective feature set and lastly employing learning algorithms that give the best prediction results. This thesis will deal with all the three domains of the malware classification problem, but will focus majorly on the fabrication of a good feature set.

# Chapter 2

# Background

With every passing day, human race is getting entangled deeper into the cyber-world, switching itself to e-commerce, online banking, social media networking etc. This has changed the modulus operandi of many human endeavors that were done manually 20 years back, empowering processes with increased efficiency and better management opportunities. Additionally, the Internet has given a sneak peek to a promising future, starting with the ease in the flow of information, social media connecting people from the different corners of the world, government organization going transparent with their operations and almost every institution shifting all their administrative paper work online. Collectively, they have made our lives easier, enabling us to leave the tiring job of data management, crowd monitoring etc., to computers.

But as the Internet has attracted more users, more grievous cybercrimes have come up. Crimes that were lurking behind due to sparse Internet availability and usage have become profitable. News about ransomware affecting huge fraction of the computer systems in the wild or attacking a particular organization and shutting its core operations for a day have recently made headline news. Many cyber incidents have occurred recently, including Mirai [1], WannaCry [2], Jaff ransomware etc. Criminals and terrorists have started using the Internet to mitigate their foul intentions, hampering critical services for hours and affecting economies. In addition to creating financial loss and great discomfort, these malicious software are now causing serious

heath problems. Several pacemakers have been compromised in the past, hospital servers shut down for upto 24 hours by malware causing serious repercussions on people's lives.

We need to act, and react strongly to such threats prevailing in our society. These activities have to be rebuffed by some sincere efforts put forward by computer security community. A lot of research has been done in the past, starting with the signature based malware detection to sandboxed automated malware analysis, yet the culprits have outmaneuvered all such efforts and are continuing to hamper services almost everyday. Anti-virus software have rendered useless in an era of millions of malware lurking around. Their signature based detection technique (the most primitive one) has become ineffective and cumbersome, especially for newly launched malware. Let us delve into malware types and how these attacks are mitigated to get a better insight of the issue at hand.

## 2.1 Malware

Malware, or malicious software, is a generic term used for any software that runs unexpectedly to lay some damage over a computer system, permanent or temporary, depending upon the ulterior motive behind injecting the malware. Whether it is a virus that spreads itself all over the filesytem and renders the computer useless, or a spyware whose sole job is to sneak into the files and send any critical information over the network to the attacker, or the latest - ransomware that encrypts all the files and asks for ransom in exchange of the decryption key, all of them come under the notorious category of malware.

This section covers the most common types of malware, ways these attacks are mitigated and the most recent defense mechanisms provided by the security vendors against them.

### 2.1.1 Types of Malware

Following malware classes are not mutually exclusive, i.e. several malware can be seen exhibiting characteristics of multiple of these classes. The classification is also not exhaustive, i.e. other types of malware may exist.

**Viruses** , as the name suggests, are those malicious software that hide themselves in other, presumably legitimate programs and spread themselves across the filesystem to infect as many things as possible. They serve as to damage the system in various ways like destroying crucial files, flooding either primary storage or secondary or even worse - both, rendering the system unusable completely. Viruses were the first seen malware, built primarily to brag about developers' technical skills. Some of the most destructive viruses [3] are Morris worm (which is a virus as well as a worm), Melissa virus [4], ILOVEYOU etc.

**Trojan Horses** are malicious software that disguise themselves as some useful program, but run malicious actions in the background. They are usually spread through some non-trustworthy website or emails, claiming to be a game, wallpaper or even malware scanners. Their malicious actions include installing keylogger, backdoor, or sniffing critical data out of filesystem etc. Some of the most destructive trojan horses are OSX/RSPlug Trojan [**?** ], Storm Trojan etc.

**Rootkits** function on a sole purpose of hiding themselves from users. They usually run with root privileges to hide their existence - processes and files from the users to avoid detection. They can be stealthy enough to make the host operating system run over a virtual machine and sit under it [5], hence making its detection almost impossible. Rootkits do not do harm to the system themselves, rather they incubate other malware like viruses, backdoors etc. Most common rootkits incidents [6] are Sony BMG copy protection rootkit [7], spambot mailbot etc.

**Backdoors** let an intruder to impersonate some user on a system usually through bypassing authentication. They can be utilized by setting up a communication channel between a compromised system and the attacker over the network. Backdoors are usually installed by other malware like trojan horses, worms

etc. It has been reported by security researchers that some computer vendors used to sell products with backdoors preinstalled in them for better technical support [8].

**Bots** are malicious software that allow their deployer/master to gain access to a system remotely. These are, inherently, backdoors installed over huge number of systems, primarily equipped to run spam campaigns, DDOS attacks etc. Some of the most deadliest botnets [9] of all time are Grum, Kraken and the most recent one, Mirai [10] etc.

**Worms** have the ability to replicate themselves to systems connected over a network to one of their victims. Technically, once a system is compromised with a worm, it starts searching for the other active nodes linked over the network, and tries to inject itself by utilizing some security vulnerability of the victim. Similar to rootkits, they do not inherently harm their victims, instead carry some virus, trojan horse or backdoor upon them. Some of the most famous computer worms [11] are ILOVEYOU worm, Code Red [12], Storm worm etc.

**Adwares** are used to show unwanted advertisements to victims, installed by the means of social engineering or pre-installed with some non-trustworthy software. Individually, they do not do any harm to the victim computer other than displaying unwanted content, but they can be accompanied by spywares or worms to spread themselves further into the network.

**Spywares** are malicious software that sniff sensitive information from the victim computer and sends it over to the attacker. This information can be bank account details, user passwords of a system, critical documents etc. depending on the attackers intentions. Some of the previous spyware attacks [13] are CoolWebSearch, Gator etc.

**Ransomwares** are the most notorious malware present these days. They encrypt the filesystem of the victim computer and demand huge sums of money (in form of bitcoins or other crypto-currencies) in exchange of the decryption key that is used to retrieve the files. There are increasing number of incidents recently reported where a ransomware spreads itself widely throughout the Internet, or probably over some small network, rendering those systems unusable until

the ransom is paid. Recent such attacks include WannaCry, Jaff ransomware, Petya [14] etc.

### 2.1.2 Available Defenses

**Anti-virus software** are the most primitive form of defense we have to our rescue. Initially, they worked solely on **signature matching technique**, that requires security analysts to generate signatures manually, which is a huge bottleneck. This technique is ineffective in detecting zero-day malware, which gave rise to other malware analysis techniques, among which the most recent ones use heuristic analysis and machine learning. Anti-virus software, or rather anti-malware software of today's era, bundle signature matching for definite classification along with heuristic analysis for raising warnings against possible threats. Some software provide automated sandboxed dynamic analysis, either on the host machine or on the cloud.

**Intrusion Detection/Prevention System** [15] is any device or piece of software that is used to detect malicious activities, either on a system or over a network. They incorporate signature matching technique with anomaly detection based on machine learning technique together to evaluate if an intrusion or policy violation has happened. Anomaly detection technique builds a user-specific profile, which stabilizes eventually, and can be used to raise alarms against any unusual behavior. Anti-virus software are also categorized under intrusion detection systems.

**Firewalls** [16] are used to monitor the incoming as well as the outgoing network traffic based on some predefined rules. There are several types of firewalls available today, different ones suited for different purposes.

**Application layer security, authentication, authorization etc.**

### 2.1.3 Triggers to Malware Development

There are several triggers of malware development, especially when the network of devices is growing rapidly. Following is a non-exhaustive list of such triggers

- Zero-day exploits are still difficult to get detected by our current defense systems. There are endless possibilities to build an exploit, hence none of the generic defenses work against them. Events like wiki-leaks making NSA zero-day vulnerabilities public, without notifying the concerned software vendors prior to it, can trigger malware development to a great extent.

- In an era of IOT, we can literally see an explosion of devices connected over the Internet. To serve millions of devices, for thousands of different purposes, we need gigantic amount of code, software, protocols etc., in place. Hence, explosion of devices has resulted into explosion of weak-spots in our network. There are so many malware attacks targeting IOT devices, like bulbs, refrigerators etc. that have weaker security than modern computer systems.

- Due to the inefficiency of current malware detection techniques, many variants of the same malware, that has been detected earlier, starts affecting users again. Methods like polymorphism -metamorphism and crypter-packers make it easier for a malware developers to build plethora of different binaries of the exact same program.

## 2.2 Malware Analysis Techniques

Malware binaries were analyzed manually by security analyst in the early days of anti-virus based protection. This was done to check if an executable was malicious and generate signatures in case they were. Efforts were made, and are still being made, to make this process automated. But, unfortunately, any generic automated malware analysis system does not work against all possible malware.

Even though the analysts still have to make the final call, automating the feature set extraction and building of a report comprising of behavioral and static analysis results, has helped them to reduce the amount of labor they have to do to analyze binaries. Moreover, application of machine learning over these reports have shown some promising results, further empowering these automated malware analysis systems.

In the context of machine learning techniques, static and dynamic analysis are employed to extract the relevant information about a binary. There is a lot of work done on dataset generation in the past, below are some of the most prominent ones.

### 2.2.1 Static Analysis

Static analysis, as the name suggests, is done without executing the malware binary. It has its biggest advantage of lesser resource and time requirement over dynamic analysis. On the other side, there are techniques to deceive static analysis altogether, such as obfuscation of a binary. This will be discussed in detail in the subsequent sections. In order to apply learning algorithms, static analysis results are to be molded into the form of a dataset. Few things that can act as significant features of this dataset are being discussed below.

**Working with the binary headers** - Program headers can give a lot of information including library imports, function call graphs, program sections and their sizes etc. Infact, it has been shown in several research work that addition of some of the optional headers like operating system, image size etc., have resulted in better insights.

**Working with the disassembly** - $n$-grams [1] of opcodes of the disassembled binaries can make a good feature set as well.

**Working directly with the binary** - 1-gram and 2-gram of bitcodes of the binaries has shown promising results in malware classification. Moreover, there is a separate party of researchers working solely on image representation of binaries and running classification algorithms on this kind of dataset.

#### 2.2.1.1 Hindrances to static analysis

Static analysis has the greatest drawback that the actual program can be disguised as some obfuscated or continually changing program. This may render static analysis

[1] $n$-gram is a continuous, usually overlapping, sequence of $n$ items of a given array

completely useless. Lets look at some of such techniques that can fool static analysis to a great extent.

**Polymorphism** in context of malware, means changing the actual binary in order to avoid detection by pattern matching. Packers or crypters are some examples of polymorphism done on binaries. Unpacking routines are attached to the packed code, that decrypts the actual program intent only on the runtime.

**Metamorphism** in context of malware, means continually changing the binary on every execution, so that a different version of it is distributed among the victims. Simple forms of metamorphism include adding random nop slides, permuting registers, adding junk code etc.

### 2.2.2 Dynamic Analysis

Dynamic analysis is done by executing a binary, preferably over a virtual/emulated system, and watching its behavior in terms of files read/written, network interaction, memory footprint etc. Below is a non-exhaustive list of dynamic analysis techniques.

#### 2.2.2.1 API/System Call Inspection

System call monitoring helps in detecting a program's interaction with the filesystem, network etc. Side-effect of any program will take place only through the system calls if it is running in user-mode, hence tracing them would give the best insight on its intentions. Generalizing, API call inspection would help in building a behavioral overview of a binary. This technique would not work if the program is running in the kernel mode.

#### 2.2.2.2 Information Flow Tracking

This technique can be used to check if any sensitive information is being handled in an unexpected way, such as being sent over the network. There are primarily two concepts here, taint source and taint sink, where taint source is responsible for

introducing new taint variables (such as some critical files, browser data etc.) where as taint sink is supposed to raise an alarm if any tainted information flows through it.

#### 2.2.2.3 Instruction Trace

Instruction trace, in the way they were executed, could be a useful feature set. There are several research works done with control flow graphs and finding their closest neighbors, though they didn't prove to be the best classifier.

#### 2.2.2.4 Multiple Path Execution

This is not a separate technique per se, rather a different way of conducting dynamic analysis. A binary is run in a virtualized environment once, at a particular instance of time while collecting its behavioral reports. It would be unfair to expect it would exhibit all its operations in that one execution. There are usually many, sometimes thousands of execution paths a binary can take. Considering just one and judging its behavior from that would most likely give inaccurate results. Hence, if somehow possible, a binary should be run in as many execution paths as possible while conduction its dynamic analysis. This will be discussed in greater detail in the next section.

## 2.3 Symbolic Execution and its Use-Case

### 2.3.1 Definition

Symbolic execution is a program analysis technique, primarily used to examine code coverage, or more specifically, discover various paths that can be taken by the program on different input sets. Symbolized inputs are interpreted as variables rather than concrete values, propagated down the program expressions in form of dependencies. If a branch condition is dependent on a symbolized input or its derivatives,

a constraint is built and solved to determine what input values will lead to which branch target (either both or one, depending on user preferences).

### 2.3.2 Why Symbolic Execution

In context of malware analysis, symbolic execution is supposed to aid the dynamic analysis engine to find as many execution paths as a binary can take. As mentioned in the earlier sections, knowledge of multiple execution paths will help in getting a better behavioral footprint of the executable.

There are numerous malware that lay dormant or inactive in unsuitable environment, for example if running in a debugging environment or over a emulator etc. Many show their malicious activities only a particular day, just like Michelangelo malware, which comes into effect only on March 6 (Michelangelo birthday). Hence, to get the most out of a binary and its dynamic analysis, it has to be tested under various circumstances. Symbolic execution could prove useful for this purpose. Exact implementation details will be elaborated in the subsequent chapters.

### 2.3.3 Shortcomings

Though it has a great significance in the software testing community, symbolic execution still lacks at some places, which tends to render this use case crippled to some extent. Its has three primary problems

- **Path explosion** due to unguarded long loops or huge number of branches in general
- **Path diversion** due to invisible code sections such as external library calls giving rise to inconsistent dependencies
- **Complex-constraints problem** probably due to non-linear dependencies leaves many execution paths of most real life software inexplorable.

# Chapter 3

# Past Work

## 3.1 Evolution of Malware Analysis

Malware analysis was initially done with pattern matching techniques (signature matching) to detect the known instances of malware present in the wild. It evolved to heuristic analysis very quickly to accommodate zero-day malicious binaries to some extent. There has been a huge development in machine learning paradigm in the last few decades, which has shown promising results on multiple fronts. Computer security community has been eagerly applying machine learning to malware analysis in order to automate this process as much as possible.

The sections below will enumerate useful tools for analysis purposes, followed by a non-exhaustive list of various research works done to apply machine learning techniques over malware analysis process in an attempt to automate it.

### 3.1.1 Binary Analysis Tools

Earlier, malware analysis was done manually to generate signatures, used later to prevent malicious attacks by already-seen malware instances. Currently, there are debuggers, memory dump tools, network scanners and/or sniffers etc to assist binary analysis. Though most of this work has been automated and is done inside a sandbox today, the security community still depends on these tools to a great extent.

There is a huge variety of tools available for every single use-case. For example, *IDA Pro* [17], *OllyDbg* [18], *GDB* [19] etc., are some of the most commonly used tools for debugging purposes. Similarly, *LordPE*, *OllyDump*, *PEiD* etc., assist in memory dumping or unpacking purposes. There are several tools that parse useful information out of binary headers, for instance, *resource hacker* can check, modify or delete any executable resources like file manifest, images, strings etc. Similarly, *dependency walker* is a windows tool distributed by microsoft, that lists out executable dependencies, i.e. its imports and exports. All the applications mentioned till now exercise their powers directly on static binary files. This comprises of static analysis, which is established to be weaker than dynamic analysis. Appearance of a binary file can be easily tweaked using obfuscators, PE/ELF header editing tools etc.

For network exploration, *nmap* [20] and its GUI extensions along with *wireshark* [21] are used majorly in the community. Process exploration is incomplete without monitoring its side-effects, such as file operations, memory footprint and registry modifications etc. Microsoft provides some tools for this purpose, such as *Process Monitor*, which displays file system interaction and registry modifications in real time, and *Process Explorer* which is an enriched task manager and system monitor. Lastly, some tools help in API/syscall monitoring, which is a tremendous enhancement to binary analysis due to the mere significance of API/syscalls. Building further upon the system call monitoring tools, there are several applications that help listing out dynamic library calls, which can be used to construct function call graph - a decent behavioral approximation of an executable. *ltrace* for library calls and *strace* for system calls are mostly used by the linux community for function call monitoring purposes. *ntrace* is their windows equivalent, which is not very well versed as its alternatives, yet is a very useful tool.

All the tools and applications described till now are being used since a long time. There are several latest additions to this list, which serve one layer above the tools discussed till now. *Yara* [22] is a malware family description tool, that operates on rules made on the basis of textual or binary patterns. *Angr* and *Triton* [23] are binary analysis frameworks, mainly comprising of symbolic and taint analysis engines. There are several other tools that are used by the malware analysts but are highly specific in their functionalities.

Due to the increase in the number of new malware being detonated everyday, manual analysis is insufficient for the security community. Moreover, dynamically monitoring an executable requires it to be run in a safe isolated environment, so that it does not render harm to the critical systems. To fulfill both these necessities, sandboxes equipped with the most of the tools described above have been developed, in order to automate the process of malware analysis in a confined yet efficient way. Some of the most popular sandboxes used in malware analysis context are *Anubis*, *CWSandbox*, *Cuckoo* [24] etc.

### 3.1.2 Application of Machine Learning Techniques

Almost every variety of machine learning algorithms have been used over malware datasets by the security researchers. Even the latest ones like deep learning have been employed, yet satisfying results are seldom found. A real-life dataset is difficult to generate, and even more grievous a problem is - absence of zero-day malware (the real culprits) in the test dataset. The problem at hand has similar solvability as that of *Belady's optimal replacement policy* for caches. The future is unpredictable, machine learning algorithms are suboptimal yet the best possible solution to the problem. The following subsections talk about the various learning algorithms applied, different data set construction techniques employed and the evolution of malware analysis under the hood of machine learning techniques.

#### 3.1.2.1 Static Analysis

One of the first research done to combine data mining with malware detection was by *Schultz et al.* [25]. They used windows PE headers, strings and fixed length byte sequence of a binary as their feature set and applied Multinomial Naive Bayes algorithm over it. They achieved a best-case accuracy of 97.11% over the dataset consisting of 4266 files. *Kotler et al.* [26] came up with a novel approach of using n-gram byte sequence, instead of non-overlapping, as done by *Schultz et al.*. They used multiple classifiers and deduced that boosted J48 classifier was performing the best. They used area under the ROC curve (AUC) with 95% confidence interval as

their evaluation parameter and got 0.9958 AUC for boosted J48 classification for a dataset of 3622 binaries.

Later, *Nataraj et al.* [27] proposed a method of visualizing binaries as gray-scale images. This helped them to employ image recognition techniques over the converted dataset. They reported an accuracy of 97.18% on a 25-class dataset of 9458 files. *Tian et al.* [28] showed function length frequency can play a significant role in malware detection. Function length is measured by calculating the number of bytes the function definition covered in a binary. They collected a 7-family dataset of 721 trojan horse malware and achieved an accuracy of 87.76% in their experiment.

*Santos et al.* [29] realized that the construction of a sufficiently diverse, balanced and labeled dataset is extremely difficult. Due to this major drawback of supervised learning algorithms, they proposed semi-supervised learning to detect malware. LLGC (Learning with Local and Global Consistency) was used to extract out the intrinsic structure of a binary.

**Limitations of static analysis**

It was established in the early years of this research that static analysis would not be sufficient to cope with the upsurge of the zero-day malware. There has to be a more robust analysis system which is less prone to the deceiving techniques that malware had started using by then. Obfuscation, header tweaking tools etc., made concealing the real intention of a binary extremely easy.

#### 3.1.2.2 Dynamic Analysis

Multiple publications came around the same time where sandboxes were starting to be used to evaluate binary's interaction with its operating system, such as [30] [31] etc. The idea behind dynamic analysis is to obtain some behavioral insight about the binary. Though, the initial developments required manual human involvement, groundwork was being laid for the scalable and automated malware analysis that is known to us today.

*Bayer et al.* [32] used Anubis sandbox to automatically extract execution traces of all the samples and built their behavior profiles which were further used as input

to an unsupervised learning algorithms. Their work also included taint information regarding the executable code itself, i.e. if the binary was touched at any point of the execution. They claimed that using a sub linear nearest neighbor finding algorithm, Locally Sensitive Hashing (LSH) made their model fast and scalable. They used a precision-recall graph to show the performance of their model. Along with that, they demonstrated the scalability of their setup by clustering a set of 75000 binaries in just three hours.

Very soon, more research was done using API/syscall sequence graph as the input to learning algorithms, such as WEKA library in [33], maximal common subgraph finding algorithms in [34] et al. *Nari et al.* [35] employed an unique approach of constructing network flow graph, which is essentially network interaction of a binary and supplying that to WEKA library to conduct clustering over the dataset.

**Limitations of dynamic analysis**

Dynamic analysis has two primary drawbacks, one high resource requirement and second limited code coverage. Several sandboxes had been developed to conduct dynamic analysis automatically and dump out relevant information for further investigation. But their resource requirements, even today, are high and time-consuming. There is not much that could be done on this front except enforcing an upper limit on the execution time of a binary.

Furthermore, a single execution path of an executable would not be a good measure of its complete intrinsic behavior. Hence, low code coverage became another challenge for dynamic analysis. There were several solutions proposed, for instance random inputs and mouse clicks, and they indeed helped reaching further down in the execution path. *Moser et al.* [36] proposed multiple path execution using symbolic execution powered by SMT solver engine. The authors demonstrated enormous increase in code coverage of the binaries run onto their setup, maximum of which was reported as high as 3413.58%

#### 3.1.2.3 Hybrid Analysis

*Santos et al.* [37] compared hybrid analysis and demonstrated that it achieves better results than both the analysis techniques performed individually. They constructed their dataset utilizing both static (sequence of op codes) and dynamic analysis (system call sequence) to employ various learning algorithms over hybrid, and individually to static and dynamic dataset, to prove their proposition. This work demonstrated the supremacy of hybrid analysis over the other contestants by using AUC as the evaluation parameter. Comparison was done between the various classifiers employed and the three dataset construction techniques used. It was established that hybrid analysis performs better than both of the other individual techniques, independent of the classifier used.

*Anderson et al.* [38] performed multi-kernel learning over the hybrid analysis dataset to classify binaries in two, benign and malicious, classes with the accuracy of 98.07%. They used static features from native binary file and disassembled assembly code, along with that conducted dynamic analysis to get control flow graph, dynamic instruction trace and system call trace. Over the accumulated data, they employed different kernels over different feature sets and trained a weighted multi-kernel Support Vector Machine to classify the binaries. They tested their approach on a dataset comprising of 780 malware and 776 benign binaries and got an accuracy of 98.07%.

There are several other combinations of malware analysis techniques that were tried to achieve state of the art performance. Primary goal of these experiments is, usually, to reduce false positives and false negatives [1] as much as possible. *Wang et al.* [39] proposed a hybrid approach to identify zero-day malware, utilizing misuse detection as the base layer with anomaly detection operating over it to further classify the outliers. This work was done over android applications using Cuckoo Sandbox for dynamic as well as static analysis. The experiments were done over a dataset comprising of 12000 benign samples and 5560 malicious ones. Their anomaly detection engine had a high true positive rate of 98.76%, false negative rate of 1.24% and false positive rate of 2.24%. The misuse detector could achieve 98.79% of accuracy in malware classification.

[1] Assuming benign cases as positive, false positive classification would mean malware classified into benign class. Similarly, false negative means benign binaries classified as malware.

**Challenges left unturned**

There are several challenges in the domain of automated malware detection that are yet to be solved. First and foremost, a well balanced (, labeled) dataset is extremely difficult to generate. Due to the critical nature of the malicious binaries, availability of such kind of dataset is kept restricted. Furthermore, zero-day malware are unpredictable and are capable of escaping a classifier which is not trained on similar type of binaries.

Due to the above stated reasons, there is always a doubt lurking that whether evaluation strategies are exhaustive and flawless. Attempts at clustering the binaries into malware classes had not been very successful. Lack of unambiguous dataset and evaluation strategies might be blamed for it.

## 3.2 Multiple Path Execution - why & how?

The previous section established the importance of high code coverage during dynamic analysis of binaries. There are several malware that lay in dormant state under unfavorable conditions, i.e. virtualized environment, date-time constraint etc. Behavioral insights are best achieved if there is maximal code coverage. There has been a very little research done in this domain in the context of malware. Following are the significant milestones achieved with respect to multiple path execution or symbolic and/or tainted execution in general.

*Bayer et al.* [32] realized the importance of taint information for detecting whether a binary is malicious or not. They tainted the binary code itself, to keep a check on whether the executable is copying itself to some other location, or changing itself in any way. This was supposed to help identify worms, trojan horses, self-modifying metamorphic code et al. This work did not involved multiple path execution in any way, but its mention is important as the idea of tainting and manipulating critical information/input to the binary started from such kind research works.

*BitBlaze* [40][41], built by the security group at Berkeley University, gained a lot of traction in the research community as it was one of the very few open source automated binary analysis sandboxes available of its time. *BitBlaze* is a suite of

tools to facilitate in analyzing a binary. It comprises of a static analysis component (*Vine*), dynamic analysis component (*TEMU*) and a dynamic symbolic execution engine (*Rudder*). *TEMU*, which is a sandboxing tool built to assist with fine grained dynamic analysis of binaries, uses *QEMU* for emulation and instruction tracing purposes. It can be configured to taint important information sources like keystrokes, network input etc. to keep a track of inputs influencing the samples' behavior.

*Moser et al.* [36] proposed the idea of multiple path execution for the first time for malware. Their work carefully devised a plan to build a multiple path execution engine over the QEMU emulator. They used symbolic execution to build branch constraints and SMT solvers to solve those constraints to explore different paths. Furthermore, they formulated steps saving and restoring program state efficiently and consistently so as to cover as many execution paths in one run as possible. They addressed several limitations of symbolic execution, like path explosion, complex constraint problem etc and proposed sub optimal solutions for them. This work, especially building the GDB extension - Trida, is highly motivated by *Moser's* work.

# Chapter 4

# Problem Description

## 4.1 Clustering with Hybrid Analysis

**Goal**

Static analysis has its own demerits, yet is a powerful tool in itself. Augmenting it with dynamic analysis can complement it by filling up its pitfalls and adding more insight to the classifier. The ultimate goal was to train a robust clustering model on hybrid feature set and on the way, test which features influence the decision the most and experiment with various feature engineering techniques over the dataset. Finally, this model was planned to be integrated with Cuckoo - an open source automated malware analysis system.

**Inspiration**

Clustering algorithms employed over static and dynamic analysis individually has shown promising results [37]. There is some research done over hybrid-clustering model as well, but those are very few and their evaluation was insufficient. Training a classifier over a decent size dataset may provide robustness against new unseen malware, i.e. zero day exploits.

**Integration with Cuckoo**

Cuckoo is a widely used open-source sandbox primarily used for automated malware analysis. Any addition to it would be easy to be carried forward. Moreover, there are so many sub-problems to automated malware analysis and all of them should

be handled on one single system to bring all the research together and give rise to a great product.

**Challenges**

Availability of malware binaries was the first challenge of this problem. Building a good dataset that comprises of sufficient number of the malware classes is a crucial step. Further, size of dataset should be big enough to make the model robust against new comers of the malware family. Apart from this, evaluating results was another challenge of this work.

## 4.2 Facilitating multi-path execution with GDB

**Goal**

There are only a few multi-path execution engines available in the open source community like TEMU. But none of them goes with cuckoo, and they take a lot of effort to set them up. Goal of this thesis is to build a multi-path execution engine that is compatible with cuckoo, and can be injected into the VMs easily.

**Inspiration**

Inspiration behind working on multi-path engine was the strong drive of getting better, more sensible features by increasing the code coverage during program execution. Further, this engine should be compatible with cuckoo, for the similar reasons stated above. Cuckoo should become a one-stop project for malware analysis.

**Challenges**

Multi-path execution running over a VM for a native process is not feasible, as it would require a lot of resources and time to take snapshots and rollback to the previous state and take another route in the execution tree. Moreover, getting instruction trace outside the VM would not be possible. Therefore, a workaround had to be planned and implemented. There were few more implementation challenges, due to the intricate nature of this work, but work dealt with satisfactorily.

## 4.3 Summary

Summarizing the task on-hand, this work will employ various clustering algorithms over hybrid dataset, i.e. the one obtain after static and dynamic analysis both. Furthermore, feature engineering techniques will be applied to test their influence over the clustering model. The best results will be integrated into CuckooML module, making it easily accessible for others aspiring to continue this work further.

In addition to that, this work will extend further to develop a multiple path execution tool. This tool will be made compatible with Cuckoo, i.e. it could be run over a debugger/tracer instead of an emulator like QEMU.

# Chapter 5

# Enhancements to CuckooML

## 5.1 Standalone Program : Clustering

This section will explain about the work done on training a clustering model in a standalone program. This was done to get the best possible classifiers and feature engineering techniques in place before committing them to Cuckoo. Integration with Cuckoo sandbox will be discussed in the next section. Here we will delve more into the research done to figure out the most suited combination of feature set + clustering algorithm to give us the best results.

### 5.1.1 Feature Set Building

First of all, all the malicious binaries were pulled from VirusTotal [42] using their private API. These binaries were then fed into Cuckoo to generate their analysis reports. Cuckoo has numerous modules like behavior analysis, screenshot, static analysis, signatures dealing with some heuristics, virustotal etc., and all of them have their respective switches to toggle their activity. This work was primarily done on behavioral and static analysis, hence switching all the other modules off. Cuckoo and its modules will be explained in length in the subsequent sections.

Below is the exhaustive list of features that were used in this work, explaining a bit of intuition behind their usage.

**Static analysis features** As mentioned in the literature, clustering was done on static features to see how well they perform. There is a huge advantage of saving on resources and time while using purely static analysis.

- **Sections**: $< section\_name >$ were added to the feature vector, taking an intuition from the Figure 5.1 which depicts a stark difference between the sections present in the malware and benign binaries.

FIGURE 5.1: Comparison between sections present in malware and benign binaries

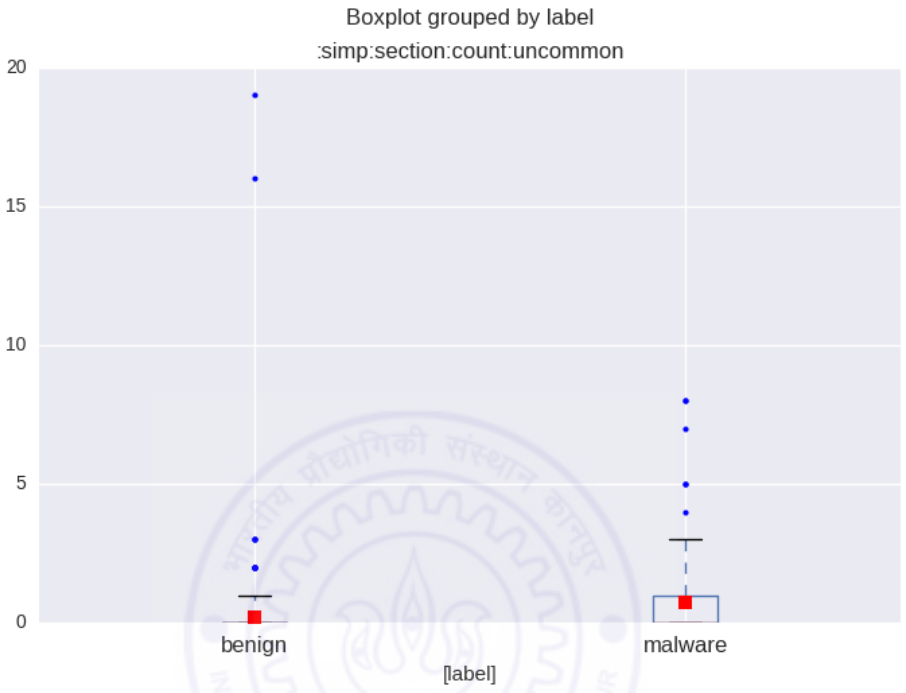
| | Name | benign | malicious | total |
|---|---|---|---|---|
| 56 | data | 0 | 12 | 12 |
| 57 | upx1 | 0 | 15 | 15 |
| 58 | upx0 | 0 | 15 | 15 |
| 59 | code | 0 | 15 | 15 |
| 60 | .didata | 0 | 18 | 18 |
| 61 | .itext | 0 | 20 | 20 |
| 62 | .bss | 1 | 25 | 26 |
| 63 | .ndata | 0 | 36 | 36 |
| 64 | .tls | 9 | 33 | 42 |
| 65 | .idata | 1 | 45 | 46 |
| 66 | .reloc | 8 | 71 | 79 |
| 67 | .rdata | 3 | 161 | 164 |
| 68 | .data | 197 | 163 | 360 |
| 69 | .text | 200 | 169 | 369 |
| 70 | .rsrc | 200 | 178 | 378 |

$< section\_count >$ was added to the feature set and was found playing a significant role in the classification. The intuition, again, comes from the fact that malware binaries have to disguise their malicious code, hence they use more number of sections on an average than benign binaries. The boxplot [1] figure 5.2, where `:simp:section:count:uncommon` refers

[1]Boxplot is a demographic used to illustrate a discrete set of values in form of quartiles. It does not make any assumption on the underlying distribution while displaying the variety of statistical parameters. Boxplots shown in this document have two tweaks made to the standard representation, i.e. mean of the data distribution is shown using a red square and outliers are shown using blue dots.

to the total number of uncommon (not in top $\lambda$) sections present in a binary, is a good assertion of the above proposition.

FIGURE 5.2: Boxplot showing variance in uncommon section count

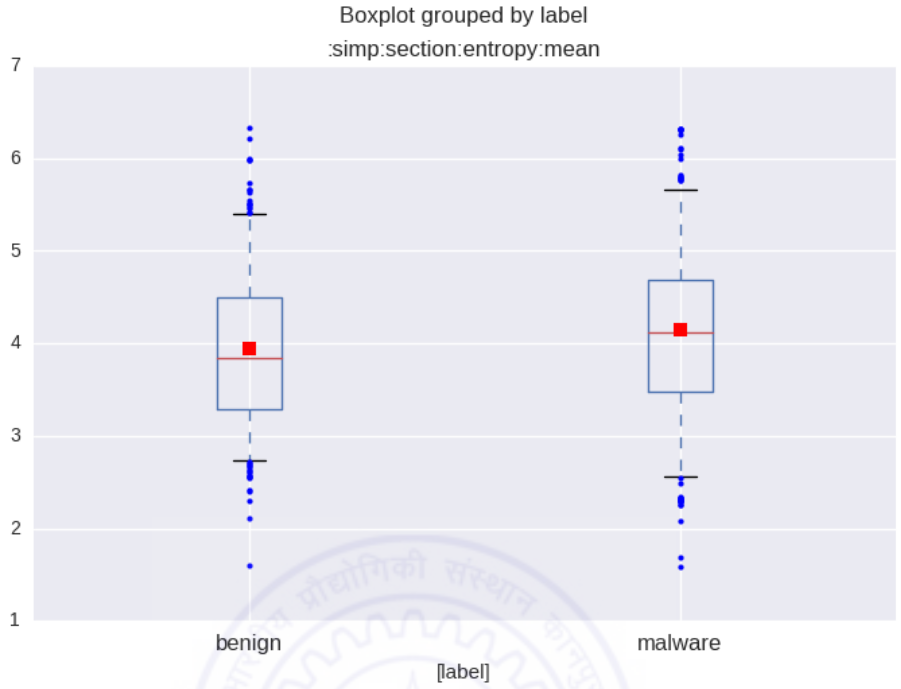



$< stat\_parameter, section\_entropy >$ [2] , where *stat_parameter* represents statistical parameters such as mean, mode, max, avg etc., were added to the feature set. In a work done by *Lyda et al.* [44], they demonstrate that the section entropies play a crucial role in determining whether a binary is packed or not. Any obfuscation, either packing or encryption, usually leads to an increase in section entropy. Figure 5.3 demonstrates the distribution of average section entropies of malware and benign binaries individually.

- **Static imports**: Library imports play a crucial role in determining a very basic overview of the intent of the binary in consideration. Hence, they are really significant for the clustering model. There is again a clear difference between the DLL imports of malware binaries as compared to benign binaries, as shown in the Figure 5.4

[2]Section entropy is the measure of randomness in the contents of a section. It is usually calculated by using Shannon Entropy formula [43].

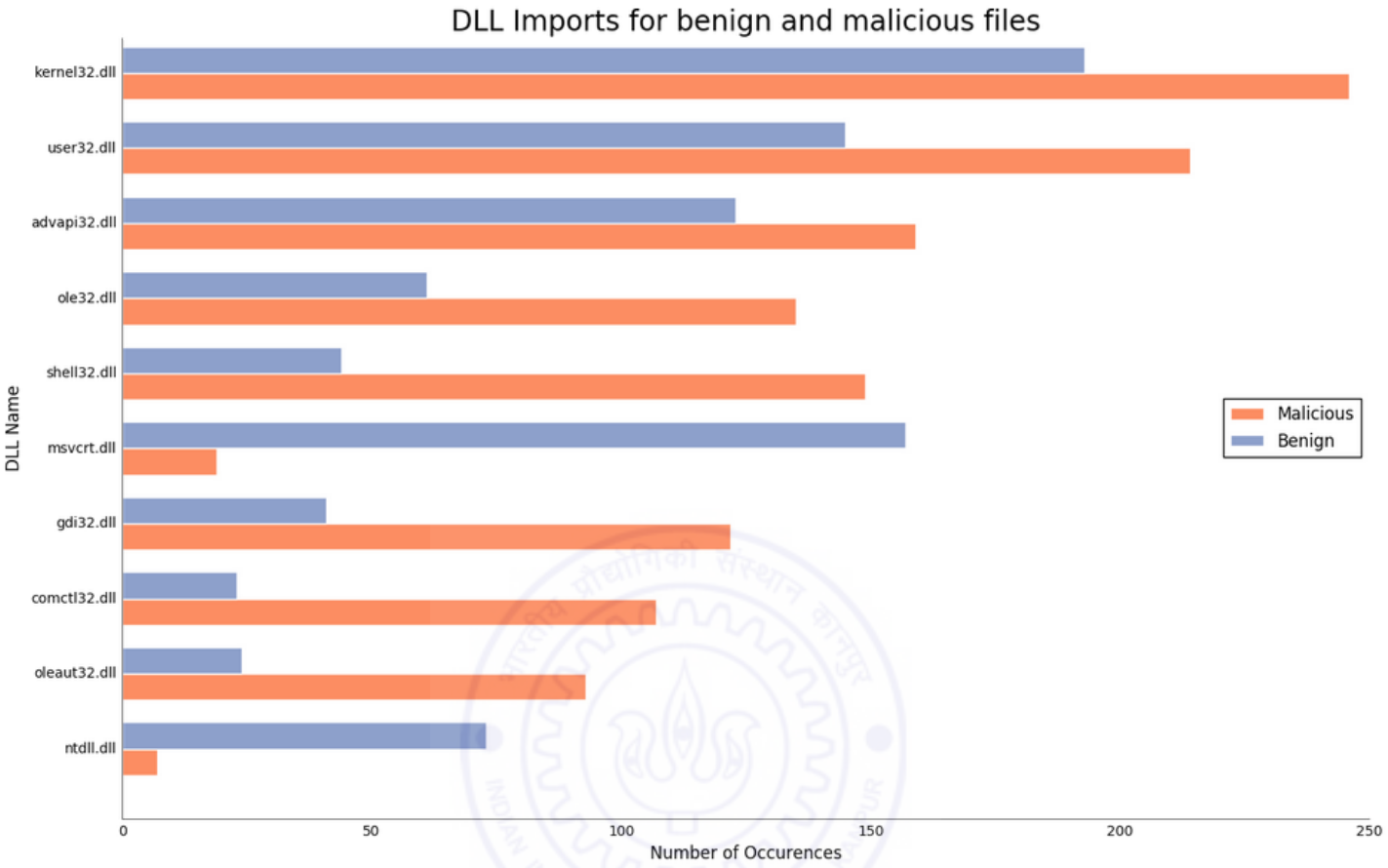
FIGURE 5.3: Boxplot showing variance in average section entropy




- **Optional Headers**: There were several research done such as [45] etc. to prove that a feature set purely comprising of the optional headers can make reasonable predictions on whether a binary is malware or benign. Hence all the optional headers were added to check if they can really make any difference to the classification. Figure 5.5 depicts the distribution of the sizes of malware versus benign images/binaries.
- ***n*-gram Bytecode**: There is always a distinguishing pattern in every binary, which is why image representations of binaries have shown good prediction results. In an attempt to exploit this fact, most frequent 1-gram bytecodes were added to the feature set and they indeed proved to be significant. Computation of 2-grams or 3-grams have huge memory requirements but low gains, hence they were discarded. Experiments were done with 1-grams and 2-grams to measure the performance gains inflicted by them.

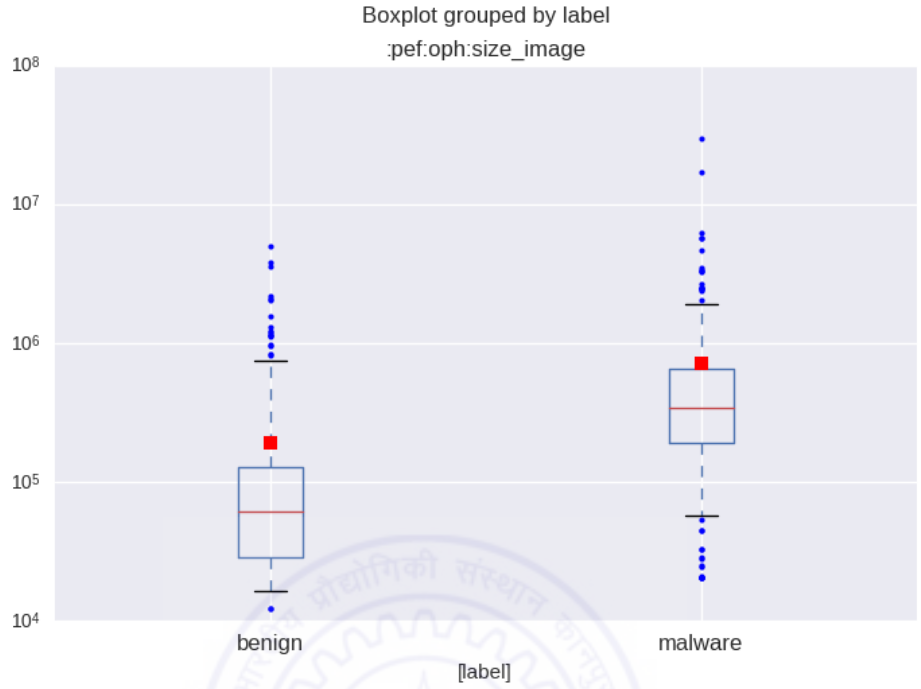
**Dynamic analysis features** Even though static feature set gave really good clusters, just to check how well dynamic features are gonna perform, clustering was done first on just dynamic features and then on the hybrid feature-set.

FIGURE 5.4: Comparison between DLL imports in malware and benign binaries




- **Mutexes modified**: Software use mutex for multiple purposes, one of which is to check if an instance is already installed. Every benign binary has a very unique static mutex name to serve the purpose. On the other side, malware binaries are left with either to use random mutex or get their mutex names distinguishable hence prone to be caught. Due to these reasons, mutexes are worth a shot, hence were added to feature set.
- **Dynamic imports**: Just like static imports, dynamic imports further add to the behavioral detection of a binary, hence play a crucial role in the feature set.
- **Files modified**: A distinguishing signature for a process can be modeled out of the files it has touched. Although, using file names directly as binary features did not add much to the classifier prediction, probably because it clutters the feature set more than adding to the insight of the

FIGURE 5.5: Boxplot showing variance in size of image - optional header



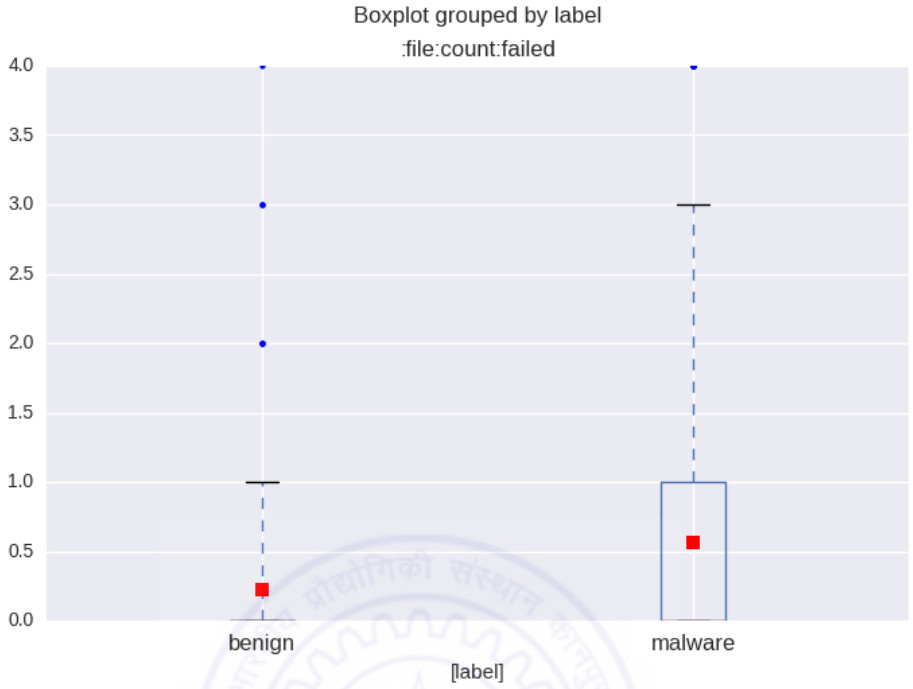

classifier. Feature binning technique can become useful for such cases. Figure 5.6 illustrates the distribution of the number of failed file accesses done by malware versus benign processes.

- **Network interaction**: There are several known malicious IPs and domains. Network interaction might help to distinguish binaries on the basis of malicious URLs accessed.
- **Registry modified**: Registries are used to do book-keeping tasks and can help build an identity of a binary solely based on which one of them were modified.

### 5.1.2 Clustering Algorithms

Clustering was applied on static and dynamic features, first individually then combined. Several of the $scikit - sklearn$ [46] clustering algorithms were tried, but the most relevant results were given by $DBSCAN$. Even according to the comparison done between several algorithms, $DBSCAN$ is the best suited for non-flat geometry

FIGURE 5.6: Boxplot showing variance in failed file access count




and uneven cluster size.

Density-based spatial clustering of application with noise, or (**DBSCAN**) [47] is the most cited clustering algorithm in machine learning literature. It builds clusters based on the density of data points. Fundamentally, it groups closely packed points, or neighbors, together into one cluster. It has two primary parameters, namely

- *min_samples* determines the size of a cluster. Formally, every cluster has some *core_samples* and there must exist at least one data point, known as *core_sample*, such that there are at least *min_samples* of other data points within *eps* distance to form a valid cluster.
- *eps* is the distance threshold to construct a valid cluster along with *min_samples* parameter. Lesser the *eps*, greater the cluster density.

Hierarchical DBSCAN [48] is an enhancement to DBSCAN proposed by the same authors. It extracts out the most prominent clusters and it is already being used in CuckooML module.

### 5.1.3 Feature Engineering

Clustering algorithms are really sensitive to feature space dimensions. Huge number of features will render even the most favored algorithm such as DBSCAN, ineffective. Hence, feature engineering has to be employed over the dataset collected, as the feature dimension in this case easily reaches tens of thousands.

**Feature extraction** : PCA, principle component analysis [49], was used as the first try in favor of feature extraction. Feature space was reduced to just 50-100 features, accounting for 94-96% of the dataset variance. Reducing feature dimension to this extent made clustering really efficient and meaningful clusters were formed.

**Feature reduction** : Feature reduction, in the simplest form means to select a subset of the feature space in an attempt to get better classification. This is usually done by measuring feature significance by training other classifier. Here, we used Random Forest [50] and XGBoost [51] classifiers to accomplish the task. Labels were assigned based on the VirusTotal malicious score.

TABLE 5.1: Table showing feature importances of top 20 features, according to XGBoost classifier

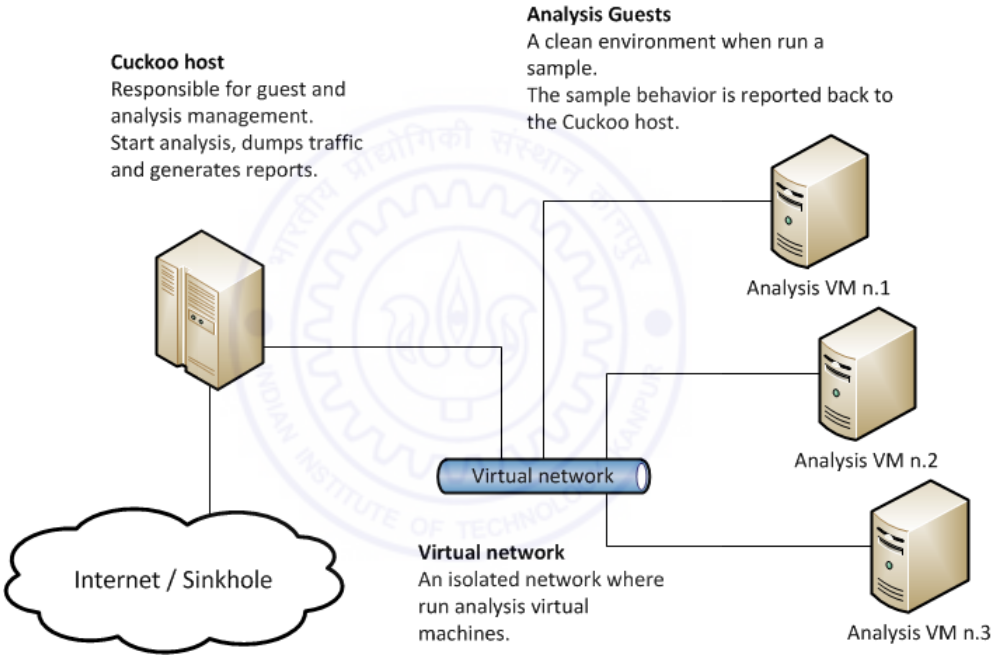
| Feature Name | Feature Importance |
|---|---|
| :file:touch:c:\documents and settings\cuckoo\local settings\temp | 0.02079002 |
| :pef:lang:neutral | 0.02079002 |
| :pef:oph:IMAGE_DIRECTORY_ENTRY_RESOURCE_rva | 0.02079002 |
| :pef:oph:IMAGE_DIRECTORY_ENTRY_RESOURCE_size | 0.02079002 |
| :simp:section:entropy:max | 0.02079002 |
| :pef:oph:IMAGE_DIRECTORY_ENTRY_IMPORT_rva | 0.022869023 |
| :pef:oph:size_image | 0.022869023 |
| :simp:section:entropy:mode | 0.027027028 |
| :simp:section:entropy:mean | 0.027027028 |
| :pef:oph:IMAGE_DIRECTORY_ENTRY_IAT_rva | 0.031185031 |
| :pef:oph:IMAGE_DIRECTORY_ENTRY_IAT_size | 0.033264033 |
| :pef:oph:address_of_entry_point | 0.033264033 |
| :simp:section:entropy:median | 0.033264033 |
| :simp:section:entropy:stdev | 0.037422039 |
| :pef:oph:IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT_rva | 0.04158004 |
| :pef:oph:size_init_data | 0.047817048 |

## 5.2 CuckooML : Integration

Cuckoo Sandbox is an opensource automated malware analysis project, maintained by a huge community. A basic introduction to Cuckoo and its architecture is discussed below.

### 5.2.1 Introduction to Cuckoo

FIGURE 5.7: Architecture of Cuckoo Sandbox. Source: docs.cuckoosandbox.org




Cuckoo sandbox is central management system that does the book-keeping job of handling program execution and their analysis with the help of a virtual machine manager like VirtualBox, Xen etc. It sits along with a VMM on a system, inspecting and instructing VMs to execute supplied binaries and collecting results once they are done. Every binary is executed in a separate VM, which starts from the uninfected-state-snapshot taken at the time of setup.

There are 3 crucial components that will matter the most to us - modules, signatures and analyzer.

**Modules** are, essentially, plug-ins to cuckoo system. They are either run before execution has begun or after the execution has reported dynamic results. For example, *static* module starts functioning as soon as an executable is put into cuckoo to parse program headers and generate static features. *mongodb* module stores data to maintain consistency while parallel execution on multiple VMs.

**Signatures** are, usually, heuristic predictions over the analysis results. For example, a signature reports a binary to be packed if its average section entropy is more than some threshold.

**Analyzer** is the component sent inside the virtual machine to directly interact with the execution happening. It is responsible to start the execution, collect analysis results and send back to the cuckoo engine sitting outside the guest environment. It carries various libraries along with it to be injected into the binary execution environment.

### 5.2.2 CuckooML Module

CuckooML is a separate open-source project, maintained by Honeynet [52]. It is just an additional module which parses Cuckoo analysis reports, constructs a feature set and trains HDBSCAN [53] or DBSCAN clustering algorithms over it. There are several configurations, or rather parameters to this module like *clustering_algorithm*, *anomaly_detection* etc., and are self-explanatory. It runs when cuckoo starts initializing its components, builds the model and dumps classification reports.

### 5.2.3 Changes made to CuckooML

CuckooML was already using HDBSCAN and/or DBSCAN, which came out to be the most optimal clustering algorithms in this work as well. In terms of feature reduction, it was pruning sparse features based on some threshold. Contribution made to this module are mentioned below.

#### 5.2.3.1 Features

In the standalone classification done earlier, few static features proved to be really significant, yet they were absent in Cuckoo engine altogether. Hence, some changes were made in *static.py* module to include optional headers and 1-gram category of features in static analysis report and later to be parsed by *cuckooml.py* module.

```
    def __get_option_headers(self):
        extracted_features = {}

        try:
            extracted_features['magic'] = self.pe.OPTIONAL_HEADER.Magic
            extracted_features['major_linker_version'] = self.pe.
    OPTIONAL_HEADER.MajorLinkerVersion

            .
            .
            .

            return extracted_features
        except AttributeError:
            return extracted_features
```

LISTING 5.1: Optional header function added to static.py

#### 5.2.3.2 Feature Extraction : PCA

Nominal feature set of CuckooML module easily reaches to tens of thousands of features, which makes feature reduction and/or extraction necessary. CuckooML has one such technique in place, which removes features with sparse vectors and it works fine. PCA is another, more efficient feature extraction technique. It has been added to the CuckooML module and the performance will be compared in the evaluation section.

```
def filter_dataset_pca(self, dataset=None, variance_coverage=0.94):
        """Feature reduction using PCA"""
```

```
        if dataset is None:
            dataset = self.features.copy()

        scale(dataset)
        pca = PCA().fit(dataset)

        variances = pca.explained_variance_ratio_.cumsum()
        num_cols = next(i for i in range(len(variances)) if variances[i
    ] > variance_coverage)

        return pd.DataFrame(data=pca.transform(dataset)[:,:num_cols])
```

#### 5.2.3.3 Feature Selection : RandomForest, XGBoost

Another feature reduction technique - selecting significant features by training a XGBoost classifier has been added. Labels were derived from malicious score from VirusTotal module.

#### 5.2.3.4 More feature engineering : Binning

**File names** are added to feature set in form of binary features, which does not add much to the accuracy of the classification. Individual files might not be used by many programs, hence would become a sparse, insignificant feature and will be ruled out in feature reduction phase. To actually make use of this data, files modified are to be put into meaningful bins, like folders touched, or file extensions etc. These features might prove to be more useful for classification. This has been added to the module.

Similarly, **section names** as features can be empowered by simply adding one more feature to the set - number of uncommon sections. Malware usually employ PE header editing tools and other deceiving techniques to hide their malignant intent as much as possible. Hence, they sometimes end up using more sections than the usual. Although, obfuscation is used by several benign software as well, even then

malware are the ones employing these techniques predominantly. To get the uncommon section list, we kept a relative threshold to decide what fraction of sections were to be called common and the rest uncommon.

Uncommon feature count was added to the feature set with respect to static library imports as well as dynamic imports individually. The DLLs that were present in uncommon list were not added to the feature set altogether. This had two-fold advantage - first, significant features are made and added to the dataset and second, huge number of features holding lesser significance are removed making the dataset short and concise.

## 5.3 Summary

This section summarizes the work done in the first part of this research. In the standalone work, static analysis was conducted over the supplied binaries and appropriate features were extracted from the lot. Clustering algorithms such as DBSCAN, K-Means, HDBSCAN etc., were then applied on the dataset to test how well does the static analysis perform. Later, while moving on to the dynamic analysis, all the work was shifted to CuckooML module. There, static features that proved to be significant in our standalone program and were absent in the module, were added. Furthermore, feature engineering was performed over the feature set constructed by the CuckooML module to achieve better clustering classification.

# Chapter 6

# Trida: GDB powered by Symbolic Execution

Objective of this part of the work was to develop a mechanism to interact with symbolic execution engine such that performing multiple path execution could become extremely simple and intuitive. Furthermore, the possibility of automating the process of multi-path execution had to be present. Keeping all these things in mind, GDB was chosen as the tracer and Triton as the symbolic execution engine.

Triton engine along with GDB powered by PEDA were used to implement a multi-path execution GDB extension. Purpose of this extension is to enable symbolic execution of a part or whole of a program while it is being debugged over GDB. Integration of the two components will help in better visualization and inspection of executables with the assistance of PEDA while symbolically finding inputs to reach different sections of the code using Triton engine.

## 6.1 Preliminary Structure

**Triton** is a dynamic binary analysis framework[23], which consists of a symbolic execution engine, a taint engine, a SMT optimizer-solver engine and a python

binding (over its C API) to access these layers. It is primarily for `X86` and `X86_64` instruction set, which makes it useful for this work.

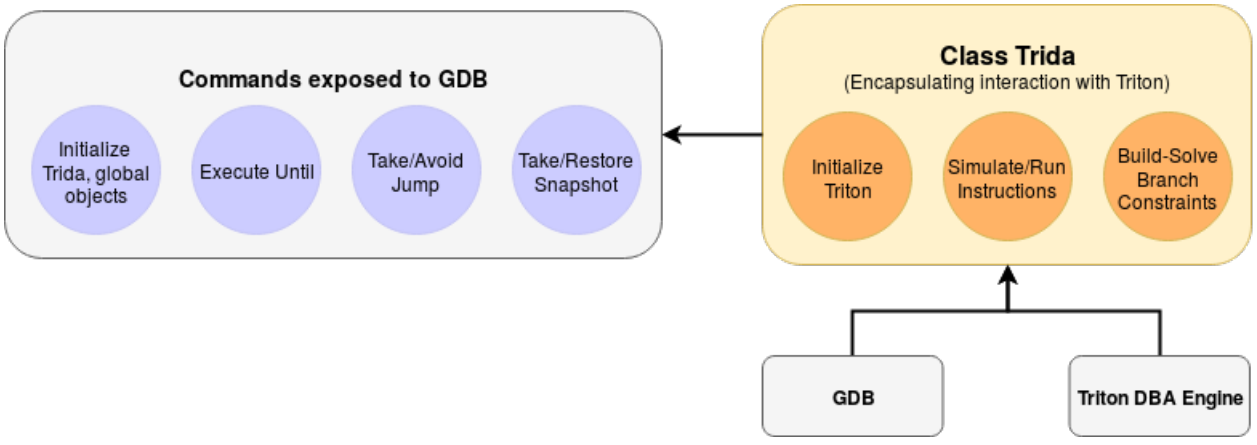
**Peda** is another really popular opensource GDB plugin that brings useful tools handy inside GDB. Basically, it abstracts out the most commonly used GDB commands in pythonly fashion. It has a well-organized source code which makes it easier to develop further over the first layer of abstraction provided by PEDA.

### Why Triton + GDB

Any symbolic execution engine, including Triton, will need a tracer like Intel PIN, QEMU etc. to get the instruction trace to ignite its components. GDB might not fall into the same category, but it will do the job just fine. It is one of the most powerful debugging tools available for Linux and Windows users. Moreover, it has a powerful python API in place, which is further empowered by PEDA. Motivation behind using Triton is the excellent open-source community backing this project. This whole combination makes the supposedly multi-path execution environment easily accessible with low setup overhead.

## 6.2 Implementation Details

FIGURE 6.1: Overview of Trida



Diving straight into the implementation details, development of Trida was done in two stages - `Trida` class and `PEDA` commands. Figure 6.1 is an architectural overview of Trida GDB extension.

### 6.2.1 Trida Class

This class is responsible to bundle all the interaction with Triton engine into one entity. Trida is supposed to provide the following functionalities to get the work done alongside GDB,

- Initialize Triton engine, turn on SMT optimizations, set system architecture and initialize emulator registers.
- Process supplied instructions into symbolic execution engine to generate dependencies down the execution path.
- Build and solve branch constraints, either to take or avoid the branch.

Figure 6.2 depicts the work-flow of activities happening inside Triton engine while Trida extension is being used over a binary. Detailed documentation of the Trida class members can be found at A.2.2.1.

### 6.2.2 PEDA Commands

Once the basic requirements of working with Triton are in place, the next step would be to expose GDB commands to facilitate co-existence and consistency of GDB and Triton execution states. These commands will be responsible for the following things primarily,

- Initializing the Triton engine through Trida class and assisting in all the further interactions between the two engines, GDB and Triton. Their interactions include operations such as executing instructions while ensuring (limited) consistency of the two engines, adding symbolic inputs and taking/avoiding symbolized jumps.

- Taking/restoring program state snapshot, i.e. capturing GDB as well as Triton program state.
- Ability to synchronize GDB with Triton program state and restore Triton to GDB current state.
- Additional support to put constraints on the inputs - `printable` and to enable short-circuiting the execution for optimization purposes - `noextern`.

List of the commands that are exposed along with some documentation can be found at A.2.2.2.

### 6.2.3 Intricate Details

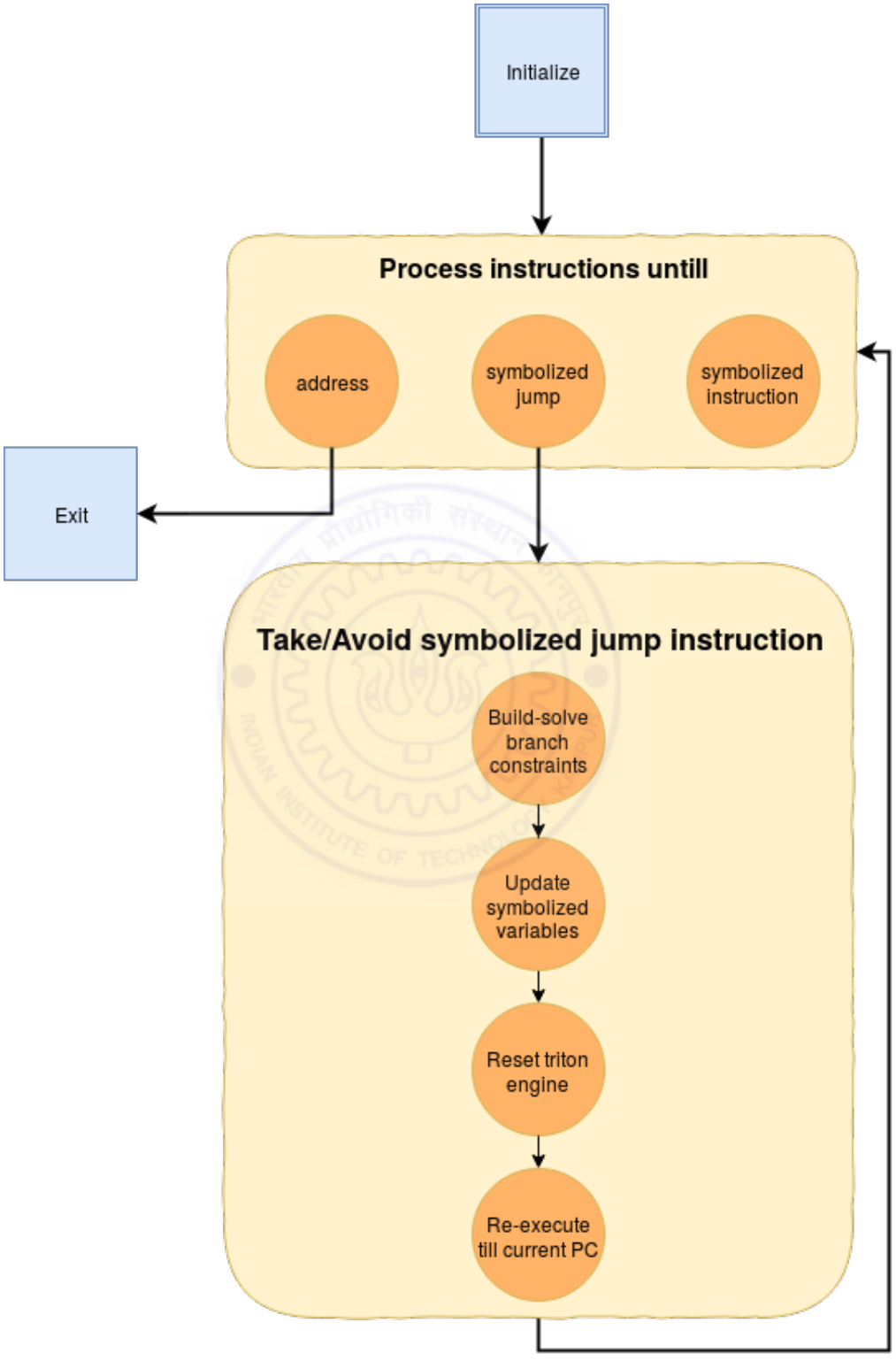
There were some dubious situations that had to be dealt with to fulfill the objective of this tool. Following are some of such cases.

**Optimizations** such as constant folding, memory caching were employed to improve performance. Constant folding refers to the collapsing of sub-AST (abstract syntax tree) of a non-symbolized-node. Memory caching was implemented to prevent frequent memory query calls to the Triton engine.

**Limited consistency** refers to, in context of Trida, maintaining the consistency with respect to current PC only, which is another optimization to enhance its performance. Memory and register values, especially those dependent on symbolized inputs, are not synced between the two engines. This can be manually done by using `symsync` command, which replicates the program state completely from Triton to GDB. Limited consistency was achieved by a sub-optimal approach, wherein instructions were re-executed till the current PC inside the Triton engine, whenever an input variable was altered (`symtake\symavoid`). A *jump counter dictionary* was maintained to keep a track on the number of times symbolized jump instructions (not all instructions [1]) had been touched, so that the re-execution could attain the correct program state.

[1] because the re-execution will happen only when `symtake\symavoid` commands are called, that in turn work only when the PC is pointing to a symbolized jump instruction

FIGURE 6.2: Triton usage work-flow wrt Trida



**Snapshotting** , while using Trida, requires capturing both - GDB as well as Triton execution state. GDB execution state can be captured by using PEDA snapshot function, that dumps all the registers and the entire memory. Triton would require to keep a copy of the *jump counter dictionary* only, to reach the desired program state. The only other thing needed would be the current PC, which would already be present in GDB snapshot.

**User-given input constraints** , play a significant role in assisting the SMT solver to deliver a desired model in lesser amount of time. This was handled by adding support for keyword `printable`, that ensures the input variables lie in printable character range. These constraints had to be added to the branch constraints everytime they were built.

## 6.3 Summary

Realizing the worthiness of multi-path execution to malware analysis, this part of the research delves into the development of a tool called Trida, a symbolic execution extension of GDB debugger. It is build over a very popular GDB extension, Peda, to utilize its abstractions of operations that were crucial to the development of Trida. The core idea behind the tool functionality is inspired by the work of *Moser et al.* [36]. This section talked about the specifics of why a particular technology was chosen to build this tool and how exactly this tool was built.

# Chapter 7

# Analysis and Conclusions

## 7.1 Clustering over hybrid malware analysis

There were several experiments done during this research work, with respect to clustering algorithms used, feature set building and further feature engineering over the dataset. But the most significant ones were added to the CuckooML module. This section will compare the performance boost seen as the changes were made to the module. It will also try to make sense out of the clusters made, check if they had any latent class of malware hidden in them.

### 7.1.1 Evaluation Setup

All the experiments were conducted over a commodity machine with Intel i7 3.40 GHz processor, 16 GB of main memory and 1 TB of hard disk. The operating system was Gnome Ubuntu 16.04 LTS.

For dataset generation, malicious binaries were pulled from VirusTotal through their private API. Python modules like *pefile*, *peframe* etc were tried to extract static features. Soon, all the analysis was shifted to Cuckoo Sandbox and its modules.

The results are evaluated over two different sizes of dataset. Initial ones are evaluated over 1000 binaries, out of which 480 were benign and 520 malware. The final results, for the comparison between various feature engineering techniques, were evaluated over a dataset of 3000 binaries. HDBSCAN clustering algorithm is used in all the evaluation results presented.

### 7.1.2 Evaluation Metric

Evaluation of clustering algorithm is an ambiguous domain, especially for malware clustering. The results of this work include popular evaluation metrics, that are Adjusted Mutual Information Score, Adjusted Random Index, Completeness, Homogeneity and V-measure. All of these metrics are explained briefly in the subsections below. Further, the last section will try to abstract out some meaningful clusters.

**Adjusted Mutual Information Score** is a measure of agreement in two assignments, by calculating their Mutual Information Score. Mutual information roughly means the interdependence of two assignments on each other. The score is in [0,1] range, 1 being the perfect.

**Adjusted Random Index** is the similarity measure of true class distribution and predicted class distribution. It measures the similarity on the scale of [-1,1], 1 being perfect score and any value close to 0 or negative is a bad score.

**Completeness** If all the members of a true class are assigned to a same cluster, the class is said to be complete. Score is in range [0,1], 1 being the perfect.

**Homogeneity** If all the members of a cluster belong to a same class, then the cluster is said to be homogeneous. Range - [0,1], 1 being the perfect

**V-measure** is the harmonic mean of homogeneity and completeness.

### 7.1.3 Comparing CuckooML : native and enhanced

**Effects of adding PCA feature extraction**

The evaluation metrics of the clustering done by the native CuckooML module in comparison to when PCA was applied to the dataset instead of its default sparse feature filter, came out to be like this

TABLE 7.1: Evaluation metrics when clustering was done by native CuckooML

| Evaluation Metric | Native CuckooML | | PCA + CuckooML | |
|---|---|---|---|---|
| | without noise [1] | with noise | without noise | with noise |
| AMI [2]Score | 0.158 | 0.124 | 0.238 | 0.134 |
| ARI [3] | 0.033 | 0.006 | 0.085 | 0.067 |
| Completeness | 0.385 | 0.362 | 0.467 | 0.388 |
| Homogeneity | 0.636 | 0.466 | 0.645 | 0.327 |
| V-measure | 0.480 | 0.407 | 0.542 | 0.355 |

**Conclusion**: All the scores got better, though very minutely, yet PCA is seen to outperform the native feature filter used in CuckooML.

**Effects of optional headers and feature binning technique**

Feature binning and optional headers were added to the feature set, evaluation metrics became something like this

TABLE 7.2: Evaluation metrics when clustering was done with optional headers and feature binning on CuckooML

| Evaluation Metric | Sparse feature filter | | PCA | |
|---|---|---|---|---|
| | without noise | with noise | without noise | with noise |
| AMI Score | 0.459 | 0.342 | 0.482 | 0.320 |
| ARI | 0.652 | 0.437 | 0.653 | 0.405 |
| Completeness | 0.574 | 0.528 | 0.589 | 0.534 |
| Homogeneity | 0.699 | 0.501 | 0.691 | 0.469 |
| V-measure | 0.630 | 0.514 | 0.636 | 0.499 |

**Conclusion**: Adding optional headers to the feature set along with feature binning technique resulted in great improvement in clustering quality. Optional headers

[1]Noise refers to the binaries that could not be clustered or classified
[2]Adjusted Mutual Information
[3]Adjusted Random Index

were added in the feature set, and feature binning was done on file names, static and dynamic library imports.

**Clustering on binary classed data**

Furthermore, clustering was done over the binary labeled data, i.e. the true labels assigned were malware or benign. Then the three feature reduction or extraction techniques were applied to see how well they were performing.

*Note:* The only difference that comes is with respect to evaluation metrics, i.e. the true labels used there would change. Along with that, feature selection algorithms use the binary classes as well to determine the most significant features out of the lot. Following are there results.

Table 7.3: Evaluation metrics using binary class labels on enhanced CuckooML

| Evaluation Metric | Sparse feature filter | | PCA | |
|---|---|---|---|---|
| | without noise | with noise | without noise | with noise |
| AMI Score | 0.197 | 0.173 | 0.213 | 0.185 |
| ARI | 0.465 | 0.317 | 0.499 | 0.329 |
| Completeness | 0.208 | 0.182 | 0.223 | 0.194 |
| Homogeneity | 0.743 | 0.624 | 0.747 | 0.612 |
| V-measure | 0.326 | 0.283 | 0.344 | 0.294 |

Table 7.4: Evaluation metrics using binary class labels on enhanced CuckooML

| Evaluation Metric | RandomForest | | XGBoost | |
|---|---|---|---|---|
| | without noise | with noise | without noise | with noise |
| AMI Score | 0.122 | 0.093 | 0.151 | 0.107 |
| ARI | 0.041 | 0.031 | 0.074 | 0.027 |
| Completeness | 0.137 | 0.104 | 0.163 | 0.116 |
| Homogeneity | 0.699 | 0.343 | 0.760 | 0.374 |
| V-measure | 0.229 | 0.160 | 0.269 | 0.178 |

**Conclusion**: The evaluation scores decreased as compared to the previous class labeling because just the two class clustering does not do justice to the numerous latent classes present into the dataset. Though, the relative increment of scores between PCA and sparse feature filtering further proves the supremacy of PCA feature extraction. Feature selection is clearly over-powered by the other alternatives.

Even though feature selection came out to be inferior to the other techniques, we used it to determine the significance of our additions to the CuckooML module. Moreover, this can act as the proof of concept for employment of feature binning technique. Following are the few significant features as obtained by RandomForest and XGBoost classifiers using binary class labels, with and without CuckooML enhancements.

TABLE 7.5: Feature selection by RandomForest classifier before and after `:file:` feature binning

| without enhancements | with enhancements |
|---|---|
| :simp:msvcrt.dll | :simp:msvcrt.dll |
| :simp:ntdll.dll | :meta:timestamp |
| :sign:.aa | :win:NtCreateFile |
| :pef:lang:neutral | :win:NtOpenKey |
| :simp:count | :dimp:proc:lsass.exe |
| :pef:lang:english | **:file:touch:c:\documents and settings\cuckoo** |
| ':simp:ole32.dll | :pef:lang:neutral |
| ':simp:wininet.dll | :simp:shell32.dll |
| ':pef:lang:chinese | :simp:count |
| ':simp:user32.dll | :simp:version.dll |
| ':win:GetSystemInfo | :pef:lang:korean |
| ':win:RegEnumKeyExA | :win:NtClose |
| ':sign:signed | :dimp:secur32.dll |
| ':win:GetFileAttributesW | :win:GetUserNameA |
| ':dimp:rasman.dll | :dimp:shell32.dll |
| ':win:SetFilePointer | :win:SetFilePointer |
| ':simp:shell32.dll | :win:GetFileSize |
| ':win:WSAStartup | :win:NtReadFile |
| ':win:GetSystemTimeAsFileTime | :win:LdrGetProcedureAddress |
| ':simp:avicap32.dll | **:file:touch:c:\documents and settings\all users** |
| ':pack:Armadillo' | :simp:user32.dll |

**Conclusion**: File binning was done by adding folder-touched features to the dataset. The effect can be clearly seen in table 7.6 - highlighted `:file:` features came into the top 20 most significant ones.

TABLE 7.6: Feature selection by XGBoost classifier before and after applying feature binning to file and static imports features

| without enhancements | with enhancements |
|---|---|
| :meta:size | :simp:count |
| :simp:count | :meta:timestamp |
| :win:NtDeviceIoControlFile | :meta:size |
| :pef:lang:korean | :pef:lang:korean |
| :meta:timestamp | :simp:msvcrt.dll |
| :win:UnhookWindowsHookEx | :win:NtDeviceIoControlFile |
| :simp:msvcrt.dll | :win:GetFileSize |
| :win:NtProtectVirtualMemory | :pef:lang:neutral |
| :pef:lang:neutral | **:simp:count:uncommon**[4] |
| :simp:rpcrt4.dll | **:file:touch:temp-folder** |
| :dimp:mutex:aa\ | :win:SHGetFolderPathW |
| :file:count:failed | :win:NtCreateFile |
| :pack:Armadillo | :win:UnhookWindowsHookEx |
| :win:NtCreateFile | :pack:Armadillo |
| :win:SizeofResource | :dimp:proc:lsass.exe |
| :win:SHGetFolderPathW | :simp:USER32.dll |
| :simp:gdiplus.dll | :win:NtDelayExecution |
| :win:NtOpenMutant | :win:FindResourceW |
| :dimp:mutex:a00 | :win:LoadResource |
| :dimp:mutex:00a | :win:SizeofResource |
| :dimp:proc:lsass.exe | :win:NtQueryDirectoryFile |
| :simp:comctl32.dll | **:file:count:failed** |

**Conclusion**: **:file:touch:temp-folder** and **:simp:count:uncommon** features became significant after they were introduced to the dataset. Furthermore, an intuition can be given behind the presence of **:file:count:failed** in this list, that the failed access on filesystem will be done more by malware than benign binaries, as they would be trying to get access to as much critical information as possible.

[4] `:simp:count:uncommon` is the count of uncommon static import DLLs, where common DLLs consist of the top 10% of the total DLLs seen

## 7.2 Trida - A multi-path execution extension for GDB

*Trida* was developed keeping in mind the simplicity and ease of use this tool should provide. It is supposed to assist in exploring multiple execution paths of x86 linux binaries. Evaluation plan of the tool includes demonstrating the execution of binaries with more than one execution paths, guarded by the arguments given to the file or the environment it is being run into. This section will explore the functionalities provided by *Trida*, demonstrate easy accessibility of challenging features and simplicity of its use. A similar tool does not exist for GDB debugger, hence any performance comparison of Trida would not be possible. This section will also examine the shortcomings of the tool and possible work-around.

### 7.2.1 Scenario 1 : *crackme_xor*

*crackme* binaries are really popular in CTF contests, supposedly testing participants' reverse engineering skills. These challenges can get trickier if the constraints, to reach the desired code section, over the user input are numerous and complicated enough to leave the participants bewildered. At these situations, symbolic engines are usually used to get through the hurdle. Though, in most of the cases, before employing symbolic execution the binary has to be inspected well using a debugger or disassembler in order to know the path that has to be followed (branches to be taken) to reach the target section. Moreover, the input to the symbolic execution has to be manually figured out.

*Trida* was built to come in handy in such situations. As it already runs over GDB, the critical input and the target path can be easily tackled as soon as they are found then and there itself. An illustration will make the significance of this tool more clear. Source code of the *crackme* binary [42] used in this demonstration can be found in Appendix A.

From analyzing the binary, it becomes clear that it takes a filename as its command-line argument, reads its contents and transfers the flow to *check* function sending it the file contents. Now, in order to reach the code block where *PASS*! is being

printed, there are some XOR operations that the file content needs to adhere to. Let us ignite *Trida* to help find the suitable input, below are the key points during the examination of *crackme* binary using *Trida*.

- Once we know that the critical input to the binary is the contents of the file being read, which is non-linearly dependent on command-line argument, **how and where does the symbolization takes place?**

  First of all, this step of symbolizing command-line arguments and anything read from a file can be automated. Let us see the logic behind that -

  - **Command-line argument or function argument**, in general, are present at a specific location (either registers or stack, depending on the architecture), hence finding them provided the debugger is currently at the start of the function frame would be trivial.
  - **File reads** are done using syscalls on user level processes. Syscall return values are also dumped on a definite location, for instance on $RSI$ register for $X86\_64$ architecture. Hence, if the debugger is present at the return of a syscall invocation, finding and symbolizing the contents read from any file would be trivial.

  Automatic symbolization of inputs is kept out of *Trida* because it aims to cater a wider variety of applications. Though, this technique can be used while developing automated multiple path execution of malware binaries.

  For our case, `start <filename>` will make GDB stop at the start of the main function block and `catch syscall 0` will initialize catchpoints at *sys_read* invocations. As soon as the input file is read, one can initialize and symbolize the memory location where the contents are kept, using `syminit` and `syminput <size> <address>` commands.

- Once we have the desired input variable symbolized, **how to decide which jump branch decision is dependent on symbolized input, and how to reach there?**

First of all, relaxing the context of this scenario and talking about multi-path execution with respect to malware, once the critical information had been put under scrutiny (by symbolizing read, socket etc., system call returns and command-line arguments), the succeeding challenge comes in how to traverse the execution path tree as much as possible. To avoid reaching the end of the execution, catchpoints can be initialized at the invocation of *sys_exit* and *halt* instruction. Then, to achieve the objective, a program-state-snapshot stack can be maintained -taking a snapshot at every symbolized jump instruction, further traversing the execution subtree leading from there and then restoring the snapshot present at the top of the stack everytime program hits *sys_exit* catchpoint. This step could be automated as well.

Returning back to our case, we need to find out the symbolized jump instruction that will lead us to the target code block. *Trida* provides with commands like `symuntil <address>` (continue until the given address), `symuntiljump` (continue until the nearest symbolized jump instruction) and `symuntilinst` (continue until the nearest symbolized instruction) to facilitate finding the desired branch instruction. Furthermore, there is another functionality that could prove useful for this case - `noextern` option for the symbolic until commands. `noextern` option tells the *Trida* engine to not stop at any instruction outside the `.text` section. This is useful because our target block and the path leading to it is all located inside the `.text` section. Hence, using `symuntiljump noextern` would directly take the program counter to the branch condition guarding the entrance of the target block.

- Once we have reached the symbolized branch that would take us to the target block, **how do we decide which branch to take?**

  If this decision is yet to be made, snapshots can be taken to ensure this program state is not lost. Commands like `symsnapshot` would have a use for this purpose. Once a snapshot has been taken, any branch can be pursued to see whether it leads towards the target code block or farther away.

If the branch target is clear from the prior inspection of the disassembly code, just like in this case, either of the `symtake` or `symavoid` command can be used to go towards the target block.

- Last catch of this challenge is, **what if the symbolized input size wasn't enough?**

  The file contents might not be of more than or equal to 5 bytes, as this *crackme* binary requires it to be. If the size of the optimal input can't be figured out from the disassembly code inspection, execution and symbolization of the input will have to be done incrementally to find the desired input size.

### 7.2.2 Scenario 2 : *unbreakable_enterprise_product_activation*

*unbreakable_enterprise_product_activation* [A.2] is a GoogleCTF 2016 challenge, that evidently has around 50 constraints to be solved to get the flag, or reach the target block. It is further noticed from the disassembly that this binary requires 51 characters of input, none of which can be a null character. This is practically impossible to do manually, even with *Trida* commands in place, without a bit of scripting or automation. There are two main obstacles in this problem,

- Building further upon the base laid in the last subsection, **what if the binary has unreasonable number of constraints?**

  Once the critical input, that will lead to the target block is identified and symbolized, and the branches to be taken are clear from the disassembly inspection, all we need to do is to take or avoid the branches to go towards the target block. But, like in this case, there may be binaries that have huge number of branches to be taken to accomplish the task. Moreover, we know the topography of the binary, hence are sure about the branches to be taken. Running such a binary through multi-path execution engine would mean a lot of resource wastage and time consumption.

Work-around to this obstacle would be to create a gdb script, utilizing *Trida* commands as many times as required and let it do rest of the job.

- Even with the script, we would not get the correct flag yet. There is one last bit left, **how to make all input characters non-null?**

  There should be a way to supply user-define constraints over the input to *Trida*, while they are being symbolized. Keyword `printable` is meant to serve this purpose to some extent. It will ensure that the input characters are printable characters, i.e. between `0X20` and `0X7E` ASCII values. Hence, the command `syminput` with `printable` keyword would give us our desired input string - the flag.

GDB script to get the flag out of this binary can be found at [19]

### 7.2.3 Shortcomings

*Trida* has inherited several shortcomings from the parent symbolic execution engine and the whole concept behind it. There are two major setbacks which hinders the achievement of absolute code coverage.

**Complex/unsolvable constraints** are non-linear dependencies which are unsolvable even for SMT solver engine, either due to some complex dependency like string manipulation or some external library function call. A suboptimal way to get around this problem is to symbolize, or rather selectively symbolize the variable that is non-linearly dependent on a symbolized input. This will result in inconsistency in the program state as any change in the non-linearly dependent variable would not be propagated to the parent variable. String manipulation has been tackled by emulating the inherent behavior and helping the solver engine to reverse that operation to help it with model building.

**Path explosion** can be due to the presence of unguarded loops or simply many branch instructions. This issue has been tackled in the research community in several ways, few of which are 1) selecting the significant code section and

sticking to it while traversing branches (partially done in this work as well - keyword `noextern`) and 2) handle loops and string operations tactically, i.e. entire loop unfolding is not done and string operations are emulated.

# Chapter 8

# Future Work

Building a robust automated malware analysis platform is not an easy task. Cuckoo Sandbox has gone very far on this objective, but there is still a lot more to do. The integration of any multiple path execution tool will make Cuckoo definitely stronger, yet some concerns would remain. This section will discuss the scope of enhancements that the Cuckoo Sandbox project has with respect to multi-path execution domain, along with its CuckooML module.

**Possible enhancements for CuckooML**

HDBSCAN clustering algorithm responded well to the binning methods employed over file, binary sections, static and dynamic import features. Feature engineering techniques, especially binning, have endless possibilities and are use-case specific. Hence, they can be further explored and applied to other features in the dataset in more innovative ways.

Furthermore, multi-layer machine learning has proved beneficial in several other domains. It has been applied for malware categorization as well and has given promising results, both with respect to accuracy of prediction and lesser resource usage.

Additionally, Cuckoo has the primary motive of analyzing large number of binaries automatically on one click. CuckooML should function with the similar objective.

Hence, adaptive learning techniques or online learning techniques can be employed to involve the newly analyzed binaries into the clustering model.

### Possible enhancements for Trida

Trida, currently, has one major issue inherited from symbolic execution - complex constraint problem. First of all, a possible work-around would be to symbolize the non-linearly or unsolvable-constraint-dependent variable and treat it as a separate input. Though it would create inconsistency and might result in crashing or breaking of program execution, but it would definitely give better code coverage. Furthermore, it could be assisted by emulating common non-linear library functions such as string manipulation.

### Integration of Trida with Cuckoo

Once Trida has sufficiently become complex-constraint compliant, it can be automated and integrated with Cuckoo to automatically explore as many execution paths as possible. There might come several problems while automating it, therein other tracers can be tried that provide better control over execution cycle.

### Multi-path execution - Downsides and Workarounds

Even when the multi-path execution has been integrated with Cuckoo, the scope of work would not end there. The time and resource intensive operation of dynamic analysis will get further burdened with multiple path exploration. This could be solved by limiting the execution paths to be traversed, whose decision (threshold, or even finer selection of paths to be explored) could become an interesting problem.

Furthermore, due to the presence of the inherent problem of path explosion, maximal code coverage with minimal resource utilization should be aimed for. There are several work done, based on heuristics primarily, on this problem but in different context. Efforts can be made to bring this to malware analysis as well.

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

# Appendix A

# Appendix A

## A.1 CuckooML Code Snippets

Changes were made to the `init_cuckooml` function that is responsible for running almost all the task CuckooML is assigned to do. When this work was started, several optimizations were made to the way this analysis was done. This section will brief about all the changes made to the module, dumping the exact code lines just under the briefings.

### A.1.1 Caching of `ML` object

First of all, the capability of caching `ML` class object was added to avoid the labor of reading all the reports and building it. Additionally, a switch was added to turn this feature off whenever required. Below is the code lines added to achieve the aforementioned objective.

```
def init_cuckooml():
        """Initialise CuckooML analysis with default parameters."""
        cfg = Config("cuckooml")

        data_directory = CUCKOO_ROOT + '/' + cfg.cuckooml.
    data_directory
```

```
        use_cache = cfg.cuckooml.use_cache
        SIGNIFICANCE_FRACTION = cfg.cuckooml.significance_fraction

        if use_cache:
                file_cache = cfg.cuckooml.file_cache
                dump_file_path = data_directory + '/' + file_cache


        if use_cache and os.path.exists(dump_file_path):
                with open(dump_file_path, 'rb') as f:
                        ml = pickle.load(f)
        else:
                # Load reports for clustering
                loader = Loader()
                loader.load_binaries(data_directory)

                # Get features dictionaries
                simple_features_dict = loader.get_simple_features()
                features_dict = loader.get_features()
                labels_dict = loader.get_labels()

                print labels_dict

                # Transform them into proper features
                ml = ML()
                ml.load_simple_features(simple_features_dict)
                ml.load_features(features_dict)
                ml.load_labels(labels_dict)

                # TEMP Deleting loader object along with some dicts
                # This is to save excessive use of memory
                # Loader object can't be used to update & save binaries
                del simple_features_dict, features_dict, labels_dict,
    loader

                if use_cache:
                        with open(dump_file_path, 'wb') as f:
                                pickle.dump(ml, f, protocol=pickle.
    HIGHEST_PROTOCOL)

.       .       .
```

### A.1.2 Removal of unnecessary data from memory

When the reports were read while constructing the `ML` object, contents of each and every report file were kept in the memory. This was a poor way of utilizing memory, which limited the number of reports being used for clustering to the number as low as 100. In order to optimize the memory utilization, data read from the report files was deleted as soon as all the required information had been parsed and stored.

Moreover, the code was breaking when the reports lacked any of the expected analysis parts. Hence, some error handling code was added to discard those binaries altogether from the dataset being constructed and prevent unexpected code breakage.

```
def load_binaries(self, directory):
        """Load all binaries' reports from given directory."""
        self.binaries_location = directory + "/"
        for f in os.listdir(directory):
                self.binaries[f] = Instance()
                try:
                        self.binaries[f].load_json(directory+"/"+f, f)
                        self.binaries[f].label_sample()
                        self.binaries[f].extract_features()
                        self.binaries[f].extract_basic_features()

                        # TEMP Deleting instance reports
                        # This is to save excessive use of memory
                        # Instance objects couldn't be used to update &
    save json
                        del self.binaries[f].report
                except AttributeError as e:
                        del self.binaries[f]
                        print >> sys.stderr, "Error in line {}".format(
   sys.exc_info()[-1].tb_lineno)
                        print >> sys.stderr, e
                except ValueError as e:
                        del self.binaries[f]
                        print >> sys.stderr, "Error in line {}".format(
   sys.exc_info()[-1].tb_lineno)
                        print >> sys.stderr, e
```

### A.1.3 Addition of optional headers and feature binning techniques

The code snippet shown below includes the changes done in order to add optional headers to the feature set and feature binning techniques over files, sections, static and dynamic library import features.

```
def extract_features(self, features, include_API_calls=False, \
                                        include_API_calls_count=False):
        """Extract features form an external object into pandas data
    frame."""
        # Preprocessing for feature binning
        simp_counter = Counter()
        dimp_counter = Counter()
        section_counter = Counter()
        # mutex_counter = Counter()
        for i in features:
                simp_counter += Counter(features[i]["static_imports"].
    keys())
                dimp_counter += Counter(features[i]["dynamic_imports"])
                section_counter += Counter(features[i]["section_attrs"
    ].keys())

        significant_count = int(SIGNIFICANCE_FRACTION*len(features))
        simp_common = zip(*simp_counter.most_common(significant_count))
    [0]
        dimp_common = zip(*dimp_counter.most_common(significant_count))
    [0]
        section_common = zip(*section_counter.most_common(10))[0]
.       .       .
.       .       .
.       .       .

        for s in features[i]["section_attrs"].keys():
                if s in section_common:
                        my_features[i][":simp:section:"+s] = 1
                else:
                        my_features[i][":simp:section:count:uncommon"]
    = \
```

```
                                my_features[i].get(":simp:section:
uncommon", 0) + 1
        # Section entropy statistics
        entropies = features[i]["section_attrs"].values()
        my_features[i][":simp:section:entropy:min"] = min(entropies)
        my_features[i][":simp:section:entropy:max"] = max(entropies)
        my_features[i][":simp:section:entropy:mean"] = statistics.mean(
entropies)
        my_features[i][":simp:section:entropy:var"] = statistics.
variance(entropies)
        my_features[i][":simp:section:entropy:median"] = statistics.
median(entropies)
        my_features[i][":simp:section:entropy:stdev"] = statistics.
stdev(entropies)
        my_features[i][":simp:section:entropy:mode"] = max(set(
entropies), key=entropies.count)

        # Vectorize optional headers
         for j in features[i]["optional_headers"].keys():
                 my_features[i][":pef:oph:"+j] = features[i]["
optional_headers"][j]

        # Categorise static imports

        my_features[i][":simp:count"] = \
                features[i]["static_imports"]["count"]
        static_imports_dlls = features[i]["static_imports"].keys()
        static_imports_dlls.remove("count")
        # Count static imports
        my_features[i][":count:simp"] = len(static_imports_dlls)
        for j in static_imports_dlls:
                if j in simp_common:
                        my_features[i][":simp:" + j] = 1
                else:
                        my_features[i][":simp:count:uncommon"] = \
                                my_features[i].get(":simp:count:
uncommon", 0) + 1

                if include_API_calls:
                        for k in features[i]["static_imports"][j]:
```

```
                                        my_features[i][":simp:" + j + ":" + k]
    = 1
                # Count static imports API calls
                if include_API_calls_count:
                        my_features[i][":count:simp:" + j] = \
                                len(features[i]["static_imports"][j])
.
.
.
```

### A.1.4 Addition of feature extraction/selection techniques

Two functions were added to the `ML` class of the CuckooML module, namely `filter_dataset_pca` - to conduct PCA feature extraction, and `feature_selection` - to conduct feature selection using any suitable classifier that is supplied as the function argument. `draw_box_plot` function was added to help visualize the significance of specific set of features on the classification of the binaries.

```
def filter_dataset_pca(self, dataset=None, variance_coverage=0.94):
        """Feature reduction using PCA"""
        if dataset is None:
                dataset = self.features.copy()

        scale(dataset)
        pca = PCA().fit(dataset)

        variances = pca.explained_variance_ratio_.cumsum()
        num_cols = next(i for i in range(len(variances)) if variances[i
    ] > variance_coverage)
        return pd.DataFrame(data=pca.transform(dataset)[:,:num_cols])


def feature_selection(self, dataset=None, labels=None, k=None,
    classifier=RandomForestClassifier):
        """Feature reduction using PCA"""
        if dataset is None:
                dataset = self.features.copy()
```

```
        if labels is None:
                labels = self.labels.copy()

        if k is None:
                k = dataset.shape[0] * SIGNIFICANCE_FRACTION

        X = dataset.as_Metric()
        Y = labels.values.ravel()
        scale(X)

        model = classifier()
        model.fit(X, Y)

        print zip(dataset.columns, model.feature_importances_)

        selected_indices = model.feature_importances_.argsort()[::-1][:
    k]
        return pd.DataFrame(data=X[:, selected_indices], columns=
    dataset.columns[selected_indices])

def draw_box_plot(self, dataset=None, labels=None, columns=[]):
        if dataset is None:
                dataset = self.features.copy()
        if labels is None:
                labels = self.labels.copy()

        for column in columns:
                df = pd.DataFrame(dataset[column]).join(labels, how='
    inner')
                df['label'].fillna('none', inplace=True)
                df[column].fillna(0, inplace=True)

                df.boxplot(by='label')
                plt.show()
```

## A.2 Trida Code Snippets

### A.2.1 Trida Source Code

*Trida* is a GDB extension built upon the forked copy of *Peda*. The complete source code can be found here - https://github.com/vineetpurswani/peda-with-triton.

### A.2.2 Documentation of Trida code

#### A.2.2.1 Trida Class

- $\_\_init\_\_()$ - Initialization step
- $simulate(stop = None, stop\_on\_sj = False, stop\_on\_si = False, no\_extern = False)$ - Processing instructions. This is done in several ways depending on the function parameters,
  1. *stop* parameter is to provide an address where the processing of instructions should stop.
  2. *stop_on_sj* is a flag asking whether execution should pause if a symbolized jump is encountered.
  3. *stop_on_si* is a flag asking whether execution should pause if a symbolized instruction is encountered.
  4. *no_extern* makes sure execution does not stop in any external function if set true.
- $build\_jmp\_constraint(pc = None, take = True)$ - Build branch constraint, if *pc* points to a jump instruction. *take* boolean decides whether the constraint to be built should correspond to branch taken or not.
- $process\_constraint(cstr)$ - Process *cstr* constraint and update the inputs with the required changes. This change should be propagated to all the dependent memory and register values as well. One way, which is used in this extension, is to re-execute the program until current *pc* with the altered inputs.

#### A.2.2.2 PEDA Commands

- `syminit` - Initialize `Trida` object
- `symuntil <addr>` - Resume execution until the given address, both in Triton and GDB.
- `symuntiljump noextern` - Resume execution until symbolized jump (of function present in `text` section only, if noextern is present)
- `symuntilinst noextern` - Resume execution until symbolized instruction (of function present in `text` section only, if noextern is present)
- `symtake` - Jump to target of the current symbolized jump instruction, and update the Triton state according to the change in input values
- `symavoid` - Avoid the current symbolized jump instruction, and update the Triton state according to the change in input values
- `syminput <input_size> <input_addr> printable` - Symbolize the given memory upto `input_size` bytes. `printable` is an optional keyword that is used to tell Trida to put printable characters constraint on the given input variables.
- `symsync` - Update GDB input memory with Triton input values.
- `symreset` - Reset Triton input memory with corresponding values from GDB.
- `symsnapshot save\restore` - Take or restore snapshot of GDB + Triton states combined.

### A.2.3 Illustration Code

Below is the source code of binary used in scenario 1 of Trida tool evaluation.

```
#include <stdio.h>
#include <stdlib.h>

/* Global serial */
```

```
char *serial = "\x31\x3e\x3d\x26\x31";

int check(char *ptr)
{
        int i = 0;
        while (i < 5){
        if (((ptr[i] - 1) ^ 0x55) != serial[i])
                return 1;
        i++;
        }
        return 0;
}

int main(int argc, char **argv) {
        FILE *ptr_file;

        if (argc != 2) {
                printf("Usage %s <passcode_file>\n", argv[0]);
                exit(0);
        }
        ptr_file = fopen(argv[1], "r");
        if (!ptr_file) {
                printf("Given passcode file doesn't exist\n");
                exit(0);
        }

        char passcode[100];
        fscanf(ptr_file, "%s", passcode);

        if (check(passcode)) {
                printf("FAIL!\n");
        }
        else {
                printf("PASS!\n");
        }
        return 0;
}
```

LISTING A.1: crackme_xor.c

The binary used in scenario 2 of Trida tool development can be found at - https://github.com/ctfs/write-ups-2016/tree/master/google-ctf-2016/reverse/unbreakable-er

Below is the GDB script written to solve the CTF challenge used in scenario 2.

```
source ~/repos/peda/peda.py
start aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
until *0x4005bd
syminit
syminput 51 0x6042c0 printable

(
.
.
symuntiljump noextern
symavoid
.
.
) 52 times

syminput
```

Listing A.2: GDB Script to solve Scenario 2 of Chapter 6

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

# Bibliography


[1] Wikipedia. Mirai (malware). https://en.wikipedia.org/wiki/Mirai_(malware), 2017.

[2] Wikipedia. Wannacry ransomware attack. https://en.wikipedia.org/wiki/WannaCry_ransomware_attack, 2017.

[3] Telegraph. Top 10 worst computer viruses. http://www.telegraph.co.uk/technology/5012057/Top-10-worst-computer-viruses-of-all-time.html, 2009.

[4] Wikipedia. Melissa virus. https://en.wikipedia.org/wiki/Melissa_(computer_virus), 2017.

[5] Samuel T King and Peter M Chen. Subvirt: Implementing malware with virtual machines. In *Security and Privacy, 2006 IEEE Symposium on*, pages 14–pp. IEEE, 2006.

[6] angelica cajimat. Top 10 worst computer viruses. http://www.cknow.com/cms/vtutor/stealth-viruses-and-rootkits.html, 2011.

[7] Wikipedia. Sony bmg copy protection rootkit sandal. https://en.wikipedia.org/wiki/Sony_BMG_copy_protection_rootkit_scandal, 2017.

[8] Techworm. Lenovo backdoor. https://www.techworm.net/2015/08/lenovo-pcs-and-laptops-seem-to-have-a-bios-level-backdoor.html, 2015.

[9] Karl Thomas. Nine bad botnets and the damage they did. https://www.welivesecurity.com/2015/02/25/nine-bad-botnets-damage/, 2011.

[10] Wikipedia. Mirai botnet. https://en.wikipedia.org/wiki/Mirai_(malware), 2017.

[11] TheFreeDictionary. Top 10 worst computer worms of all time. http://encyclopedia2.thefreedictionary.com/Top+10+Worst+Computer+Worms+of+All+Time, 2011.

[12] Wikipedia. Mirai botnet. https://en.wikipedia.org/wiki/Code_Red_(computer_worm), 2017.

[13] Margaret Rouse. Top 10 spyware threats. http://whatis.techtarget.com/definition/Top-10-Spyware-Threats, 2011.

[14] Wikipedia. Petya malware. https://en.wikipedia.org/wiki/Petya_(malware), 2017.

[15] Wikipedia. Intrusion detection system. https://en.wikipedia.org/wiki/Intrusion_detection_system, 2017.

[16] Wikipedia. Firewall (computing). https://en.wikipedia.org/wiki/Firewall_(computing), 2017.

[17] Hex-rays. Idapro. https://www.hex-rays.com/products/ida/, 2017.

[18] ollydbg.de. Ollydbg. http://www.ollydbg.de/, 2017.

[19] GNU. Gdb. https://www.gnu.org/s/gdb/, 2017.

[20] nmap.org. Nmap. https://nmap.org/, 2017.

[21] wireshark.org. Wireshark. https://www.wireshark.org/, 2017.

[22] Victor Alvarez. Yara - the pattern matching swiss knife for malware researchers. virustotal.github.io/yara/, 2017.

[23] Jonathan Salwan. Triton - a dba framework. https://triton.quarkslab.com/, 2016.

[24] cuckoosandbox.org. Automated malware analysis - cuckoo sandbox. https://cuckoosandbox.org, 2016.

[25] Matthew G Schultz, Eleazar Eskin, F Zadok, and Salvatore J Stolfo. Data mining methods for detection of new malicious executables. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 38–49. IEEE, 2001.

[26] Jeremy Z Kolter and Marcus A Maloof. Learning to detect malicious executables in the wild. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 470–478. ACM, 2004.

[27] Lakshmanan Nataraj, S Karthikeyan, Gregoire Jacob, and BS Manjunath. Malware images: visualization and automatic classification. In *Proceedings of the 8th international symposium on visualization for cyber security*, page 4. ACM, 2011.

[28] Ronghua Tian, Lynn Margaret Batten, and SC Versteeg. Function length as a tool for malware classification. In *Malicious and Unwanted Software, 2008. MALWARE 2008. 3rd International Conference on*, pages 69–76. IEEE, 2008.

[29] Igor Santos, Javier Nieves, and Pablo Garcia Bringas. Semi-supervised learning for unknown malware detection. In *DCAI*, pages 415–422. Springer, 2011.

[30] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668, 2011.

[31] Mohamad Fadli Zolkipli and Aman Jantan. An approach for malware behavior identification and classification. In *Computer Research and Development (ICCRD), 2011 3rd International Conference on*, volume 1, pages 191–194. IEEE, 2011.

[32] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *NDSS*, volume 9, pages 8–11, 2009.

[33] Ronghua Tian, Rafiqul Islam, Lynn Batten, and Steve Versteeg. Differentiating malware from cleanware using behavioural analysis. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pages 23–30. IEEE, 2010.

[34] Younghee Park, Douglas Reeves, Vikram Mulukutla, and Balaji Sundaravel. Fast malware classification by automated behavioral graph matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, page 45. ACM, 2010.

[35] Saeed Nari and Ali A Ghorbani. Automated malware classification based on network behavior. In *Computing, Networking and Communications (ICNC), 2013 International Conference on*, pages 642–647. IEEE, 2013.

[36] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 231–245. IEEE, 2007.

[37] Igor Santos, Jaime Devesa, Felix Brezo, Javier Nieves, and Pablo Garcia Bringas. Opem: A static-dynamic approach for machine-learning-based malware detection. In *International Joint Conference CISIS12-ICEUTE 12-SOCO 12 Special Sessions*, pages 271–280. Springer, 2013.

[38] Blake Anderson, Curtis Storlie, and Terran Lane. Improving malware classification: bridging the static/dynamic gap. In *Proceedings of the 5th ACM workshop on Security and artificial intelligence*, pages 3–14. ACM, 2012.

[39] Xiaolei Wang, Yuexiang Yang, Yingzhi Zeng, Chuan Tang, Jiangyong Shi, and Kele Xu. A novel hybrid mobile malware detection system integrating anomaly detection with misuse detection. In *Proceedings of the 6th International Workshop on Mobile Cloud Computing and Services*, pages 15–22. ACM, 2015.

[40] Heng Yin Juan Caballero et al. Dawn Song, David Brumley. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, December 2008.

[41] BitBlaze: Binary analysis for computer security. http://bitblaze.cs.berkeley.edu/.

[42] virustotal.com. Virustotal. https://www.virustotal.com/, 2017.

[43] wiktionary.org. Shannon entropy. https://en.wiktionary.org/wiki/Shannon_entropy, 2017.

[44] Robert Lyda and James Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, 5(2), 2007.

[45] Karthik Raman et al. Selecting features to classify malware. *InfoSec Southwest*, 2012, 2012.

[46] Scikit. Clustering. http://scikit-learn.org/stable/modules/clustering.html, 2016.

[47] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[48] Ricardo JGB Campello, Davoud Moulavi, Arthur Zimek, and Jörg Sander. Hierarchical density estimates for data clustering, visualization, and outlier detection. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 10(1):5, 2015.

[49] Wikipedia. Principal component analysis. https://en.wikipedia.org/wiki/Principal_component_analysis, 2017.

[50] Wikipedia. Random forest. https://en.wikipedia.org/wiki/Random_forest, 2017.

[51] DMLC. Xgboost. https://xgboost.readthedocs.io/, 2017.

[52] Kacper Sokol. Cuckooml. https://honeynet.github.io/cuckooml/, 2016.

[53] Steve Astels Leland McInnes, John Healy. Hdbscan. http://hdbscan.readthedocs.io, 2017.

[54] Daan Pepjin. As the world moves toward ubiquitous connectivity, it is time to ramp up trust. http://www.iotevolutionworld.com/m2m/articles/432897-as-world-moves-toward-ubiquitous-connectivity-it-time.htm, 2017.