Detecting and Modeling Polymorphic Shellcode

A thesis

In

The Departement

Of

Concordia Institute of Information Systems Engineering

(CIISE)

Presented in Partial Fulfillment of the Requirements

For the Degree of Master Of Applied Science In Information Systems

Security

at

Concordia University

Montreal, Quebec, Canada

November 2010

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: Omar Nbou

Entitled: Detecting and Modeling Polymorphic Shellcode

and submitted in partial fulfillment of the requirements for the degree of

Master Of Applied Science In Information Systems Security

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Jamal Bentahar	Chair
Dr. Amr Youssef	Examiner
Dr. Hamou-Lhadj	Examiner
Dr. Debbabi	Supervisor

Approved by ____________________
Chair of Department or Graduate Program Director

____
Dean of Faculty

Date 09/05/2011

Abstract

Detecting and Modeling Polymorphic Shellcode

Omar Nbou

In this thesis, we address the problem of modeling and detecting polymorphic engines shellcode. By polymorphic engines, we mean programs having the ability to transform any piece of malware into many instances consisting of different code but having the same functionality as the original malware. Typically, polymorphic engines work by encrypting the target malware using various encryption techniques and providing a decryption module in order to execute the newly encrypted instance. Moreover, those engines have the ability to mutate their decryption routine making them unique from one instance to another and hard to detect. Our analysis focuses on polymorphic shellcode, which is shellcode that uses a polymorphic engine to mutate while keeping the original function of the code the same. We propose a new concept of signatures, shape signatures, which cope with the highly mutated nature of those engines. Those signatures try to identify the constant part as well as the mutated part of the deciphering routines. This combination is able to cope with the highly mutated nature of those engines in a much more efficient way compared to traditional signatures used in most intrusion detection systems. The second part of the thesis aims at modeling those polymorphic engines by showing that they exhibit common characteristics. The analysis of bit positions and byte composition of decoders shows us that polymorphic decoders exhibit a specific byte composition and can be mapped.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivations

Remote exploitation remains one of the most important threats for information systems security. The growing number of software programs developed without taking into consideration security aspects, makes code injection attacks a very appealing way for attackers to compromise systems. A code injection attack can be described as the exploitation of a computer bug caused by the processing of invalid data. The number of vulnerabilities found in computer programs has become so important that computer industry and researchers shifted their attention to detecting exploits rather than improving the software programs. As a consequence, tools such as Network Intrusion Detection Systems (NIDS) and antiviruses are used to ensure the detection of such abnormalities or exploits. Huge databases are kept up-to-date for the sake of fingerprinting or generating signatures that can help in identifying such pieces of malware. The challenge for an attacker is to find and exploit vulnerabilities as quickly as possible before a signature could be generated. However, attackers have recently discovered very efficient techniques to evade NIDS and anti-viruses by developing polymorphic engines. As such, a traditional structure for a buffer overflow based exploit, consisting of a No Operation Performed section (NOP), a shellcode and a return address, moves into a new structure consisting of:

- Obfuscated NOP section using multi-argument no-operation instructions
- Decoder or decryption routine (subject to mutation from one instance to another)
- Encrypted shellcode (where the encryption keys are modified from one instance to another)
- Return address

Thus, known signatures or piece of malware can be easily concealed through the use of a powerful polymorphic engine. Many approaches are taken when it comes to detect polymorphic engines. Some of them focus on detecting the NOP section whereas others draw attention to detecting the return address in the payload. Models using static binary analysis and emulation techniques are also used as detection mechanisms as well as signature-based models. The results are the same for all those detection systems: polymorphic engines are able to evade them. Hackers are improving their polymorphic algorithms and making their engines more and more powerful. However, we believe that there is a strong need in improving the current signature-based detection into models that cope with the new threats resulting from the use of polymorphic engines. So far, most signature-based detection models fail against polymorphic engines because they do not take into into account the polymorphic nature or are not able to express it in their detection mechanism. We strongly believe that the issue for signature-based detection is to implement a signature model that is able to cope or express that polymorphic nature efficiently. If this condition is fullfilled, signature-based detection could become very efficient against such engines and could bring interesting results. In this respect, improvement of detection mechanisms against such engines is becoming challenging and important in the area of malware detection.

1.2 Problem Statement

A polymorphic engine has the ability to generate many instances of the same piece of malware with a high degree of uniqueness. In other words, all the generated instances look very different from each other but yet achieve the same functionalities. As a consequence, the use of traditional signature becomes completely useless against such polymorphic techniques. Song et al. [3] show that in order to cope with such polymorphic behavior, traditional signature detection system need to model a space in the order of $O(2^{8n})$ sequences. They extended the analysis by developing two metrics: variation strength and propagation strength. The former measures the degree of exploration of the byte spectrum, which simply means measuring the degree of variety in the bytes used from one instance to another. The latter measures the degree of uniqueness in the instances generated from one polymorphic engine. Their results show that most powerful polymorphic engines exhibit high degrees of variation and propagation strength. As such, we believe that there is a need to improve the concept of signature in order to better catch the highly mutating nature of those engines. Obviously, one can conclude that traditional signature detection models (with bit per bit pattern-matching) are completely useless.

1.3 Objectives

The main objective of the thesis is to propose a new approach to a more efficient detection and modeling of polymorphic engines. More specifically, we aim at achieving the followings:

- Study the state of the art proposals in the area of polymorphic shellcode detection and modeling
- Propose a new model of signature to better cope with polymorphism
- Elaborate a model based on evolutionary algorithms to extract those signatures
- Validate the proposed approach through experimentation
- Model polymorphic engines by extracting common exhibited characteristics:
  - Study the bits positions of the newly extracted signatures to show the possibility of effectively mapping the decoders by locating the areas subject to high mutation and areas corresponding to the extracted signatures
  - Propose experimental approaches in order to validate the conclusions drawn from studying bit positions
  - Analyze the bytes composition to show that each polymorphic engine exhibits a specific composition that uniquely characterizes it
  - Provide experimental results showing repeated patterns in terms of byte composition for each engine

1.4 Contributions

The main contributions of the thesis can be summarized as follows:

- Comparative study of the state of the art techniques in terms of polymorphic shellcode detection. We identify five main families of detection mechanisms: no-operational instruction detection, return address detection, static binary code analysis, emulation-based detection and signature-based detection.

- Proposal of a new model of signatures that better deals with polymorphism techniques. We believe the approach that consists of identifying invariant elements of polymorphic decoders as well as mutated part is original. In fact, signature-based detection techniques could be enhanced and should not be exluded.

- Different modeling approaches for polymorphic engines. By different, we mean characterizing each engine through its byte composition as well as mapping its decoders. The mapping consists of showing that we can effectively predict the areas that are subject to high mutations as well as areas that correspond to the newly generated signatures.

- Providing new elements about polymorphic engines' behavior, such as byte composition characteristics and mapping of their decoders that could constitute new leads for modeling those engines.

1.5 Approach

We revise the notion of signature by taking into account the mutation points as well as the invariant codes. This will lead us to what we call shape signature. The general approach of the thesis is to develop first a genetic algorithm [16] engine that extracts what we call shape signatures. By shape signatures, we mean signatures that locate constant parts as well as mutated parts of polymorphic decoders. Then, we analyze the bit position of those signatures in order to show that we can map the general shape of polymorphic decoders. By mapping, we mean that we can consistently predict the areas of high mutation as well as the areas containing the shape signatures. The aim is to show that the positions of those signatures do not occur along the whole decoder. They rather shift in a predefined area of the decoder. With respect to that, we are able to effectively locate areas along the decoder that are subject to high mutation as well as area that corresponds to constant patterns. Finally, we analyze the byte occurrences of those polymorphic engines and show that the same bytes keep repeating themselves. By identifying three main elements, which are the NOP elements, the byte spectrum, and the other bytes, we identify those polymorphic decoders in terms of their byte composition by extracting rules of compositions that characterize them. The aim is to show that we can differentiate those polymorphic decoders, in terms of composition, from a regular NOP section or any random sequence of bytes.

1.6 Research Issues and Results

The main issue, when it comes to signature detection, is to cope with the ability of polymorphic engines to generate polymorphic decoders that require an order of $O(2^{8n})$

sequences to catch all polymorphic behaviors with n being the decoder's length. We know from [10, 11, 12, 13] that those engines leave artefacts in their mutation processes due, mainly, to consistency constraints. By consistency constraint, we mean that no matter how random the engine might look, it must be consistent by creating polymorphic decoders that perform their job reliably. From a hacker's perspective, the challenge is to avoid having a too long sequence of those invariant codes that could constitute a reliable signature. Having those two elements in mind, the issue is to come with a model that is able to express them. It is complex and challenging because the two elements are contradictory and yet they are related. The methodology that we follow is to extract moulds that better characterize the polymorphic decoder. Instead of trying to identify all the possible instances that a polymorphic engine could generate, we look for a higher-level or more abstract model that can better define all those possible instances. As a consequence, we come with a new model of signatures called shape signatures that is able to detect constant parts, due to consistency constraints, and mutated parts due to polymorphism. In this regard, we make use of a genetic algorithm that is able to extract shape signatures from large sets of polymorphic decoders. The obtained results are very promising in the sense that the model of shape signatures holds.

The next step is to find a way to model polymorphic shellcode generated by polymorphic engines. The modeling of such engines is challenging because similar work is very rare. Thus having a reliable starting point that we could build on or improve from is key. The shape signatures could constitute the foundation from which we can elaborate the modeling of polymorphic shellcode. As such, we study the bits' position of the newly extracted shape signatures and come to the conclusion that polymorphic decoders could be mapped effectively. By doing so, we mean that we could predict where the shape signatures appear, as well as the mutated parts, along the decoder's space. Our modeling is extended by analyzing the byte composition of polymorphic decoders. The motivation is to find a way to characterize those decoders by analyzing their byte spectrums. Our results show that polymorphic decoders exhibit a constant pattern in their composition allowing us to recognize them uniquely.

1.7 Organization

In Chapter 2, we present a detailed description of related work on detecting and modeling polymorphic engines. We propose a classification of such approaches in order to provide the reader with a better overview of how researchers detect and model polymorphic engines. We also present some of the techniques used by the most powerful polymorphic engines known until today such as ADMmutate [14], Shikata Ga Nai [15] and Tapion [27]. The aim is to show how hackers improve their evasion techniques. In Chapter 3, we provide a detailed description on the theory of genetic algorithms. The description is purely theoretical and will help to familiarize the reader with such evolutionary technique. In Chapter 4, we define the new concept of shape signature and explain how it relates to genetic algorithms. Therein, we describe in details some of the inner-workings of the engine, then discuss and validate the obtained shape signatures. Chapter 5 explains how decoders generated from polymorphic engines can be mapped. By mapping, we mean that we can consistently predict the areas of high mutation as well as the areas containing the shape signatures. Chapter 6 is dedicated to composition analysis of decoders generated from those engines. By composition, we mean examining the byte spectrum produced by polymorphic decoders. Finally, some concluding remarks as well as a discussion of future work are presented in Chapter 7.

Chapter 2

Background

2.1 Related Work

We find various approaches when it comes to analyzing polymorphic engines, more specifically polymorphic shellcode in the context of a buffer overflow based exploit. The proposed mechanisms for detecting or modeling such engines use a wide variety of techniques that are often unrelated. However, we believe that a classification can better summarize how the problem is approached. In this respect, we have been able to identify five families or categories:

- NOP section detection: This basically includes techniques for detecting the no-operation instructions that precede the piece of malware
- Return address detection: It can be summarized as attempts to detect the possible existence of a return address and thus of an exploit

- Static binary code analysis: It includes techniques that consist of converting network data into binaries and then analyze the flow of executed instructions
- Emulation-based detection technique: In this category, network data is transformed into a binary format that will be executed in a safe environment. During the execution, attempts are made in order to detect polymorphic behavior
- Signature-based detection: This denotes the technique adopted in this thesis. It involves searching for known malicious patterns in executable code

We aim at describing the major research proposals for each category to give the reader a better insight of the state of the art in the field. For each description, we detail the approaches and give their limitations with respect to polymorphic engines.

2.1.1 No-Operational Instruction Detection

Buffer overflow based exploits usually consist of three major sections: the NOP section, the shellcode section, and the return address. Even though polymorphic shellcode based exploits present a slightly different structure (with the addition of the decoder routine), the no-operation section is still present. As a consequence, the detection focuses on identifying the NOP section since polymorphic techniques are so difficult to model and predict. [17, 18] note in their analysis that the rate of detecting the no-operation section is higher than the rate of trying to find a signature or analyzing the flow of execution of such malware. Their results are promising since the NOP section consists of mainly repeating the same op-code for a certain number of times. Consequently, when having the latter pattern, we can effectively detect a NOP section. However, the release of ADMmutate polymorphic engine [14] makes this approach completely obsolete since it is found that the number of x86 no-operation instructions do not consist of only three or four instructions but to more than sixty instructions. Thus any combination of these instructions could constitute a NOP section without having to repeat the same instruction. Consequently, the intrusion detection systems NGSEC [19] and Snort [20] released functionalities incorporating such instructions, expanding their no-operation instruction library. Based on the new library of no-operation instructions, they elaborated algorithms to detect the occurrence of such instructions and thus the NOP section. Initially the results were promising until the discovery of multi-argument no-operation instructions [21]. Basically, it is a no-operation instruction that takes arguments. The arguments could be any piece of byte, even though they do not belong to the NOP library. Having regular instructions inserted between sequences of no operation instructions, the NOP section is completely obfuscated. The detection of these instructions fail since it becomes impossible to identify a sequence of contiguous NOP instructions. Consequently, the no-operational instruction detection becomes impossible when dealing with multi-argument NOPs.

2.1.2 Return Address Detection

The second category deals with detecting the return address part for buffer overflow based exploits. The idea is interesting in the sense that it assumes that the range of addresses for a process, when executed, lies in specific areas of the memory. As such, Buttercap [1] aims at performing such detection technique by identifying the range of possible return memory addresses for existing buffer-overflow vulnerabilities. However, the method has many limitations:

- It relies heavily on the target operating system. In fact, the techniques for completing an exploit differ from one operating system to another, thus, implementing such detection mechanism is challenging.

- In some cases, the completion of an exploit does not require the use of a return address as it is the case for Windows-based systems. SEH based exploits [22] are a perfect example.[1]

- All new versions of Linux, Windows and Mac respective operating systems come with Address Space Layout Randomization[2] (ASLR) [23]. As a consequence, new ranges of addresses have to be considered and explored making the detection prone to false positives.

- ASLR is created in order to make the guess of the return address in the presence of vulnerable programs nearly impossible. However, new exploitation techniques beating the ASLR exist. Thus, the range of address to consider is even bigger and hard to predict leading to an increase of both false positives and false negatives in the detection.

We come to the conclusion that such an approach is very hard to implement, consequently, its efficiency is highly questionable.

[1]SEH: Structured Exception Handlers are exceptions that a program cannot handle itself, then control is passed to a SEH address that has code that can be used to show a dialog box explaining that the program has crashed

[2]ASLR: technique which involves randomly arranging the positions of key data areas such as the base of the executable, position of libraries, heap, and stack, in a process's address space

2.1.3 Static Binary Code Analysis

Before addressing the topic of static binary code analysis, it is important to mention that we do not intend to question its efficiency. Static binary code analysis is indeed a powerful method for analyzing binaries. Having seen the inefficiency of previous approaches (No-operational instruction detection and return address detection), some initiatives considered treating polymorphic shellcode as executables or binaries. As a consequence, static binary code analysis is considered as a reliable technique to perform such task. In such approach, disassembling techniques are used in order to transform network data into binary code. The control flow of the code is then analyzed statically in order to detect patterns or features that could characterize polymorphic shellcode behavior. By static, we mean that the translated binary is not executed. The approach is restricted to analyzing the disassembled binary. Consequently Payer et al. [6], Chinchani et al. [25] and Kruegel et al, [26] developed interesting mechanisms performing such tasks. However, the process of translation is itself very challenging. In fact, most of the time, the conversion from network data into binaries is prone to errors making the conclusions that could be drawn from such detection techniques highly questionable. Moreover, powerful polymorphic engines implement techniques that resist static analysis such as self-modifying code. In fact, Polychronakis et al. [4] give a complete description on how static binary code analysis could be easily evaded.

2.1.4 Emulation-Based Detection

It is important to mention that we do not challenge the efficiency of emulation-based techniques but rather question their efficiency in detecting polymorphic shellcode. In order to defeat static analysis evasion techniques of polymorphic shellcode, emulation-based detection is proposed as an enhancement of the previous approach. Polychronakis et al. [4] are the first to propose such detection. Their approach differs from the previous ones in the sense that they do not stop to the translation from network data to binaries and the analysis of the disassembled binary. In addition. they execute the binary resulting from the translation in a safe environment and try to identify polymorphic shellcode behavior from the execution. Such approach faces many problems in the context of polymorphic shellcode detection. The process of emulation itself presents many problems. The most obvious one is the risk of facing infinite loops when trying to emulate the execution of the newly translated binaries. Most of the time, the newly translated binaries do not correspond, in reality, to executable code leading to many bugs and inconsistencies in the execution. Often the inconsistencies result in infinite loops during the execution. Moreover, there is real difficulty in identifying where the polymorphic shellcode starts and where it stops. As a consequence, we can question the reliability of conclusions that can be drawn from such an approach. The second problem faced with this detection is the ability for polymorphic engines to evade it. We mention just some few evasion techniques to show the flaws that can be faced with emulation detection. As we have mentioned previously, there is a risk with emulation to face infinite loops. One solution proposed by Lanjua et al. [7] and Zang et al. [28] consists in imposing a threshold limiting the number of executed instructions so that infinite loops are avoided. Consequently, the implementation of endless but actually finite loop in the shellcode can evade such detection. Thus, before the shellcode is detected, the execution ends, causing a false negative. Moreover, a way of detecting polymorphic shellcode behavior, using emulation, is the decoding process. When the decoding routine is launched, the decryption process is done at regular time interval revealing a decoding behavior that could be detected. Tapion [27], as an example, possesses a decryption mechanism involving a random CPU time usage. Moreover, most powerful polymorphic engines have techniques that involve executing the shellcode in a random fashion disturbing the flow of execution. Thus, possible analysis of shellcode behavior using emulation is highly questionable. For these reasons, we believe that emulation based detection is not a reliable solution in order to deal with polymorphic shellcode.

2.1.5 Signature-Based Detection

This category is close to the approach adopted in this thesis. In the following, we present and discuss some related research proposals. In this setting, it is to mention the result proposed by Song et al. [3] on the impossibility of detecting and modeling polymorphic shellcode using conventional signatures. Their work is important in the sense that it is the most successful attempt in understanding and quantifying the strength of polymorphic shellcode. In this paper, the authors present a quantitative analysis of the strengths and limitations of polymorphic shellcode. They also consider the impact of this analysis on the current practices in intrusion detection. Their analysis provides a new and useful way to understand the limitations of the current generation of signature-based techniques. Consequently, the authors developed two metrics: variation strength and propagation strength in order to quantify the strength of a polymorphic engine. The first metric measures the degree of exploration of the byte spectrum, which simply means measuring the degree of variety in the bytes used from one instance to another. The second metric measures the degree of uniqueness in the instances generated from one polymorphic engine. The authors state that most powerful polymorphic engines exhibit high degrees of variation and propagation strength. Based on that, they come to the conclusion that current generation of signature-based techniques are limited and inadequate for detecting polymorphic shellcode. As stated by Song et al. in [3], we need to model a space in the order of $O(2^{8n})$ sequences to cope with polymorphism. In fact, better models of signatures should be proposed in order to deal with the high degree of mutation of polymorphic shellcode. Our motivations are based on the fact that even when they mutate, polymorphic engines leave artefact making them potentially subject to detection. Moreover, the created polymorphic decoders must be consistent by performing their task which is decoding the encrypted shellcode. Consequently, any technique or attempt to make the decoder look highly random or polymorphic is greatly reduced by this consistency constraint. Finally, the mutation techniques themselves are subject to detection. Based on that, a model of signature that takes into consideration the possible mutating zone and the constant part is able to effectively express the behavior of a polymorphic engine. The polymorphic characteristic is expressed through the mutating part and the constraint of consistency is expressed through the constant part. The conclusions drawn concerning the ability of polymorphic engines to possess high variation and propagation strengths are valid. However, the analysis should also take into consideration the fact that perfectly polymorphic decoder cannot be created because of the consistency constraints that we have mentioned before. The authors assume the existence of perfect polymorphic decoders in the algorithms they used in their genetic algorithms techniques without taking into consideration that there is a constraint factor. In other words, the decoders must be consistent in the sense that they must perform their work as decoders. Consequently, any attempt to make the decoder look polymorphic is subject to those constraints reducing the performance (in terms of making the decoder look polymorphic). Otherwise, the generated sequences of bytes will not make any sense since they will correspond to random instructions without any meaning or task. This constraint factor is not taken into consideration in the experiments conducted in [3]. We believe the experiments could have been more reliable by not even touching the decoders sequence of bytes and just analyzing them as they are. In this respect, our analysis draws much important conclusions in order to model polymorphic decoders. For these reasons, signature-based detection and modeling of polymorphic shellcode should not be abandoned.

Autograph [10] is a tool for signature generation against worms. It basically consists of two main phases: a first phase called suspicious flow selection and a second phase called signature generation. We elaborate more on the second phase since it is the one related to signature-based detection. However, we quickly describe the first phase to give an insight on its semantics. The first phase aims at identifying meaningful data to analyze. Basically, it acts as a filter to detect network packets that correspond to a worm behavior. The focus is on successful flows from IPs that make unsuccessful connection to more than S destinations. In other words, it identifies the scanning functionality proper to all worms. Autograph relies on the assumption that worms should exhibit common byte patterns that uniquely identify them. Based on this, the suspicious flows identified from the first phase are divided into small blocks. Each small block is ranked according to its occurrences; the higher is the occurrence, the higher is the prevalence. Obviously, this process becomes very ineffective against worms encoding their contents using a polymorphic engine since finding such common patterns is itself challenging for highly polymorphic engine. Moreover, the use of traditional pattern matching is not suited against polymorphic engines. We will show in the sequel that our shape signatures are much more effective against those engines.

Honeycomb [11] is a complex system for automating the process of signature generation for Network Intrusion Detection Systems (NIDS). It applies pattern-matching techniques and protocol conformance checks on many levels of the protocol hierarchy. We are not elaborating on the honeycomb architecture but we focus on the algorithms used in their pattern matching for the sake of generating attack signatures. The signature creation algorithm operates in the following manner. For each incoming packets, the following steps are executed:

- The connection state is checked for the new packet: when a connection exists, the state is updated; otherwise, a new state is created.
- The type of packet is verified: when the packet is outbound, the process stops; otherwise, the process continues.
- A protocol analysis is then performed at both the network and transport layers for IP, TCP and UDP headers. The analysis aims at detecting anomalies that correspond to misbehavior. The techniques used for detection of such abnormal traffic are common ones used by all NIDS. Once the packet is identified as abnormal, an analysis signature is created at this point.
- For each stored connection or each previous recorded packets:

– Honeycomb [11] compares the header to match possible IP addresses, TCP sequence numbers, common destination ports, etc. In other words, any information that might relate the packet to a previous connection is investigated.

– Once a connection is found, Honeycomb performs pattern detection on the data part or the exchange messages for a possible final signature.

As we can see, there are many problems that can be stated for the system such as the reliability of how packets are related to each other but at this point, we solely focus on the pattern detection technique used by Honeycomb. Pattern matching is based on the LCS algorithm [54] (Longest Common Subsequence). The whole issue is to know if the LCS algorithm is effective against polymorphic engines. The LCS is very efficient in finding common parts exhibited by a polymorphic engine. However, if the engine is very polymorphic, those common parts tend to be very small and sparse. Thus, it becomes ineffective when generating a reliable signature.

Polygraph [12] is a framework for generating signature attacks against polymorphic worms. We believe that this framework is the closest one, in its approach and design, to our work. The main architecture of Polygraph is discussed briefly as we focus on the way signatures are generated rather than the whole framework. The architecture consists of a network tap, which is deployed at a key point of a network such as a bridge point that links a network to the Internet. Then, traffic passes through a flow classifier that converts reassembled flows into contiguous byte flows. The next phase consists in analyzing the recorded traffic so that the suspicious flows are distinguished from the innocent ones. The distinction is based on detection techniques used by regular NIDS. The next phase is the core of the framework, which is the Polygraph signature generator. As we have mentioned previously, we are not going into the details of how the data is gathered. Let us investigate the algorithms used in order to perform pattern matching. Polygraph relies on three main techniques for signature generation:

- Conjunction signatures
- Token subsequence signature
- Bayes signature

The conjunction signatures consist of extracting all common tokens or chunks of bytes for the gathered data without taking into consideration the order in which they appear (their relative order). These tokens are labeled as the first phase signatures. The used algorithm is the Smith-Waterman local sequence alignment [55] for alignment issues. The next phase consists of the token subsequence signature. It uses the previously extracted tokens and tries to identify an order in which the tokens appear. Finally, the last phase of the algorithm is the creation of Bayes signatures. In this phase, each token is associated with a score and an overall threshold. These signatures exhibits the following probabilities:

- The probability that the token is in the suspicious flow
- The probability that the token is in the innocent flow

In our opinion, the pattern extraction mechanism used by Polygraph is the most efficient one among the signature-based detection mechanisms, especially the ability to try to find a sequence, or an order, in which the tokens appear. However, Polygraph fails in detecting highly polymorphic engines or worms for the following reasons. The algorithm relies heavily on the first phase of extracting common tokens. If the engine is highly polymorphic those tokens are very small in size and important in number. Knowing that highly polymorphic engines are using the whole byte spectrum, the number of common tokens is very high. Consequently, the signatures generated lead to many false positives when used in the detection process, and Polygraph is unable to extract a reliable pattern.

Hamsa [13] is a network-based automated signature generation system for polymorphic worms. Hamsa , like most signature-based detection frameworks, looks for invariant sequences of polymorphic malware. Even though, the algorithm for token extraction is quite similar to Polygraph [12], Hamsa differentiates itself in the sense that it looks for executable code instead of byte strings. In other words, it is not a simple token extraction, the engine tries to analyze the extracted tokens and see if they correspond to something meaningful. The basic steps that the engine goes through are the following:

- Extract the basic blocks
- Generate the corresponding Control Flow Graph (CFG)[3]
- Color the code on each node of the CFG
- Records fingerprints into a table
- If the number of fingerprints exceeds a certain threshold, they are identified as malware code

[3]CFG: In a control flow graph each node in the graph represents a straight-line piece of code without any jumps or jump targets. Directed edges are used to represent jumps in the control flow.

However, we can see that hamsa encounters the same problems as Polygraph [12]. When encountering a highly polymorphic engine, those tokens tend to be very small and important in number as stated previously. Highly polymorphic engines are using the whole byte spectrum, and so the number of common tokens is very high. Consequently, the number of fingerprints generated tends to increase leading to a high rate of false positives.

To summarize, most of signature-based detection mechanisms rely heavily on the invariant codes that polymorphic engines are subject to. We believe the idea to be relevant in the sense that those invariant codes come from the consistency constraint that polymorphic shellcode needs to fulfill. However, attackers are also aware of this observation. Accordingly, they implement engines that aim at inserting highly mutating code in between those invariant codes. This makes the detection mechanism useless, at best, subject to a high rate of false positive. We believe that the concept of signatures needs to be improved by incorporating signatures that detect invariant codes as well as mutated zones.

2.2 Polymorphic Engines

This thesis focuses on the most powerful polymorphic engines that are known so far. Our choice is based on the work done by Song et al. [3] since it is the only known work that has been successfuk in quantifying and measuring polymorphic engines. We also take into account the latest developments done in the area of polymorphic engines. In this regard, the choices come down to the following three engines: *Tapion* [27], *Shikata Ga Nai* [15] and *ADMmutate* [14]. In this section, we survey these three engines and the techniques that are used in order to make them polymorphic.

Tapion [27] was created by Piotr Bania to avoid code detection. It is probably, so far, the most powerful polymorphic engine since it incorporates a wide panel of evasion techniques. More specifically, it is the only engine possessing the ability to evade emulation-based techniques. Whereas the other two engines are resistant to most detection mechanisms, they do not possess functionalities designed to evade emulation-based detection. Its main techniques of polymorphism are the following:

- The decryption key is generated in a random fashion
- There is a swapping of shellcode blocks or pieces
- The decoder's length is constantly changing from one instance to another
- It uses multiple decryptor layers, thus performing the decryption of the shellcode as well as some of its decoder parts
- There is garbage padding in between instructions in order to prevent the detection of long invariants codes
- The garbage padding is unpredictable or random
- It uses random register in order to prevent detection. This is an important feature since many polymorphic engines are very often vulnerable to signature detection because of register usage
- Anti-emulation techniques are used such as performing decryption at random intervals and pseudo-infinite loops

The next engine is *Shikata Ga Nai*[15]. It is the official polymorphic engine of the *Metasploit* framework [15] for its shellcode generator. It exhibits the following features:

- It uses a polymorphic XOR additive feedback encoding against a four byte key
- The key is chosen according to a context-key selection. In other words, a memory map is created and then a sequence of bytes is selected from it to fit the desired key-length
- It uses dynamic instruction substitution and dynamic block ordering. The aim is to create on-the-fly decoders that look different from one instance to another
- Permutations of about 1.3 millions are used by the engine from one instance to another
- It selects dynamically the registers

The last engine, namely *ADMmutate*[14] was developed by the so-called ADM CREW. It is one of the breakthrough in polymorphic engines since it is the first ,historically, to implement new generation of polymorphic techniques that constitute the basic foundations for all the newly implemented polymorphic engines. It uses the following main techniques:

- Non-operational padding: which is padding of NOP instruction in between instructions. The padding is done using various non-operational instructions (being single or multi-byte argument), making them appear random and immune to possible pattern detection

- Out-of-order decoder generation: specifying where certain core instructions may be located in the decoder. This is mainly done by extending or reducing the length of the decoder to shift the bit positions of those operational instructions

- Random generation of instructions: randomly selecting instructions without disturbing the execution flow. This allows to explore the whole byte spectrum preventing the use of the same instructions from one instantiation to another. It also prevents against static binary code analysis as well as emulation detection

- Random key selection: selecting randomly keys, thus avoiding any prediction on how keys and offsets are selected

In summary, we described the main techniques of polymorphism used by the aformentioned engines. It is important to notice that each of these engines perform its own mutation techniques even tough at the end the goal is the same which is creating polymorphic decoders capable of evading detection.

2.3 Conclusion

Polymorphic engines constitute a real threat in the sense that they are able to defeat various types of detection mechanisms. Whether using network detection, emulation techniques or any of the discussed mechanisms, hackers always find ways to improve evasion techniques. Consequently, it is important not to drop any of the discussed mechanisms but rather improve them in order to cope with polymorphic engines. Most importantly, it appears necessary to model those engines in order to better understand their inner workings.

Chapter 3

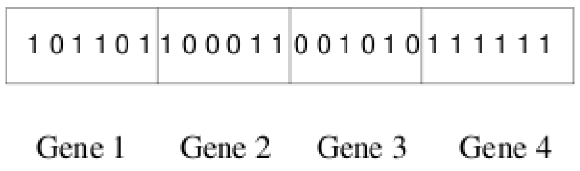
Genetic Algorithms

In this section, we discuss the main elements that compose any Genetic Algorithm (GA) [16]. Moreover,we will discuss how these concepts will be applied in this thesis. Genetic algorithms is a popular technique in evolutionary computation trying to emulate the natural process of evolution. It is an established technique that is known to be efficient in heuristic search problems. The main challenge in the use of GA is to adequately define the problem. In our case, the problem consists in finding a common pattern for a given population of decoders for a specific polymorphic engine. Our choice is motivated by the fact that polymorphic engines behave like entities in nature. They are able to generate from a common basis multiple instances. During the process of instantiation, the generated entities are subject to mutations making them unique in terms of bytes. Yet, those same entities belong or come from the same polymorphic engine.

3.1 Main Components

GA have two distinct elements: the individuals and the populations. An individual represents a single solution whereas a population represents the set of individuals currently involved in the search process. A chromosome can be seen as the raw genetic information of an individual (its genotype) that can be decomposed into genes. Figure 3.1 illustrates a representation of a chromosome (encoded by bit strings); genes are the subdivisions or building blocks that form the chromosome.

Figure 3.1: Chromosome Representation in Binary



Genes are in fact the basic instructions for building a GA whereas a chromosome represents a sequence of genes. They are bit string of arbitrary lengths. Each individual in a GA can be valued or quantified by a fitness function. The fitness function is the value of an objective function to indicate how good the individual is, as well as how close he is to the optimal one. A population is a collection of individuals that are going to be evaluated according to the criterion defined by the fitness function. A population has two important aspects that must be considered:

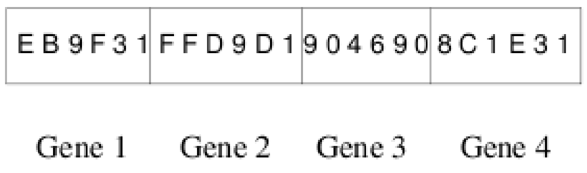
- The initial population generation

- The population size

For each problem, the size of the population depends on the complexity of the defined problem that needs to be solved. In theory, the initial population should have a large pool of genes in order to explore the whole search space. All the possible combination of genes should be present. Consequently, the initial population is often chosen randomly. In our case, polymorphic engines claims to generate such randomness, thus solving the problem of exploring the whole search space. In fact, Song et al. [3] have shown that a powerful polymorphic engine (which is the case of the chosen engines) is able to effectively explore the whole byte spectrum. On the other hand, our fitness function tries to identify common patterns to each gene in their reproduction process. In this respect, we can have a glimpse of the main components that will compose the genetic algorithm engine for shape signature extraction.

The encoding is another important aspect of GA. It deals with representing each individual gene according to a specific alphabet. Since we are dealing with shellcode, an appropriate representation of the genes is the hexadecimal alphabet (0-9, A-F). Figure 3.2 gives us an idea of how the chromosomes (which corresponds to shellcode) are encoded.

Figure 3.2: Chromosome Representation in Hexadecimal



3.2 Breeding

The breeding process is the core engine of GA. Through this process, the search creates new possible fitter individuals. Three steps characterize it:

1. Parents' selection
2. Crossing of parents to create new individuals
3. Creating the new population by replacing the old individuals by the new ones

Once the process terminates, the newly created population inherits characteristics from the previous one. The most important idea to keep in mind is that the genetic information is passed on from one generation to another in the breeding process.

3.2.1 Selection

Selection consists of choosing parents from the population for crossing or reproduction. The purpose of the selection is to focus on fitter individuals hoping that their offsprings (childs generated from the reproduction process) have higher fitness. The method randomly selects chromosomes from the population according to their fitness values. The higher the fitness, the more chances an individual has to be selected. The selection pressure is the degree of how much we favor the best individuals (the ones with the highest fitness values). In other words, it is the threshold set, in terms of fitness value, for deciding which individuals are selected for reproduction. One important issue in selection is the convergence rate of GA, which depends largely on the selection pressure value. The convergence rate expresses how close we are towards finding the solution. The higher the selection pressure is, the higher the convergence rates are. The problem is that high selection pressure results in quick convergence toward a solution that might be wrong. A low selection pressure results in a low convergence rate slowing down the GA by generating unnecessarily time to find the optimal solution.

Two types of selection schemes can be distinguished: proportionate selection and ordinal-based selection. Proportionate selection picks out individuals based upon their relative fitness selection. On the other hand, ordinal-based selection selects individuals based on their rank within a population. As a consequence, a predefined threshold must be defined in order to decide which individuals must be selected.

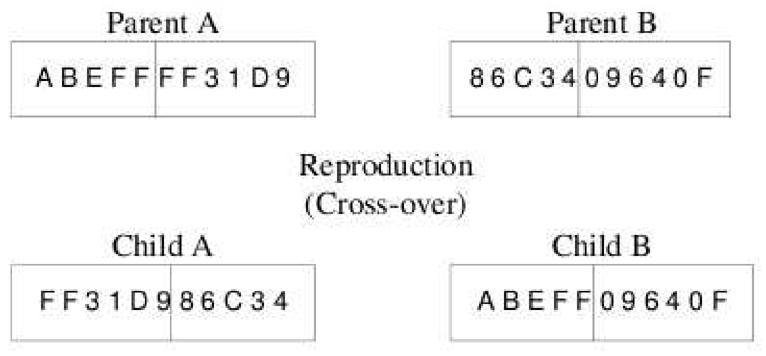
3.2.2 Crossover

The crossover is the process of taking two parents and producing a child from them. In other words, it is a reproduction process. After the reproduction, the child goes through a selection process that determines its fitness or ability to live for the next rounds. The crossover can be defined (roughly) in three steps:

1. Random selection of two individuals as parents
2. Selection of a crossing sites along the string length
3. The position values are swapped between the two strings in a random fashion

In fact, there are many techniques of crossover (single point, two point, n-points, etc). In our case, we use an n-points crossover with constant crossing points. Consequently, we choose to cross the string every n-bit as we will explain latter in this thesis. Figure 3.3 illustrates the concept of crossover using a single point crossover.

Figure 3.3: Single-Point Crossover



3.2.3 Replacement

The last stage of the breeding process is the replacement. The reproduction process ends up with four individuals: the two parents and the two offsprings. However, not all of them can constitute the new population. We need to determine which individuals goes through the next rounds. We can clearly see that the way we replace individuals, determines the convergence of the solutions. In fact, there are many replacements techniques but they all fall into two main categories: generational updates and steady state updates. The generational update consists of generating N children from a population of N parents to form the next population at the next stage. We replace all the old population with the new one making inter-generational reproduction impossible. The second family of replacement techniques is steady state updates where new individuals are inserted into the new population instead of completely erasing the old population.

As a conclusion, these are the main elements that constitute a genetic algorithm engine. We can already see that our problem can fit into this type of evolutionary algorithms. In the next section, we describe John Hollands schema theorem [29]. The aim is to show how effective the genetic information is passed and how effective we could be in finding patterns. The schema is also important since it is a kind of introduction into the concept of shape signatures.

3.3 Schema Theorem

The schema theorem is based on John Hollands schema theorem [16, 29] which describes a scheme as a template for describing a subset of chromosomes with similar sections. The schema consists of bits or hexadecimal characters and meta-characters (wildcards). The schema is a kind of template for describing similarities among patterns in chromosomes. Holland derived an expression that describes the extent at which those schemata are passed from one generation to another. A good schema propagates in next generations in an increasing fashion. As a consequence, schemata that are low-order well-defined and above average fitness are preferred and constitute building blocks of the GA.

A schema is a similarity template describing subset of string displaying similarities at certain positions. In our case, it is formed by the following ternary alphabet 0-9, A-F, X with X being a wildcard allowing the description of all possible similarities among string of a particular length and alphabet. The GA model in schema theory is based on proportionate selection in which the probability of selecting a solution in a current population is proportional to its fitness. The schema theorem is called the Fundamental Theorem of Genetic Algorithm. For a given schema H, let:

- $m(H, t)$ be the relative frequency of the schema H in the population of the t^{th} generation
- $F(H)$ be the mean fitness of the elements of H
- $O(H)$ be the number of fixed bits in the schema H, called the order of the schema
- $\sigma(H)$ be the distance between the first and the last fixed bits of the schema called the definition length of the schema
- f is the mean fitness of the current population
- P_c is the crossover probability
- P_m is the mutation probability
- l is the length of the code

Then:

$$E[m(H, t+1)] \geq m(H, t)\frac{F(H)}{f}[1 - P_c\frac{\sigma(H)}{l-1} - O(H)P_m]$$

The main information that can be derived from the schema theorem formula is that the number of individuals matching a schema H grows at each time step like the ratio of the fitness $F(H)$ and the mean fitness of the current population f. Consequently, an above average schema, i.e, a schema with $F(H) > f$, will proliferate successfully in the next generation. At the opposite, a below average schema will not.

If we relate the formula to polymorphic engines, we can consider a schema as a potential pattern. If we use a traditional signature matching mechanism, a pattern must match a polymorphic decoder's portion bit per bit. Knowing from Song et al.[3] that polymorphic decoders exhibit a high variation and propagation rate, we expect to get $F(H) < f$ since the diversity and uniqueness of generated shellcode is so high that the schemata are always below average. Consequently, those schemata are not able to proliferate resulting on the impossibility to generate a signature for a polymorphic decoder. On the other hand, if we change the model of signature by trying to detect the mutation points by considering wildcards in the pattern and the constant points, we are able to increase the fitness of the schema leading to potential above average schemata (with $F(H) > f$). Thus, they are able to proliferate and candidate signatures could be found.

3.4 Conclusion

Genetic algorithms offer a convinient way of studying and representing polymoprhic decoders. The way genetic information is passed (in our case, the genetic information is mainly the byte) allows us to extract possible common patterns for each polymorphic engine. The search heuristic property of GA is very useful for search problem. In our case, the challenge is to look for common patterns. In the case of polymorphic engines, the addition of meta-characters with regular hexadecimal characters makes the extraction of shape signatures accuurate and precise where constant parts can be represented by regular hexadecimal characters whereas mutated part can be represented by wildcard characters. This model of signature is potentialy able to generate above average schemata compared to a traditional signature model, leading to potential signatures.

Chapter 4

Shape Signatures

In this chapter, we detail the functioning of the genetic algorithm engine and how it allows us to extract the shape signatures. Our starting point is the fact that polymorphic engines, even when they mutate, leave artefacts making them potentially subject to detection. [10, 11, 12, 13] have shown the validity of such assumption. Moreover, those engines must obey the structure of assembly language by decoding the encrypted shellcode properly. Thus, any polymorphism technique attempt to make the decoder look highly random is greatly reduced by the consistency constraint. Finally, the mutation techniques themselves are subject to detection. In order to evade detection mechanism based on traditional signature detection, highly polymorphic engines pad mutated zones in between those potentially detectable parts and constantly change their byte positions. Consequenty, signature-based detection work done so far [10, 11, 12, 13] is either failing at extracting signatures or having too small tokens leading to a high false-positives. Based on this, we define a new concept of signature called *shape signatures*. This type of signatures takes into consideration, in their detection mechanism, the possible mutated zone and the constant part. We believe that this combination is able to effectively express the behavior of a polymorphic engine by taking into account their high polymorphism (through the mutated part) and the constraint of consistency (the constant part). We are able to explore the whole decoder space by using GA. Thus, even with a shift of bit position, we are still able to detect potential patterns. The way signatures are extracted through our GA engine is explained in details in 4.3. Before detailing the engine, it is important to stress the fact that we are basically dealing with two families of polymorphic engine: fixed-length decoder engines and changing length decoder engines. They both rely on the same pattern extraction technique but present some differences regarding the crossing point selection. In the next section, we present the common points between the two engines and explain the differences when dealing with both families.

4.1 Population Generation

Our detection is based on the deciphering engine. Consequently, we select a well-known polymorphic engine and run it against a single shellcode. Knowing the size of the shellcode, we extract from the output generated by the engine the deciphering part and store it in a file. If the engine generates various length deciphering decoders or adds jumps or padding, we do not alter the obtained output to preserve the genetic information carried by each member of the population of decoders. By doing so, we get a large-size population of deciphering routines for a specific polymorphic engine. This constitutes the input or initial population. The binary representation for the GA is obviously an hexadecimal one (since we are dealing with shellcodes). The decoders are the chromosomes for the GA. Assuming the polymorphic engine is for example *Shikata Gai Nai*[15], we call P the population generated from it. Accordingly $P = \{D_1, D_2, D_3, \ldots, D_n\}$ where D_m is a decoder and n the size of the population. At each epoch, reproduction must occur through crossovers and recombination. To this end, we need to define what type of crossover is used and how the recombination occurs. After each epoch, the fitness function evaluates each individual and proceeds with the selection and replacement of population. The next sections are dedicated to the detailed explanation of how these steps are executed. For each polymorphic engine, we generate a set of 1000 decoders. Based on that initial population set, we extract the signatures. We can observe the strength of a polymorphic engines by only generating very few instances of polymorphic shellcode and compare their differences. At the same time, we need to evaluate their full strength by generating an important number of instances. Consequently, we believe the number 1000 to be a good compromise between performance and evaluation. For experimental testing, we create three new sets of 1000, three sets of 5000 and three new sets of 10000 decoders in order to validate the various results we come across.

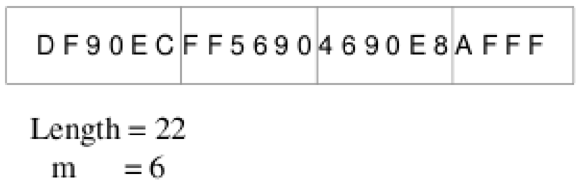
4.2 Genetic Algorithm for Signature Detection

In this section, we detail how the genetic algorithm engine works with respect to our problem, which is the extraction of valid shape signatures. We detail the crossover process, then describe how the reproduction and recombination phase are executed. Afterwards, we define the fitness function of the GA and explain the selection and replacement phases.

4.2.1 Crossover

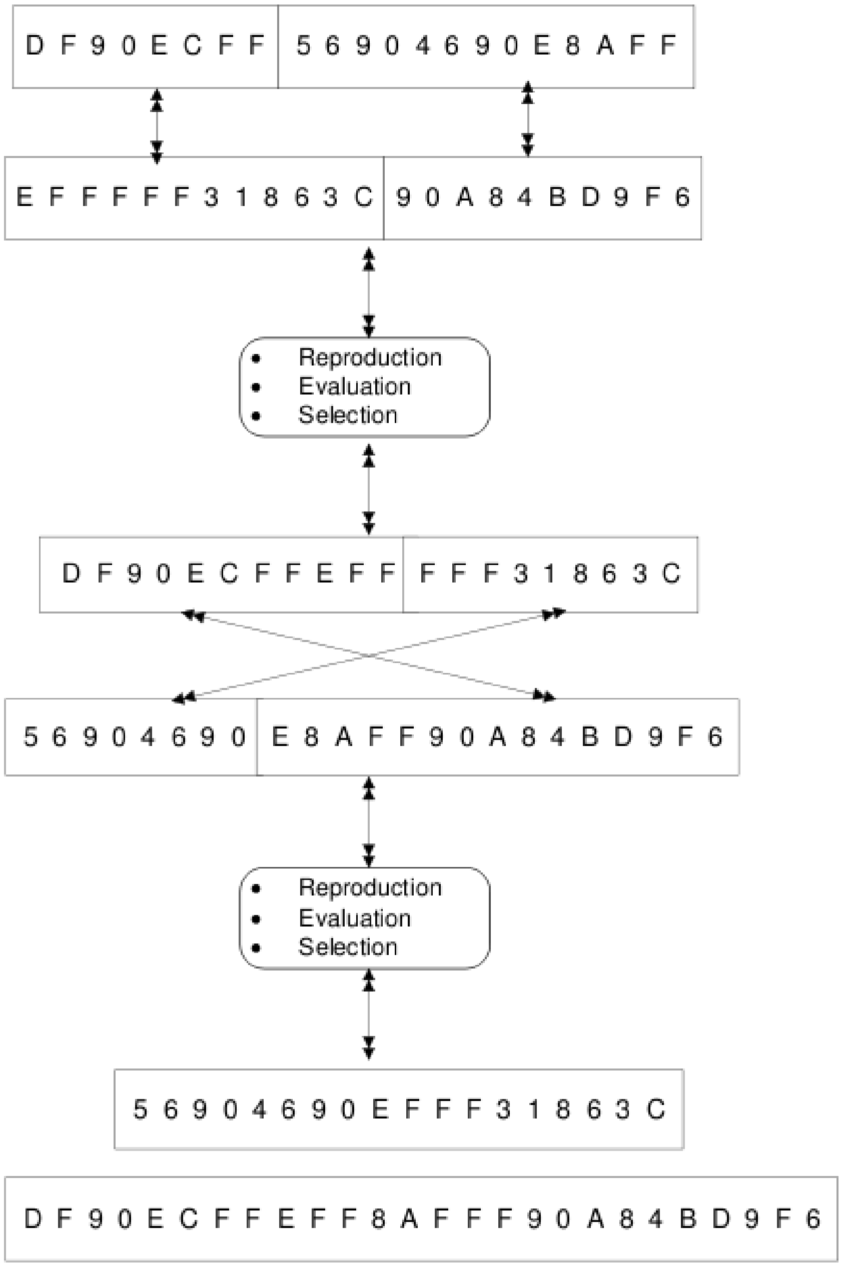
Since the size of the decoders is relatively important, especially when an hexadecimal encoding is used, we need multiple crossover points. Those multiple crossover points generate the building blocks for reproduction, and thus, for signature detection. The motivation behind the use of multiple crossovers is closely related to the Schema Theorem [29], which states that good schemata with short definition length propagate and grow rapidly in the population. Usually, when a crossover (being single point or n-point) is chosen, the crossing point tends to be random across the decoder at each iteration. In our case, this randomness leads to invalid genes or building blocks. Indeed, if the crossing point is chosen randomly at each round, we end up with sequences of bytes mixing with each other. Consequently, the generated genes do not make sense semantically. By defining a n-point crossover with fixed crossing point, we generate genes that remain the same across rounds preserving their semantics. In other words, at each round, the same crossing point is chosen for each chromosome in the current population (at each n bits, we choose a crossing point at that position). Let $D_1 \in P$ and L_1 be the length of D_1 where D_1 is a decoder from a population P. Let m be the crossing point length with $m < L_1$. Accordingly, a crossing point at each m bits is defined in Figure 4.1.

Figure 4.1: N-Point Crossing Points for a Chromosome



To illustrate the inability of using random crossing points, let consider two decoders D_1 and D_2 of the same length. In this example, we use a single-point crossover and assume the selection and the fitness evaluation to occur at epoch 0. The two chromosomes go through the process of reproduction and recombine to produce two offsprings. Figure 4.2 illustrates how the use of random crossing point leads to the creation of non-logical chromosomes.

Figure 4.2: Random Crossing-Points Effects



D F 9 0 E C F F | 5 6 9 0 4 6 9 0 E 8 A F F
E F F F F F 3 1 8 6 3 C | 9 0 A 8 4 B D 9 F 6
• Reproduction
• Evaluation
• Selection
D F 9 0 E C F F E F F | F F F 3 1 8 6 3 C
5 6 9 0 4 6 9 0 | E 8 A F F 9 0 A 8 4 B D 9 F 6
• Reproduction
• Evaluation
• Selection
5 6 9 0 4 6 9 0 E F F F 3 1 8 6 3 C
D F 9 0 E C F F E F F 8 A F F F 9 0 A 8 4 B D 9 F 6

In summary, if the crossing point keeps changing, we end up with non-logical building blocks. This because the shellcode chunks keeps mixing with one another and so the detection of patterns becomes impossible. Thus, it is important to keep the crossing point constant to generate logical building blocks for the reproduction. We can then work on them in order to extract common patterns that might exist. In our GA engine, the user is asked to input the crossing point. Usually, a crossing point of length m such that m is approximately one fourth of L (where L is the decoder's length) is a good candidate for the pattern extraction or signature detection. This choice is justified by the following reasons. According to Song et al. [3], in average no more than $\frac{1}{3}$ of any polymorphic decoder is subject to transformation or mutations. We have also to recall the schema theorem [16, 29] property stating that low-order, well-defined average fitness schemata will combine to form high-order, above order fitness schemata. By taking into account those two factors, we have come to the conclusion that a crossing point of length $m \equiv \frac{L}{4}$, with L being the decoder's length, is an appropriate choice.

4.2.2 Reproduction and Recombination

In this section, we detail how the reproduction and recombination processes occur. We also describe the reproduction sequence generation, the process of parents' selection for pairing and the reproduction process itself.

4.2.2.1 Reproduction Sequence

Our N-points crossing technique generates logical building blocks that are used for the reproduction, recombination and evaluation process. The next question that remains is how the reproduction and thus the recombination occur. Let D_n and D_m be two elements belonging to a population P of decoders. Let m be the crossing point length for the engine. The two decoders are assumed to be of the same length. It is important to mention that some polymorphic engines generate various length shellcodes. Consequently, we need to transform the GA [16] into a Fast Messy Genetic Algorithm (FMGA) [30]. The latter algorithm is detailed in section 4.5. Being fixed or variable length decoder engine, the reproduction/recombination process deals with decoders using a unified length. After that, a sequence of reproduction must be defined.

Our reproduction algorithm performs the following operations. At each epoch or round, a random reproduction sequence is generated for the given population. Our function reproduction sequence takes as input the number of building blocks or genes generated using the N-points crossover, let call it N. It creates a $N \times N$ matrix with the indexes referencing to the genes for both parents. Whenever a value of 1 is encountered in the matrix, the corresponding genes must be associated. It randomly creates the reproduction sequence between genes. It is important to mention that all the genes are used only once in the recombination. Let D_n and D_m be the parents and m be the crossing point. We generate N genes for each parent. Let $A = \{0, 1, 2, \ldots, N-1\}$ and $B = \{0, 1, 2, \ldots, N-1\}$ be two arrays representing the indexes of genes for the parents used for reproduction. Each time we select an index for a parent, the corresponding genes is retrieved from the pool of selection. At the end, all the genes are used and paired together to produce two offsprings. The reproduction of two parents results always in two offsprings. This is justified by the fact that we want to increase the accuracy of the signature or pattern detection by using all the genes from each decoder or chromosome and not just some chunks. The generated reproduction sequence remains constant for one round or epoch. In the next round, the reproduction sequence is changed.

4.2.2.2 Parent Selection

At each epoch or round, we need to select who is going to reproduce with whom. We assume at this stage that the selection, using the fitness function has occurred. Once the reproduction sequence has been produced, a function called pairing, which takes as input the population at the corresponding epoch, is responsible for pairing the parents according to the produced reproduction sequence. The function generates a pool of indexes POOL $= \{0, 1, 2, \ldots, S\}$ where S is the size of the population. Indexes are picked up randomly and removed from the pool. Afterwards, they are used to generate a long sequence called PAIRING until the pool is emptied. If S is an even number, then PAIRING $= \langle p_1, p_2, p_3, \ldots, p_S\rangle$. If S is odd, then PAIRING $= \langle p_1, p_2, p_3, \ldots, p_{S+1}\rangle$ where p_i is a random index from POOL. Afterwards, we generate the pairs of parents for reproduction by taking each two index successively from the pairing sequence resulting in reproduction pairs (p_i, p_{i+1}). At the end, all members or individuals from the population are melting together; it is important to make sure that none of the individuals from a population in a certain epoch is missed.

4.2.2.3 Reproduction

At this point, we have the reproduction sequence as well as the pairing sequence. The next phase is to perform the reproduction. The reproduction sequence determines how the genes are mixed together in order to produce a new offspring. The pairing sequence determines with whom each parent is going to melt. Having those parameters, the reproduction takes place.

4.2.3 Fitness Function

The fitness function is the core element of the engine, since it constitutes the basis for the selection process when moving from one epoch to another. As such, the solution candidates are selected at the end of the GA to give us the final signatures or patterns. The fitness function is based on the schema theorem [29]. Consequently, it deals with the following alphabet: $\{0, 1, \ldots, 9\} \times \{A, B, \ldots, F\} \times \{X\}$ where X is a wildcard. The fitness function is based on a sliding window technique. The fitness function works on the genes or building blocks generated from the reproduction. Our engine must be able to detect patterns among new offsprings based on the ternary alphabet described earlier. This pattern should be able to detect changing parts of shellcode chunks as well as fixed parts. The fitness function takes as input a population of decoders. For each single decoder, the function works on its building blocks generated from the reproduction Let D be an individual from a population P at an epoch n and let m be the crossing point length such that: $D = \{d_1, d_2, d_3, \ldots, d_j\}$ where d_j is a building block or gene of the decoder. Let w be the sliding window with $w < m$. Let us define the operator $\otimes$. Let $\Sigma = \{0, 1, \ldots, 9\} \cup \{A, B, \ldots, F\}$ and let Σ^+ be the set of all non-empty finite hexadecimal sequences (over σ). Then, $\otimes : \Sigma^+ \times \mathbb{N}^+ \rightarrow \mathcal{P}(\Sigma^+)$ where $\mathcal{P}$ is the power set.

The operator $\otimes$ is defined in the following manner: we use functional programming element to define our operator such as rec[1],hd[2],tl[3] and @[4]. n-seq is a function taking two arguments: a building block and the sliding window.

[1]rec: means recursive function

[2]hd: hd is a single list element and means head of list

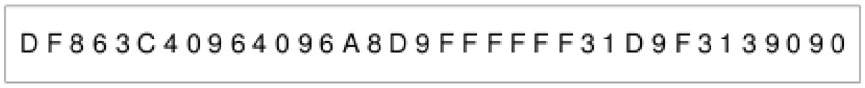
[3]tl: means rest of the list

[4]@: means concatenation

```
$\otimes(d_j, w) =$ let rec n-seq$(d_j, w) =$
        if $m = 0$ then $<>$
            else $hd(d_j)@n - seq(tl(d_j), w - 1)$
in
        let $r = n - seq(d_j, w))$
        in
            $\{r\} \cup \otimes(tl(d_j), w)$
        end
end
```

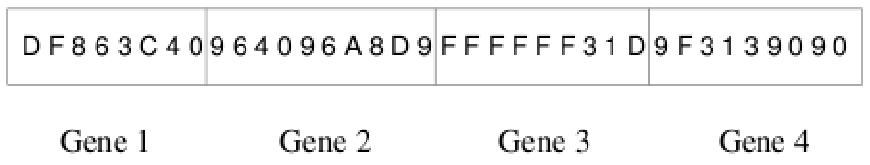
Consequently, $\otimes(d_j, w) = \{w_1, w_2, w_3, \ldots, w_j\}$ with $\forall w_i \in \otimes(d_j, w)$, we have $w_i = w$. Each w_i is a sub-sequence from the building block d_j. In order to illustrate this, we consider the following example. Let assume that we have a decoder D of length 36 bytes illustrated in Figure 4.3 .

Figure 4.3: 36-Byte Decoder Example


DF863C40964096A8D9FFFFFF31D9F3139090

The crossing point length is such that $m = 9$. Thus, each 9 bytes, we have the building blocks or genes. Figure 4.4 illustrates the generation of genes for each decoder:

Figure 4.4: Decoder's Genes



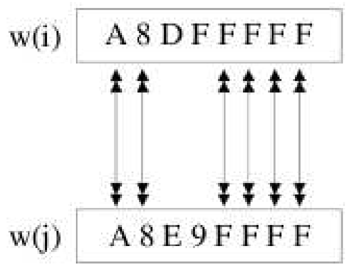
Thus $D = \{d_1, d_2, d_3, d_4\}$ where $d_i = Gene_i$. We have a sliding window $w = 5, (w < m)$. By applying the operator $\otimes$ we get the following for $\otimes(d_i, w) = \{w_1, w_2, w_3, w_4, w_5\}$. Table 4.1 shows the windows extraction for a gene:

Table 4.1: Window Extraction for a Gene

Gene 1		
w_1	=	DF863
w_2	=	F863C
w_3	=	863C4
w_4	=	63C40
w_5	=	3C409
w_6	=	C409

w_6 is discarded since $w_6 < w$. The same applies for the other building blocks. Since the defined fitness function is based on the schema theorem [29], it checks for correlation between hexadecimal values as well as wildcard. The problem inherent to wildcards is the fact that they should not be used extensively. Otherwise, each fragment of shellcode, when compared with another, produces a positive correlation. Therefore, we check that at least $\frac{2}{3}$ of the bytes correlate with each others. If it is the case, then the fitness value increases. For instance: let $w_i =$ "$A8DFFFFF$" and $w_j =$ "$A8E9FFFF$". The function compares each character with each other. If $\frac{2}{3}$ or more of the characters are similar, then we have a candidate pattern. Figure 4.5 illustrates the idea:

Figure 4.5: Correlation Process



Here we can see that 6 out 8 hexadecimal values correlate. Consequently the following pattern "$A8XXFFFF$" is a candidate since $\frac{6}{8} \geq \frac{2}{3}$. If the correlation is total, then we have a pattern with no mutation part. The correlation function checks for fixed bits as well as mutated bits. Let define the operator $\oplus$ that performs that correlation:

- $w_i \oplus w_j = 1$, if the two segments correlate for $\frac{2}{3}$ or more.
- $w_i \oplus w_j = 0$, if the two segments correlate for strictly less than $\frac{2}{3}$.

The operator is such that:

```
$\otimes \bowtie (u, v) =$ let $h_u = hd(u)$
                 $h_v = hd(v)$
in
                 $(h_u = h_v)@ \bowtie (tl(u), tl(v)$
end
```

Then correlation is such that:

correlation$(u, v, w) =$ let $l = \bowtie (u, v)$

in

$$\frac{\sum_{i=0}^{n} l_i}{w}$$

end

The fitness value computation proceeds in the following manner. Let $F(D_n) = (w_{d_i}, w_{d_j}, w)$ be the fitness function value where: w_{d_i}, w_{d_j} are windows extracted from respectively d_i, d_j such that $w_{d_i} = \otimes(d_1, w)$ and $w_{d_j} = \otimes(d_j, w)$. w is the window size and D_n a decoder or chromosome from the current population. Thus, the following is the formula for the fitness function:

$$F(D_n) = \sum_{i=0}^{n} [\forall w \in w_{d_i} \oplus \sum_{j=0}^{m} (\forall w \in w_{d_j})] \quad with \quad i \neq j$$

In other words, when dealing with a decoder, the following steps are followed in the correlation process:

1. Generate the windows for each gene or building block.

2. For each gene and for each of its windows, apply the correlation function against windows from other genes of the same decoder. The process continues until all windows are checked.

3. At the end of the process, compute the fitness value for the decoder.

Having defined the fitness function, we need to explain the selection process that results from it.

4.2.4 Selection

The selection phase is another core element of the engine. Since we base the fitness function and more generally our GA engine on the schema theorem [29], the selection process must be a proportionate one. In other words, the selection must pick up individuals based on their relative fitness values. This make sense since such selection is recommended when the final optimal solution is unknown .After applying the fitness function, we get the correlation value for each member of the population at a certain epoch or round. In the next step, we compute the probability of each member based on their fitness value. We basically compute the total fitness for all members of the population and then compute their individual probabilities. By doing so, we got their relative probability of being selected. Let P be the population such that: $P = \{D_1, D_2, D_3, \ldots, D_n\}$ with D_n being a decoder or chromosome. After applying the fitness function, we get the individual fitness value for each member. Let FIT be the individual fitness for each population members such that: FIT $= \{F_1, F_2, F_3, \ldots, F_n\}$ with $F_n = F(D_n)$ (F being the fitness function defined in the previous section). Let $F(total)$ be the total fitness value for all population members. $F(total) = \sum F_i$. The individual probability of each member is $P_i = \dfrac{F_i}{F(total)}$. Let P be the list ofprobabilities for each member of the population: $P = \{P_1, P_2, P_3, \ldots, P_n\}$ with $P_i = \dfrac{F_i}{F(total)}$. This process respects the requirement of being proportionate. Having the relative probabilities of selection for each population member, we need to proceed with the selection. The selection goes by setting a threshold probability corresponding to the probability of selecting an individual for the initial population. At epoch 0 or at $t = 0$, if the total population is *total_init*, then the threshold is $thresh = \dfrac{1}{\textit{total\_init}}$. If the probability of an individual is greater or equal to the threshold, then the individual is selected for the next round, otherwise he is dropped.

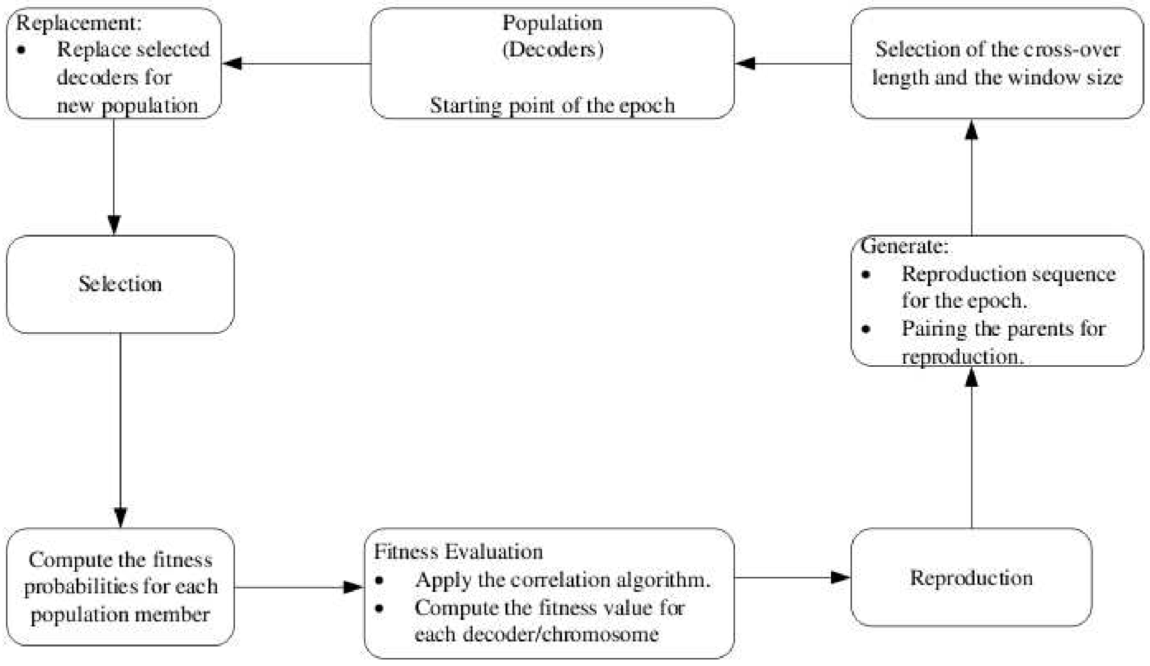
4.2.5 Replacement

The next step of the engine is the replacement process. There are two main types of replacements: generational updates and steady state updates. In our GA, generational update is selected, which means that the old population is replaced entirely by the new one making inter-generational reproduction impossible. The reason behind this choice is that we are aiming to look for possible pattern extraction and see if they propagate from one generation to another. Thus, it is more convenient to work on the building blocks that have passed the selection test to see if they are again reproduced with blocks from another decoder.

4.2.6 Search and Termination

Our engine runs for 1000 rounds or epochs, once that number has been reached, the last population is our candidate for signatures or patterns. Figure 4.6 summarizes the functioning of the engine.

Figure 4.6: Summary of the Engine


Replacement:
• Replace selected decoders for new population
Population (Decoders)
Starting point of the epoch
Selection of the cross-over length and the window size
Selection
Generate:
• Reproduction sequence for the epoch.
• Pairing the parents for reproduction.
Compute the fitness probabilities for each population member
Fitness Evaluation
• Apply the correlation algorithm.
• Compute the fitness value for each decoder/chromosome
Reproduction

At this stage, we have described how the engine operates. However, we still do not have the final signatures. In the next section, we discuss the extraction of signatures from the pool of candidate solutions at the end of the engines rounds.

4.3 Signature/Pattern Extraction

Once the engine or GA ends, we have a set of decoders that are candidate solutions for signatures or patterns for the analyzed polymorphic engine. The problem is that those decoders or chromosomes are not the solution themselves. The building blocks constituting the decoder contain pieces of shellcode that are patterns and others that are not. The main reason behind that is the fact that a decoder can pass the selection by having some of its genes containing patterns and others that do not. Our aim is to traverse the whole solution population and extract those patterns. Once the patterns are extracted, we need to verify their validity. In other words, we need to check the pattern's validity against the initial population, which have not been altered by the reproduction process. Moreover, the building blocks that we have do not contain wildcards. Accordingly, we need to include them in the pattern extraction.

4.3.1 Patterns extraction

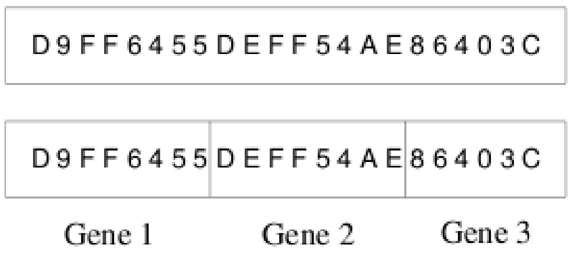
Pattern extraction is somehow similar to the correlation function (fitness function), but with some differences. Let S be the final population or solution population such that $S = \{S_1, S_2, S_3, \ldots, S_n\}$, each S_i being a decoder or chromosome. The function extracts patterns according to the following steps:

- Take each decoder and extract its building blocks knowing the window size.
- From each window generated from a single gene (by applying the operator $\otimes$), we check if it is correlated against all other windows from other genes of the same decoder. The correlation algorithm is the same as the one from the fitness function.
- If a correlation exists,then the analyzed window is a candidate pattern.
- Generate a signature by replacing the uncorrelated parts by a wildcard.
- Insert the signature into the pool of patterns. If the signature already exists,discard it, otherwise, insert it into the pool of patterns.
- Repeat the process until all the genes or building blocks are traversed.

- Repeat the process until all the decoders are traversed (until the whole population is checked against our extraction algorithm).

At the end of the process, we have the signatures. Let us illustrate the extraction mechanism by taking the following example. Let D be a member of the last population with $w = 6$ and $n = 8$. From the crossing length n, we get the genes or building blocks presented in Figure 4.7.

Figure 4.7: Decoder's Building Blocks



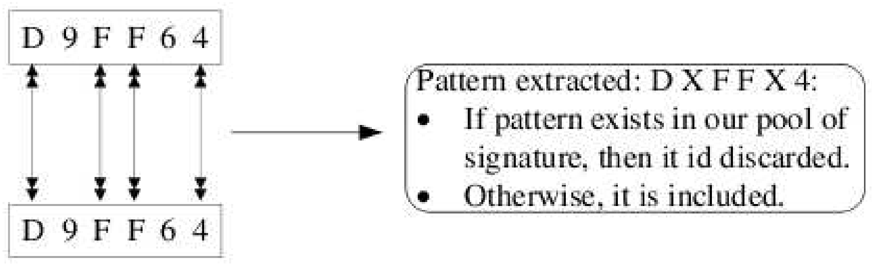
Now we need to apply the operator $\otimes$ against all the genes to generate their windows:

Table 4.2: Extracted Windows from Genes

Gene 1	Gene 2	Gene 3
$w_1 = D9FF64$	$w_1^{'} = DEFF54$	$w_1^{''} = 86403C$
$w_2 = 9FF645$	$w_2^{'} = EFF54A$	
$w_3 = FF6455$	$w_3^{'} = FF54AE$	

We just focus on w_1 for the correlation. By checking w_1 against all other windows, we see that it correlates at $\frac{2}{3}$ (or more) with $w_1^{'}$. Figure 4.8 illustrates the pattern extraction process.

Figure 4.8: Pattern Extraction


D 9 F F 6 4
D 9 F F 6 4
Pattern extracted: D X F F X 4:
• If pattern exists in our pool of signature, then it id discarded.
• Otherwise, it is included.

Then, the same process is repeated with all the other windows from the same decoder and the whole process is repeated until all the solution population is traversed. At the end of the extraction, we get the signatures or patterns from the studied polymorphic engines.

4.3.2 Filtering Process

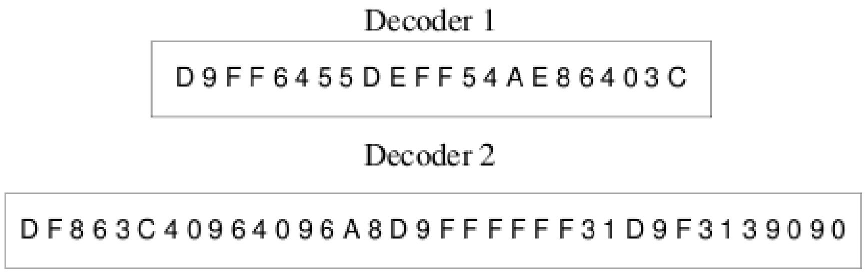
The importance of the filtering process resides in the fact that we need to check the validity of the final signatures against the initial population who has not been altered by the reproduction process. The filtering consists in taking each extracted patterns and checking if it is present in more than 190 decoders out of 200 decoders selected from the initial population. Before showing the experimental results, we need to explain first how the engines work against changing-length decoders. The process that we have described so far is exactly the same. The only difference with changing-length decoders is the fact that we apply some algorithms prior to launching the engine, to generate a population of decoders of the same length.

4.4 Changing-Length Decoders

For polymorphic engines that generate changing length decoders, the process is very similar to the signature detection described in section 4.3. However, some steps for generating fixed-length decoder are performed before the engine launching. In this section, we describe how this transformation is done.

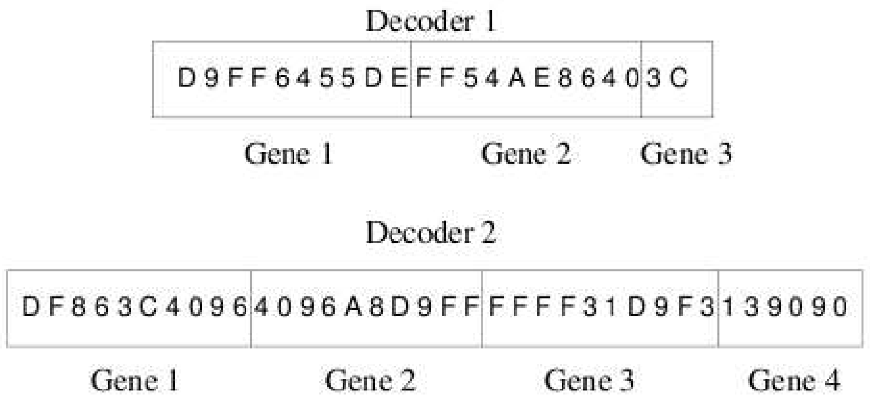
The problem with changing length decoders is that the described N-point crossover generates for each decoder a different number of building blocks or genes during the reproduction process. As a consequence, we generate a reproduction sequence for a population that has each of its members having a different number of genes or building blocks. One solution would be to drop some genes along the reproduction process, but we might lose some existing patterns. In fact, dropping some genes is inevitable. Accordingly, we must find a way to include as much genes as we can for each decoder. In order to explain why the fixed N-point crossover does not work, let take the following example. Let D_1 and D_2 be two decoders from an initial population with different lengths, illustrated in Figure 4.9.

Figure 4.9: Different Length



Let assume that the crossing-point length $n = 10$. The resulting building blocks or genes are shown in Figure 4.10 .

Figure 4.10: Various Length Decoders' Building Blocks



The generation of a reproduction sequence becomes complicated when dealing with a population that has a large number of individuals. Consequently, we need to drop some genes from one of the decoder (probably the one with the biggest length) in order to couple the two decoders. Even if we are able to overcome the population size problem, we still need to drop a large number of genes in order to make the reproduction successful, which is not a good idea. In fact, we must ensure that we can take the maximum number of genes. Genetic algorithms have a special class of algorithms called Fast Messy Genetic Algorithms (FmGA) [30] that deals with an initial population of individuals with various lengths.

4.5 Fast Messy Genetic Algorithms

Fast messy genetic algorithms are a class of genetic algorithms dealing with populations of variable length. It is developed by Goldberg, Deb and Kargupta [30]. It contains three main phases:

- The initialization phase: It deals with defining a sizing equation. More precisely, it determines to which size or length we should extend the decoders in order to make them all having the same length. The sizing equation must overcome what is called the noise problem. In our case, the noise problem is generating logical length and crossing point so that in the next phase, we are able to generate logical building blocks.

- The Building Block Filtering (BBF) phase: It produces the genes or building blocks so that they respect the sizing equation. It represents an important phase since it generates logical building blocks in order to ensure that the pattern or signature extraction produces logical output. Moreover, it ensures that we can take the maximum number of genes that could be taken from each gene to make the deletion bits very low for each decoder regardless of their lengths.

- The juxtaposition phase: In this phase, we take the building blocks produced from the previous phase and combine them together to form the final decoders.

These are the main phases that are added to the genetic algorithm when dealing with a population composed of chromosomes having different length.

4.5.1 Initialization Phase

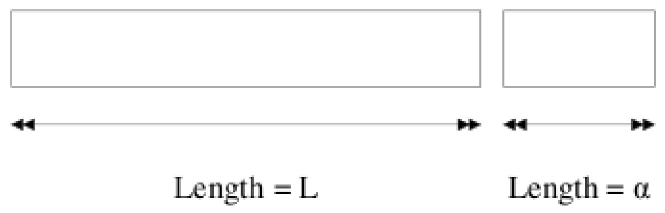
Our initialization phase consists of deriving the sizing equation. The process goes through the following phases:

- Extract the minimum length decoder and the maximum length decoder.
- Check if the minimum length is a prime number:
  - If true, then the minimum length is the crossing point
  - Otherwise, the user is asked to input his crossing length. The crossing length that is entered must be a multiple of the minimum length. In other words, $minimum\_length$ ***mod***[5] $crossing\_length = 0$.
- Generate the length to which all decoders extend to. This length is the next number greater than the maximum length decoder such that it is a multiple of the crossing point.

Executing this phase generates logical building blocks. To illustrate the idea, let take the following example. Assume we have a decoder of length l and we want to extend the decoder to a length that is not a multiple of the crossing point length. Thus, we have to extend the decoder by a length α as illustrated in Figure 4.11.

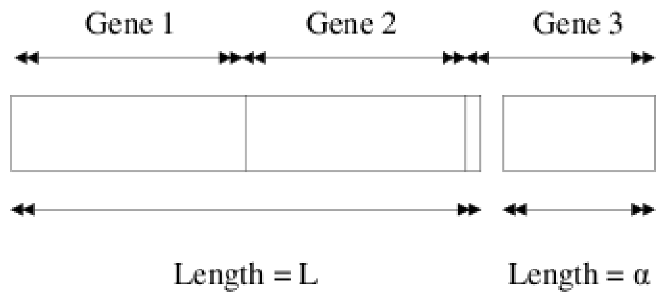
[5]mod: modulo

Figure 4.11: Extended Decoder



If $1 + \alpha$ is not a multiple of the crossing point length, then we end up with non-logical building blocks as presented in Figure 4.12 .

Figure 4.12: Non-Logical Building Blocks



We can see that Gene 3 does not make sense. The extended block is added to an original portion of the decoder. Thus, in terms of bytes or shellcode, Gene 3 does not make any sense since the bytes are mixed up with one other. Consequently, we must choose the length of reference to which all decoders will extend to in a way to avoid such non-logical building blocks. After identifying the maximum length in the pool of decoders, we choose the length of reference to be the next greatest number after the maximum such that it is a multiple of the crossing point. As an example, if we know the maximum length is 62 and the crossing point 6, then the length of reference becomes 66. At this stage, we can still get non-logical building blocks. Our next phase definitely ensures the production of logical building blocks for each decoder.

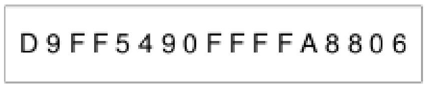
4.5.2 Building Block Filtering and Juxtaposition

From the sizing equation or initialization, we came out with two important parameters: the crossing point length N and the length to which the decoders must be extended to denoted by L. In order to ensure the selection of logical building blocks, this phase goes through the following processes:

- Create, for each decoder, a pool of indexes that refers to each bit position of the decoder, let call it POOL such that POOL $= \{p_1, p_2, p_3, \ldots, p_n\}$ where p_n refers to a bit position of the decoder.
- Initialize a variable called le to zero, then while($le \times N < L$):
  - Pick a random index from POOL.
    - If $index + cross\_length < length$ of the decoder, create a building block of length N having the selected index as a starting point and add it to a pool of genes called GENES. Delete the selected index from POOL and increment le.
    - Otherwise, select another index.
  - Repeat the process until $le \times N > L$ (until a decoder of size L is obtained).
- Repeat the process until we traverse the whole population.

At the end, we obtain a decoder that is not be affected by the crossing-point problem that we have described previously. Moreover, since for each decoder we are trying to reach a length greater than the maximum length decoder, the rate of bit deletion is very low. Let us illustrate the process by the following example. We assume we have a population of decoders such that the maximum length decoder is 25 and the minimum length decoder is 16. Our crossing point is $N = 8$. Thus, $L = 32$, which represents the next greater number than 25 such that it is multiple of N. Let us see how the process goes through a decoder D_1 of length 17 as illustrated in Figure 4.13 .

Figure 4.13: Initial Decoder



D9FF5490FFFFA8806

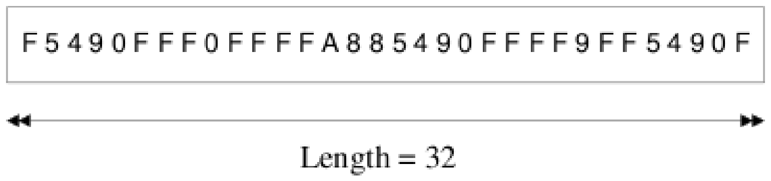
In this case, we have: POOL $= \{0, 1, 2, 3, ..., 16\}$. We proceed with the Building Block Filtering (BBF), $le = 0$ and loop until $le \times N > L$. The following Table 4.3 details the genes extraction:

Table 4.3: Genes Extraction

Index Selected	Verification	Gene Extracted	Genes' Pool
Index 3	$3 + N = 11 < 17$	$B_1 = F5490FFF$	B_1
Index 11	$11 + N = 19 > 17$	None	B_1
Index 7	$7 + N = 15 < 17$	$B_2 = 0FFFFA88$	B_1, B_2
Index 4	$4 + N = 12 < 17$	$B_3 = 5490FFFF$	B_1, B_2, B_3
Index 1	$1 + N = 9 < 17$	$B_4 = 9FF5490F$	B_1, B_2, B_3, B_4

At the end, we end up with the following decoder shown by Figure 4.14 .

Figure 4.14: Final Decoder



We can see that the decoder contains logical building blocks and are not be affected by the N-point crossing over. The main reason behind this is the fact that the crossing point is always in a bit position that generates logical building blocks (since the building blocks are multiple of the crossing point length).

The juxtaposition phase is embedded within the BBF phase since it only consists of appending the genes between them to form a new decoder.

4.5.3 Filtering

The filtering takes place before the BBF process. We describe first the BBF phase because of its importance even though the filtering happens before. In this respect, filtering comes when the length of a decoder is not long enough to reach the length L derived from the sizing equation. Thus such decoders are eliminated from the initial population. This operation of elimination constitutes the filtering part of the FMGA [30].

4.5.4 Engine

At the end of the FMGA process [30], we end up with a population of decoders having the same length, the process of pattern and signature extraction remains the same. Having decoders of the same length, the process described in previous sections for pattern extraction takes place.

4.6 Results

We test the engine against the three polymorphic engines; one engine generating fixed length decoders and the two others generating changing-length decoders. Shikata Ga Nai is our choice for the fixed length decoder engine , ADMmutate and Tapion are our choice for changing-length decoders. For the Shikata Ga Nai engine, we got the following patterns at the end of the engine:

Table 4.4: Extracted Patterns for Shikata Ga Nai

Patterns
Xd9742
d97424
97424f
7424f4
XXb116
XXd974
424f4X
24f4XX
Xb116X
b116XX

For Tapion engine, we get the following patterns at the end of the engine:

Table 4.5: Extracted Patterns for Tapion

Patterns
X9fXd9fXd9
Xd9fXd9fXd
d9fXd9fXdX
9fXd9fXdXe
9fXd9fXd9X
d9fXdXfXd9
d9fXd9XXd9
d9fXd9fXd9
9fXd9XXd9f
d9fXdXeXd9
Xd9eXd9fXd
d9eXd9fXdX
fXd9fXd9fX
XfXd9fXd9f
dXfXd9fXd9
d9XXd9fXd9
d9fXX9fXd9
dXeXd9fXd9
9XXd9fXd9f
9fXdXfXd9f
d9fXd9fXX9
9fXd9fXdXf
XXX9bdbe3d
0f8fXXfXff
f8fXXfXfff
8fXXfXffff

For ADMmutate, we get the following results:

Table 4.6: Extracted Patterns for ADMmutate

Patterns
XX81c3
X81c3X
81c3XX
83c601
XX9640

Table 4.6 – part II

Patterns
X96409
964096
4096XX
64096X
83cXX1
8Xc6X1
XX83c6
X83c60
3c601X
e8XXff
964XX6
c601XX
XX4096
9X4X96
X4096X
964X9X
X64X96
X83cX0
83cX0X
X9X409
9X409X
96XX96
X964X9
964X96
64X96X
8XXfff
XXffff
X9640X
9640XX
96X0X6
Xfffff
X96X09
96X09X
83cX01
83XX01
XX6409
X64096
6409XX
ffffff
8XcX01

We can see from these results that we need another filtering for the patterns. As an example, let take the results from Shikata Ga Nai. We get: “d97424”, “7424f4” and “97424f”. In fact, those patterns correspond to one long signature which is : “d97424f4”. We see that the signatures need to be merged. Another problem that we observe for the signatures is the similarities between them. Let us take the case of ADMmutate, we can see the following patterns: “964096”, “964X9X” and “9X4X96”. In fact, they somehow correspond to the same signature. Consequently, they must be reduced to one signature corresponding to “964096”. At the end, we need to implement an automated mechanism in order to deal with such similarities and then filter the patterns. Fortunaltely, the *Levensthein distance* [31] can help us in the matter as explained hereafter.

4.7 Filtering

The filtering process goes through three main phases, but before explaining them in detail, we dedicate the next section for introducing briefly the concept of *Levensthein distance* [31]

4.7.1 Levensthein Distance

The *Levensthein distance* is used to compute the amount of difference between two strings. It consists of giving the minimum number of operations needed to transform one string to another. The operations consist of insertion, deletion and substitution of a single character.The concept was developed by Vladimir Levensthein and it is often used to measure or determine how similar are two strings.

4.7.2 Filtering Process

The filtering process goes through three main phases. The phases are the following:

- Locate patterns having the less number of wildcard characters and compute the *levensthein distance* against all other patterns:
  - If the distance is less or equal to 2 then probably the two strings are very similar (in the form "964096" and "964X9X"). The signature that is checked against is deleted.
  - Otherwise, it means that the two patterns are completely different and thus kept.
- Take the input from the last operation and merge signatures if they can be merged.
- Take the input from the merging and take each single signature and compare it with all the others. We check the number of common bits at the same position without taking into consideration the wildcard characters X.
  - If the number is greater or equal than $\frac{2}{3}$ of the string, then take the one with less wildcards and drop the other.
- Take the input from the three level of filtering and repeat again the three filtering phases. If from one round to another, the number of signatures remains the same then the process is stopped.

After the engine generated the patterns (see previous Section 4.6), we apply to the output the filtering process in order to extract the final *Shape Signatures*. We get the following signatures at the end presented in Table 4.7.

Table 4.7: Final Shape Signatures for Polymorphic Engines

Shikata Ga Nai	Tapion	ADMmutate
XXd97424f4XX	f8fXXfXffff	e8XXfff
XXb116XX	XfXd9fXd9fX	Xffffff
	XXX9bdbe3d	XX964096XX
		XX83c601XX
		XX81c3XX

4.8 Validation

The results we got previously were extracted from a single set of 1000 decoders. In order to validate the results, we need to test the signatures against the sets that we have generated. As a reminder, we have generated 3 new sets of each 1000 decoders, 3 new sets of each 5000 decoders and 3 new sets of each 10000 decoders for each polymorphic engine. The validation consists of analyzing each decoders from each set and check if the shape signatures that we have extracted exist. We then provide the rate of detection for each set. At the end, we average the various rates from each set to give the final rate of detection for each signature. The detection rates for the extracted signature applied on Shikata Ga Nai polymorphic engine and the corresponding average rates are presented in Table 4.8 and Table 4.9 respectively.

Table 4.8: Detection for Shikata Ga Nai Signatures

Sets	Signatures	Rate of Detection
1000_1	XXd97424f4XX	100 %
	XXb116XX	0 %
1000_2	XXd97424f4XX	100 %
	XXb116XX	0 %
1000_3	XXd97424f4XX	100 %
	XXb116XX	0 %
5000_1	XXd97424f4XX	100 %
	XXb116XX	0 %
5000_2	XXd97424f4XX	100 %
	XXb116XX	0 %
5000_3	XXd97424f4XX	100 %
	XXb116XX	0 %
10000_1	XXd97424f4XX	100 %
	XXb116XX	0 %
10000_2	XXd97424f4XX	100 %
	XXb116XX	0 %
10000_3	XXd97424f4XX	100 %
	XXb116XX	0 %

Table 4.9: Average Detection for Shikata Ga Nai Signatures

Signatures	Rate of Detection
XXd97424f4XX	100 %
XXb116XX	0 %

The detection rates for the extracted signatures applied on ADMmutate polymorphic engine and the corresponding averages rates are presented in Table 4.10 and Table 4.11 respectively.

Table 4.10: Detection for ADMmutate Signatures

Sets	Signatures	Rate of Detection
1000_1	XX81c3XX	99.4 %
	XX83c601XX	80.2 %
	XX964096XX	80.2 %
	Xffffff	98.88 %
	e8XXfff	98.88 %
1000_2	XX81c3XX	98.7 %
	XX83c601XX	78.7 %
	XX964096XX	80 %
	Xffffff	98 %
	e8XXfff	98 %
1000_3	XX81c3XX	99.5 %
	XX83c601XX	78.25 %
	XX964096XX	77.4 %
	Xffffff	99 %
	e8XXfff	99 %
5000_1	XX81c3XX	99.2 %
	XX83c601XX	78.9 %
	XX964096XX	79.5 %
	Xffffff	98.5 %
	e8XXfff	98.5 %
5000_2	XX81c3XX	99 %
	XX83c601XX	79.3 %
	XX964096XX	78.25 %
	Xffffff	98 %
	e8XXfff	98 %
5000_3	XX81c3XX	99.16 %
	XX83c601XX	79 %
	XX964096XX	79 %
	Xffffff	98.14 %
	e8XXfff	98.14 %
10000_1	XX81c3XX	99.16 %
	XX83c601XX	78.6 %
	XX964096XX	79 %
	Xffffff	98.24 %
	e8XXfff	98.24 %
10000_2	XX81c3XX	99 %
	XX83c601XX	78.54 %
	XX964096XX	79.7 %
	Xffffff	98.46 %
	e8XXfff	98.46 %
10000_3	XX81c3XX	99.1 %
	XX83c601XX	78.8 %
	XX964096XX	79 %

Continued on next page

Table 4.10 – continued from previous page

Sets	Signatures	Rate of Detection
	Xffffff	98.43 %
	e8XXfff	98.43 %

Table 4.11: Average Detection for ADMmutate Signatures

Signatures	Rate of Detection
XX81c3XX	99.13 %
XX83c601XX	79 %
XX964096XX	79.11 %
Xffffff	98.4 %
e8XXfff	98.4 %

The detection rates for the extracted signatures applied on Tapion polymorphic engine and the corresponding average rates are presented in Table 4.12 and Table 4.13 respectively.

Table 4.12: Detection for Tapion Signatures

Sets	Signatures	Rate of Detection
1000_1	XXX9bdbe3d	98.7 %
	XfXd9fXd9fX	76 %
	f8fXXfXffff	100 %
1000_2	XXX9bdbe3d	98.6 %
	XfXd9fXd9fX	78 %
	f8fXXfXffff	100 %
1000_3	XXX9bdbe3d	98.5 %
	XfXd9fXd9fX	74.7 %
	f8fXXfXffff	100 %
5000_1	XXX9bdbe3d	98.54 %
	XfXd9fXd9fX	76.4 %
	f8fXXfXffff	100 %
5000_2	XXX9bdbe3d	98.56 %
	XfXd9fXd9fX	76.36 %
	f8fXXfXffff	100 %
5000_3	XXX9bdbe3d	98.4 %
	XfXd9fXd9fX	77.24 %
	f8fXXfXffff	100 %
10000_1	XXX9bdbe3d	98.63 %
	XfXd9fXd9fX	76.79 %
	f8fXXfXffff	100 %
10000_2	XXX9bdbe3d	98.47 %
	XfXd9fXd9fX	76.31 %
	f8fXXfXffff	100 %
10000_3	XXX9bdbe3d	98.59 %
	XfXd9fXd9fX	75.47 %
	f8fXXfXffff	100 %

Table 4.13: Average Detection for Tapion Signatures

Signatures	Rate of Detection
XXX9bdbe3d	98.55 %
XfXd9fXd9fX	76.36 %
f8fXXfXffff	100 %

4.9 Interpretation

The average rates of detection we get against the three polymorphic engine are very promising. Even when tested against all the sets, the signatures produced are valid. The only exception is the signature “XXb116XX”, which must be dropped, generated for Shikata Ga Nai which has a 0% rate of detection. But, overall the results prove that the extracted shape signature holds since we get high average rates of detection for all engines with 76.36% being the lowest rate. This confirms that the GA engine is very effective and that our concept of Shape Signature holds. We are able to locate invariant parts of polymorphic shellcode as well as mutated parts and incorporate them into a signature.

4.10 Conclusion

We can conclude by saying that genetic algorithms are very efficient in detecting patterns of polymorphic engines. The Schema Theorem [29] holds since we are able to effectively extract reliable signatures. Our concept of shape signature suits polymorphic engines. The ability to detect constant parts as well as mutated part is able to effectively cope with the polymorphic nature of such engines. Our assumption that shape signatures are able to produce above average schemata is verified . The results show that, as opposed with traditional models of signatures, which require a huge number of sequences to cope with polymorphism according to Song et al. [3] (in the order of $O(2^{8n})$ with n being the length of the decoder), our model generates a very limited number of signatures. The average detection resulting from the experiments shows that the shape signatures produces very interessting rates. The rest of the thesis is dedicated to modeling polymorphic engines. We show that the decoders generated from those engines do not look as random as they claim to be. In fact, we can consistently predict the location of the shape signature as well as areas of high mutation allowing us to map their decoders. Moreover, those engines possesse unique compositions, in terms of bytes, that can help us characterize them uniquely.

Chapter 5

Analysis of Shape Signature Positions

In this Chapter, we explain how decoders generated from polymorphic engines can be mapped. By mapping, we mean that we can consistently predict the areas of high mutation as well as the areas containing the shape signatures. Our genetic algorithm engine for extracting shape signatures shows some promising results. Usually, highly polymorphic engines make those signatures appear at different bit positions using sliding techniques. Moreover, they also have the ability to change their length, making the change of positions even more effective. However, the constraint of consistency makes the sliding or changing of bit positions not as random as it might appear. In other words, the shape signatures extracted cannot appear just at any bit positions otherwise the consistency of the decoder is compromised. In other words, even when changing its length, an engine must respect the structure of assembly language. The ability of some engines to change their length might make those signatures appear at very different positions. However, even when the length of the decoder changes, it must appear at positions that do not break the consistency of the decoder with respect to the length of the decoder. By analyzing each time the bit positions relatively to the length of decoder, we might be able to model those polymorphic engines. In fact, we believe that the signatures do not cover a large space of the decoder (in terms of bit positions) as it is claimed rather, they have a small area in which they shift positions.

5.1 Approach

Basically, we can differentiate two types of polymorphic engines: fixed-length engines e.g. Shikata Ga Nai and changing-length engines e.g. ADMmutate. The challenge is to define a model for analyzing the starting and ending positions for the extracted signatures for both models. In order to do this, we define what we call relative positions. Let us assume we have a decoder of length L. The signature starts at a bit position s and ends at a bit position e. Then, the relative start is: $\frac{s}{L} \times 100$ and the relative end is $\frac{e}{L} \times 100$. In order to analyze the shape signature positions, we go through the following steps:

- For each decoder for a specific engine, we record the relative start and the relative end for each shape signature found.
- We compute the mean relative start and the mean relative end for each signature.
- Then, we compute the standard deviation for the recorded relative starts and relative ends to quantify the gap between the measurements.
- Finally, we average the results for each measurements to get the final mean relative start, mean relative end, standard deviation for the relative starts and relative ends.

The process is applied for each set of population that is generated for each single polymorphic engine. The average results computed in the last steps allow computing the final mean relative start, the final mean relative end, the final standard deviation for relative starting positions and the final standard deviation for relative ending positions.

5.2 Results

In this section, we present the results for the three polymorphic engines. The following notations are used in the tables summarizing the obtained results:

- MRS: Mean Relative Start.
- MRE: Mean Relative End.
- STD_DEV MRS: Standard Deviation for Mean Relative Start.
- STD_DEV MRS: Standard Deviation for Mean Relative End.

The shape signature position analysis for the extracted signatures applied on Shikata Ga Nai polymorphic engine and the corresponding average shape signature position analysis are presented in Table 5.1 and Table 5.2 respectively.

Table 5.1: Shape Signature Position Analysis for Shikata Ga Nai

Sets	Signatures	MRS	MRE	STD_DEV MRS	STD_DEV MRE
1000_1	XXd97424f4XX	27.825	48.196	12.247	12.247
1000_2	XXd97424f4XX	27.492	47.862	12.238	12.238
1000_3	XXd97424f4XX	27.177	47.548	12.119	12.119
5000_1	XXd97424f4XX	27.408	47.779	12.158	12.158
5000_2	XXd97424f4XX	27.666	48.037	12.112	12.112
5000_3	XXd97424f4XX	27.914	48.284	11.997	11.997
10000_1	XXd97424f4XX	28.063	48.434	11.915	11.915
10000_2	XXd97424f4XX	27.843	48.213	11.964	11.964
10000_3	XXd97424f4XX	27.728	48.098	12.104	12.104

Table 5.2: Average Shape Signature Position Analysis for Shikata Ga Nai

Signatures	MRS	MRE	STD_DEV MRS	STD_DEV MRE
XXd97424f4XX	27.679	48.05	12.094	12.094

The shape signature position analysis for the extracted signatures applied on Tapion polymorphic engine and the corresponding average shape signature position analysis are presented in Table 5.3 and Table 5.4 respectively.

Table 5.3: Shape Signature Position Analysis for Tapion

Sets	Signatures	MRS	MRE	STD_DEV MRS	STD_DEV MRE
1000_1	f8fXXfXffff	97.836	98.92	0.61	0.507
	XfXd9fXd9fX	45.466	46.537	28.866	28.889
	XXX9bdbe3d	1.5	2.482	0.845	0.911
1000_2	f8fXXfXffff	97.806	98.909	0.622	0.522
	XfXd9fXd9fX	43.611	44.707	28.832	28.858
	XXX9bdbe3d	1.508	2.505	0.83	0.897
1000_3	f8fXXfXffff	97.755	98.816	0.651	0.541
	XfXd9fXd9fX	46.945	48.039	28.622	28.638
	XXX9bdbe3d	1.546	2.548	0.938	1.014
5000_1	f8fXXfXffff	97.818	98.915	0.626	0.533
	XfXd9fXd9fX	45.036	46.123	28.685	28.701
	XXX9bdbe3d	1.517	2.51	0.916	0.982
5000_2	f8fXXfXffff	97.793	98.892	0.633	0.534
	XfXd9fXd9fX	44.439	45.531	28.884	28.901
	XXX9bdbe3d	1.507	2.502	0.909	0.971
5000_3	f8fXXfXffff	97.811	98.911	0.623	0.528
	XfXd9fXd9fX	45.041	46.13	29.001	29.022
	XXX9bdbe3d	1.528	2.523	0.882	0.95
10000_1	f8fXXfXffff	97.812	98.911	0.625	0.534
	XfXd9fXd9fX	44.694	45.783	29.133	29.15
	XXX9bdbe3d	1.533	2.526	0.878	0.945
10000_2	f8fXXfXffff	97.815	98.91	0.624	0.528
	XfXd9fXd9fX	45.345	46.433	29.231	29.246
	XXX9bdbe3d	1.518	2.51	0.879	0.944
10000_3	f8fXXfXffff	97.798	98.898	0.625	0.533
	XfXd9fXd9fX	45.083	46.173	28.887	28.905
	XXX9bdbe3d	1.537	2.533	0.887	0.952

Table 5.4: Average Shape Signature Position Analysis for Tapion

Signatures	MRS	MRE	STD_DEV MRS	STD_DEV MRE
f8fXXfXffff	97.805	98.902	0.626	0.526
XfXd9fXd9fX	45.073	46.162	28.904	28.923
XXX9bdbe3d	1.521	2.515	0.885	0.951

The shape signature position analysis for the extracted signatures applied on ADMmutate polymorphic engine and the corresponding average shape signature position analysis are presented in Table 5.5 and Table 5.6 respectively.

Table 5.5: Shape Signature Position Analysis for ADMmutate

Sets	Signatures	MRS	MRE	STD_DEV MRS	STD_DEV MRE
1000_1	XX81c3XX	46.03	50.116	7.277	7.353
	XX83c601XX	62.158	67.377	9.172	9.175
	XX964996XX	62.285	67.503	9.047	9.03
	Xffffff	96.47	99.957	0.499	0.154
	e8XXfff	94.769	98.256	0.695	0.232
1000_2	XX81c3XX	46.146	50.284	7.778	7.946
	XX83c601XX	61.723	66.981	9.028	9.026
	XX964996XX	62.475	67.706	8.408	8.405
	Xffffff	96.443	99.96	0.495	0.151
	e8XXfff	94.725	98.242	0.686	0.228
1000_3	XX81c3XX	46.086	50.198	7.24	7.298
	XX83c601XX	62.194	67.437	8.617	8.581
	XX964996XX	62.45	67.677	8.969	8.963
	Xffffff	96.447	99.953	0.502	0.16
	e8XXfff	94.74	98.246	0.697	0.232
5000_1	XX81c3XX	46.379	50.475	7.451	7.613
	XX83c601XX	62.374	67.573	8.768	8.768
	XX964996XX	62.372	67.564	8.688	8.695
	Xffffff	96.485	99.966	0.481	0.134
	e8XXfff	94.777	98.259	0.692	0.231
5000_2	XX81c3XX	46.271	50.385	7.678	7.862
	XX83c601XX	62.226	67.434	8.749	8.746
	XX964996XX	62.304	67.527	8.797	8.799
	Xffffff	96.47	99.965	0.489	0.14
	e8XXfff	94.758	98.253	0.695	0.232
5000_3	XX81c3XX	46.279	50.395	7.872	8.062
	XX83c601XX	62.522	67.747	8.798	8.798
	XX964996XX	62.395	67.604	8.81	8.808
	Xffffff	96.472	99.966	0.485	0.137

Continued on next page

Table 5.5 – continued from previous page

Sets	Signatures	MRS	MRE	STD_DEV MRS	STD_DEV MRE
	e8XXfff	94.759	98.253	0.691	0.23
10000_1	XX81c3XX	46.267	50.382	7.819	8.001
	XX83c601XX	62.335	67.554	8.887	8.883
	XX964996XX	62.259	67.483	8.807	8.818
	Xffffff	96.467	99.962	0.499	0.145
	e8XXfff	94.757	98.252	0.705	0.235
10000_2	XX81c3XX	46.057	50.157	7.31	7.402
	XX83c601XX	62.271	67.498	8.83	8.834
	XX964996XX	62.359	67.583	8.869	8.875
	Xffffff	96.468	99.964	0.492	0.14
	e8XXfff	94.756	98.252	0.704	0.235
10000_3	XX81c3XX	46.301	50.41	7.421	7.543
	XX83c601XX	62.563	67.788	8.79	8.789
	XX964996XX	62.509	67.736	8.774	8.783
	Xffffff	96.465	99.963	0.489	0.143
	e8XXfff	94.752	98.251	0.695	0.232

Table 5.6: Average Shape Signature Position Analysis for ADMmutate

Signatures	MRS	MRE	STD_DEV MRS	STD_DEV MRE
XX81c3XX	46.2	50.311	7.54	7.675
XX83c601XX	62.263	67.487	8.849	8.844
XX964996XX	62.378	67.598	8.796	8.797
Xffffff	96.465	99.962	0.492	0.145
e8XXfff	94.754	98.251	0.695	0.232

5.3 Interpretation

As we can see for the three engines, the relative positions for both start and end for a signature are very similar for each set of population. The same applies for their standard deviations. We can deduce that the area where the signatures appear in average seems to be the same for each set of population. The standard deviation shows us the differences in terms of positions for each signature. In other words, it shows us the area of shift for each shape signature. Consequently, we can see that, in reality, the shape signatures do not cover the whole dimensions of a decoder (in terms of bits positions). They rather cover a small area in which they shift positions. The changing length ability of some engines might seem to make those engine covering a large area of the decoders. However, if we look at the relative positions with respect to the decoder length, we see that it is not the case. Based on those results, we can confirm our assumption that the constraint of consistency of the decoder, is an important factor that allows to effectively map the structure of a polymorphic decoder. Based on this, we can map the decoders for each of the polymorphic engines by locating :

- The shape signatures
- Their area of shift
- The zone of high mutations of the decoder
- The potential zone of mutations

The following Figures 5.1, 5.2 and 5.3 summarize the analysis:

Figure 5.1: ADM Mapping

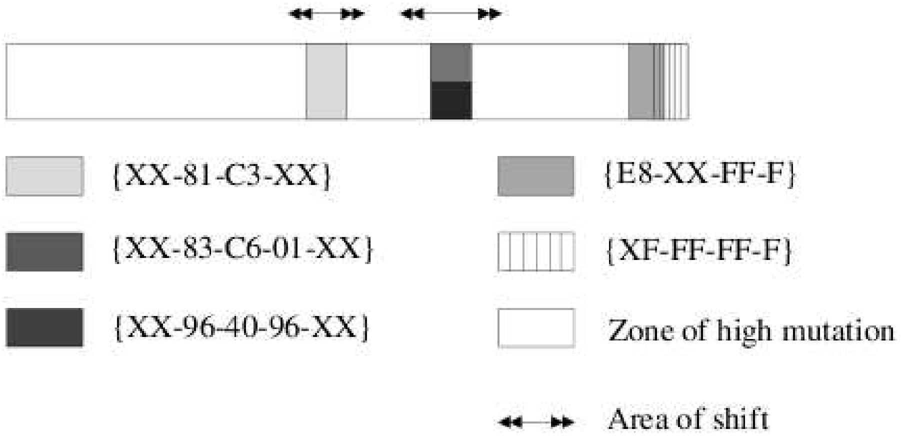


Figure 5.2: Tapion Mapping

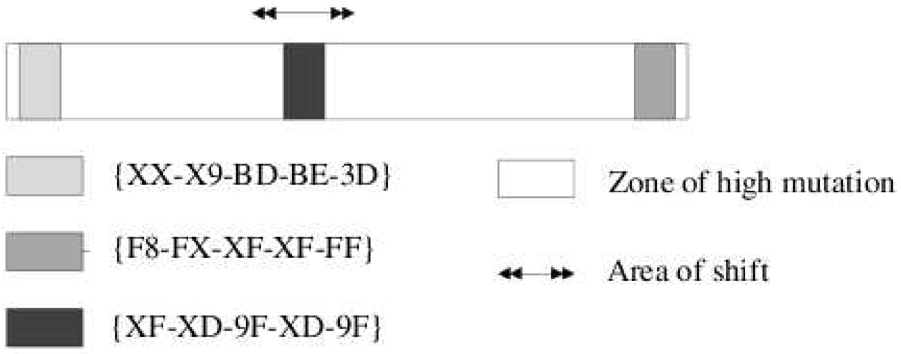
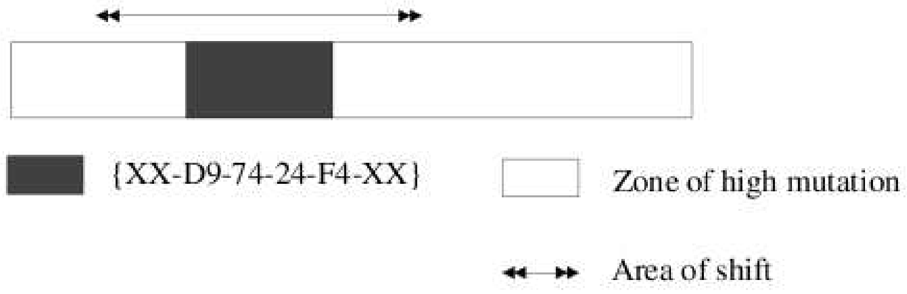


Figure 5.3: Shikata Ga Nai Mapping



5.4 Verification

In this section, we verify whether our conlusions in terms of shape signatures bit positions hold or not. For that purpose, we try to identify if there is a correlation between the detection rate of each signature for each polymorphic engine and the detection rate of the same signatures by taking into account their position in the decoder. We aim at verifying their actual location based on the previous results, especially the average shape signature position analysis value. If the correlation between the two rates is the same, then we can safely confirm that our modeling holds and that the polymorphic engines effectively comply with the consistency constraint, which restrain their shape signatures to shift with only a small extent within the decoder's dimension. We test each polymorphic engine for all the sets and then average the results at the end.

The detection for the extracted signatures with and without position verification applied on Shikata Ga Nai polymorphic engine is presented in Table 5.7.

Table 5.7: Detection for Shikata Ga Nai Signatures without and with Position Verification

Sets	Signatures	Detection	Detection with Position Verification
1000_1	XXd97424f4XX	100 %	100 %
1000_2	XXd97424f4XX	100 %	100 %
1000_3	XXd97424f4XX	100 %	100 %
5000_1	XXd97424f4XX	100 %	100 %
5000_2	XXd97424f4XX	100 %	100 %
5000_3	XXd97424f4XX	100 %	100 %
10000_1	XXd97424f4XX	100 %	100 %
10000_2	XXd97424f4XX	100 %	100 %
10000_3	XXd97424f4XX	100 %	100 %

The detection for the extracted signatures with and without position verification applied on Tapion polymorphic engine is presented in Table 5.8.

Table 5.8: Detection for Tapion Signatures without and with Position Verification

Sets	Signatures	Detection	Detection with Position Verification
1000_1	XXX9bdbe3d	98.7 %	98.7 %
	XfXd9fXd9fX	76 %	76 %
	f8fXXfXffff	100 %	100 %
1000_2	XXX9bdbe3d	98.6 %	98.6 %
	XfXd9fXd9fX	78 %	78 %
	f8fXXfXffff	100 %	100 %
1000_3	XXX9bdbe3d	98.5 %	98.5 %
	XfXd9fXd9fX	74.7 %	74.7 %
	f8fXXfXffff	100 %	100 %
5000_1	XXX9bdbe3d	98.54 %	98.54 %
	XfXd9fXd9fX	76.4 %	76.4 %
	f8fXXfXffff	100 %	100 %
5000_2	XXX9bdbe3d	98.56 %	98.56 %
	XfXd9fXd9fX	76.36 %	76.36 %
	f8fXXfXffff	100 %	100 %
5000_3	XXX9bdbe3d	98.4 %	98.4 %
	XfXd9fXd9fX	77.24 %	77.24 %
	f8fXXfXffff	100 %	100 %
10000_1	XXX9bdbe3d	98.63 %	98.63 %
	XfXd9fXd9fX	76.79 %	76.79 %
	f8fXXfXffff	100 %	100 %
10000_2	XXX9bdbe3d	98.47 %	98.47 %
	XfXd9fXd9fX	76.31 %	76.31 %
	f8fXXfXffff	100 %	100 %
10000_3	XXX9bdbe3d	98.59 %	98.59 %
	XfXd9fXd9fX	75.47 %	75.47 %
	f8fXXfXffff	100 %	100 %

The detection for the extracted signatures with and without position verification applied on ADMmutate polymorphic engine is presented in Table 5.9.

Table 5.9: Detection for ADMmutate Signatures without and with Position Verification

Sets	Signatures	Detection	Detection with Position Verification
1000_1	XX81c3XX	99.4 %	99.4 %
	XX83c601XX	80.2 %	80.2 %
	XX964096XX	80.2 %	80.2 %
	Xffffff	98.88 %	98.88 %
	e8XXfff	98.88 %	98.88 %
1000_2	XX81c3XX	98.7 %	98.7 %
	XX83c601XX	78.7 %	78.7 %
	XX964096XX	80 %	80 %
	Xffffff	98 %	98 %
	e8XXfff	98 %	98 %
1000_3	XX81c3XX	99.5 %	99.5 %
	XX83c601XX	78.25 %	78.25 %
	XX964096XX	77.4 %	77.4 %
	Xffffff	99 %	99 %
	e8XXfff	99 %	99 %
5000_1	XX81c3XX	99.2 %	99.2 %
	XX83c601XX	78.9 %	78.9 %
	XX964096XX	79.5 %	79.5 %
	Xffffff	98.5 %	98.5 %
	e8XXfff	98.5 %	98.5 %
5000_2	XX81c3XX	99 %	99 %
	XX83c601XX	79.3 %	79.3 %
	XX964096XX	78.25 %	78.25 %
	Xffffff	98 %	98 %
	e8XXfff	98 %	98 %
5000_3	XX81c3XX	99.16 %	99.16 %
	XX83c601XX	79 %	79 %
	XX964096XX	79 %	79 %
	Xffffff	98.14 %	98.14 %
	e8XXfff	98.14 %	98.14 %
10000_1	XX81c3XX	99.16 %	99.16 %
	XX83c601XX	78.6 %	78.6 %
	XX964096XX	79 %	79 %
	Xffffff	98.24 %	98.24 %
	e8XXfff	98.24 %	98.24 %
10000_2	XX81c3XX	99 %	99 %
	XX83c601XX	78.54 %	78.54 %
	XX964096XX	79.7 %	79.7 %
	Xffffff	98.46 %	98.46 %
	e8XXfff	98.46 %	98.46 %
10000_3	XX81c3XX	99.1 %	99.1 %
	XX83c601XX	78.8 %	78.8 %

Continued on next page

Table 5.9 – continued from previous page

Sets	Signatures	Detection	Detection with Position Verification
	XX964096XX	79 %	79 %
	Xffffff	98.43 %	98.43 %
	e8XXfff	98.43 %	98.43 %

The average detection for the three polymorphic engines' shape signatures is summarized in the following Table 5.10:

Table 5.10: Average Detection for Engines without and with Position Verification

Engine	Signatures	Detection	Detection with Position Verification
Shikata Ga Nai	XXd97424f4XX	100 %	100 %
Tapion	XXX9bdbe3d	98.55 %	98.55 %
	XfXd9fXd9fX	76.36 %	76.36 %
	f8fXXfXffff	100 %	100 %
ADMmutate	XX81c3XX	99.13 %	99.13 %
	XX83c601XX	79 %	79 %
	XX964096XX	79.11 %	79.11 %
	Xffffff	98.4 %	98.4 %
	e8XXfff	98.4 %	98.4 %

5.5 Conclusion

According to the results, there is a strong correlation between the rate of signature detection and the rate of the same signatures detection by taking into consideration their expected location. This confirms that the mapping of polymorphic engines decoders holds. More generally, it confirms the previous assumption that polymorphic engines' shape signatures, do not explore the whole decoder's dimensions (in terms of bit positions) but rather shift in a defined area of the decoder. The main reason behind this is the constraint of consistency they must obey. In other words, the fact that the decoder obey to some semantics reduces the ability of shifting the bits in a random fashion. In the next chapter, we try to characterize even more polymorphic decoders by looking at their byte composition. The aim is to find out if we can differentiate those polymorphic decoders, from their compositions, from any other random sequence of bytes.

Chapter 6

Decoder Analysis Composition

In this Chapter, we try to improve the modeling of polymorphic engines by exploring their byte compositions. In the previous Chapter, we have successfully shown that it is possible to consistently predict shape signature locations and areas of high mutations making polymorphic shellcode decoder subject to consistency constraints. In this chapter, we explore the byte composition of the studied polymorphic engines. Additionally, we show that it is possible to characterize polymorphic shellcode from its byte composition.

6.1 Byte Spectrum Analysis

Before starting analyzing the decoder composition, we need to conduct some experiments in order to check the nature of bytes that are used in the three polymorphic engines' decoders. From the analysis, we can then start to develop a mechanism in order to analyze the composition of those engines. The experiment consists of recording, for each engine, the number of occurrences for each byte from $\{0 \times 00, \ldots, 0 \times FF\}$. We define, for each studied polymorphic engine, three samples: a population number of 1000 decoders, a population number of 5000 decoders and a population number of 10000 decoders. For each sample, we generate three different sets resulting in total of 9 samples. Finally, we conduct the experiments on each sample and record the results.

6.2 Results

Hereafter, we examine the results of byte spectrum analysis that we obtain for each engine. The x-axis refers to the byte spectrum $\{0 \times 00, \ldots, 0 \times FF\}$ while the y-axis examines the number of occurences of those bytes. The sets are named in the following manner: "Set A_B" where A refers to the population number and B to the set label $(1^{st}, 2^{nd}, \ldots)$. The byte occurences of each set for the Shikata Ga Nai byte spectrum are illustrated in the charts from Figure 6.1 to Figure 6.9.

Figure 6.1: Shikata Byte Spectrum for Set 1000_1

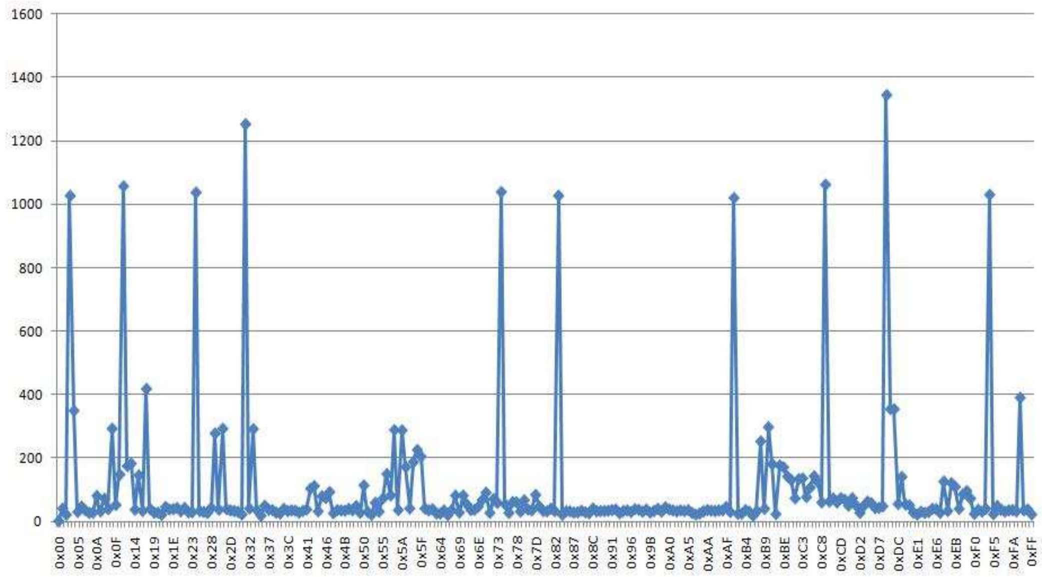


Figure 6.2: Shikata Byte Spectrum for Set 1000_2

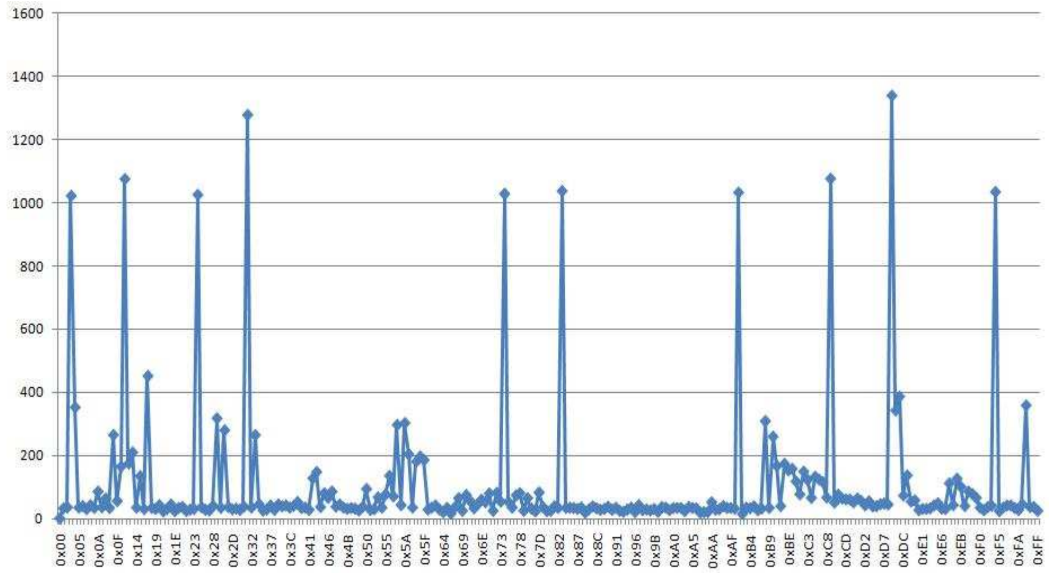


Figure 6.3: Shikata Byte Spectrum for Set 1000_3

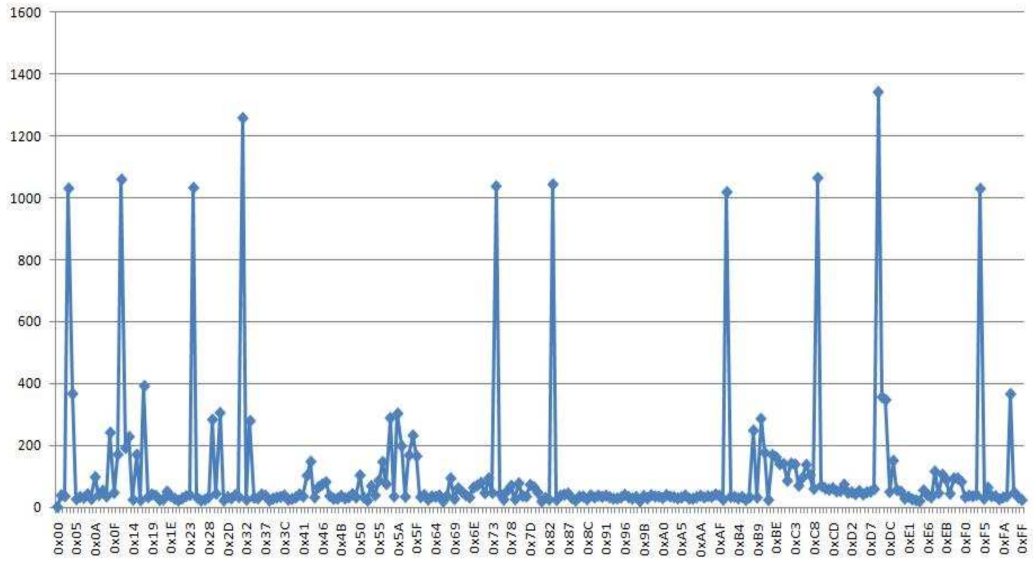


Figure 6.4: Shikata Byte Spectrum for Set 5000_1

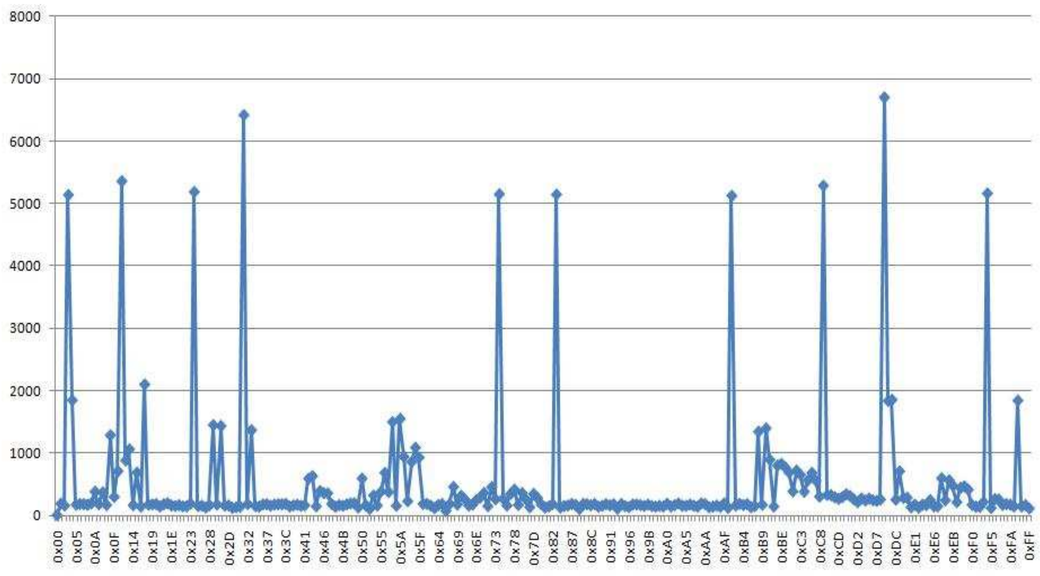


Figure 6.5: Shikata Byte Spectrum for Set 5000_2

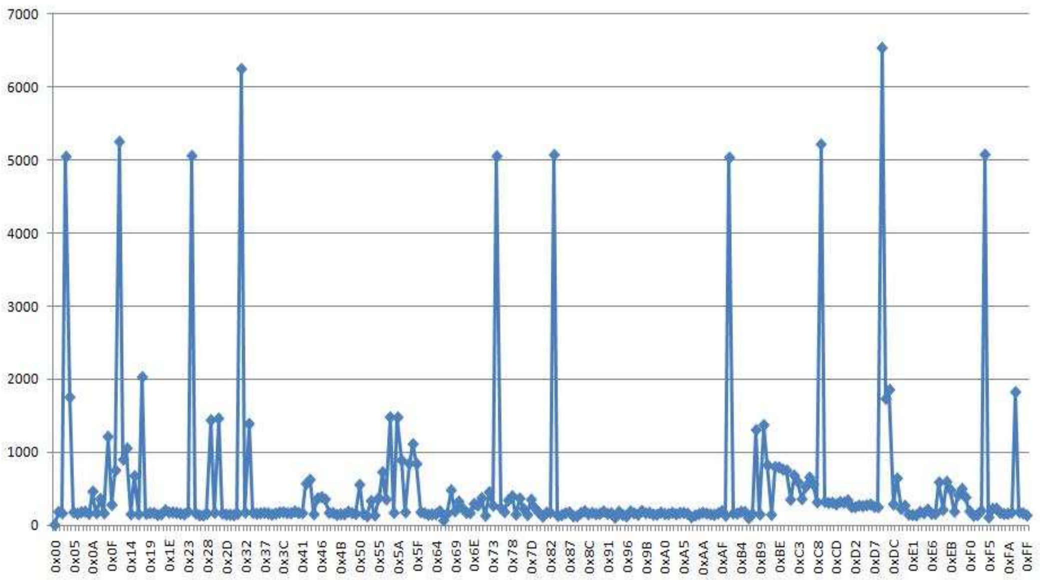


Figure 6.6: Shikata Byte Spectrum for Set 5000_3

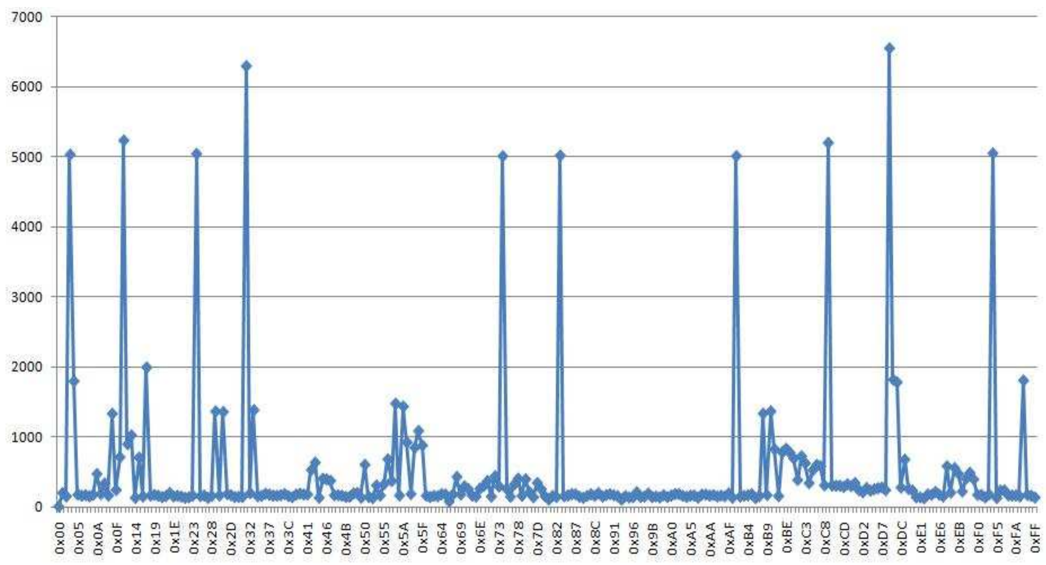


Figure 6.7: Shikata Byte Spectrum for Set 10000_1

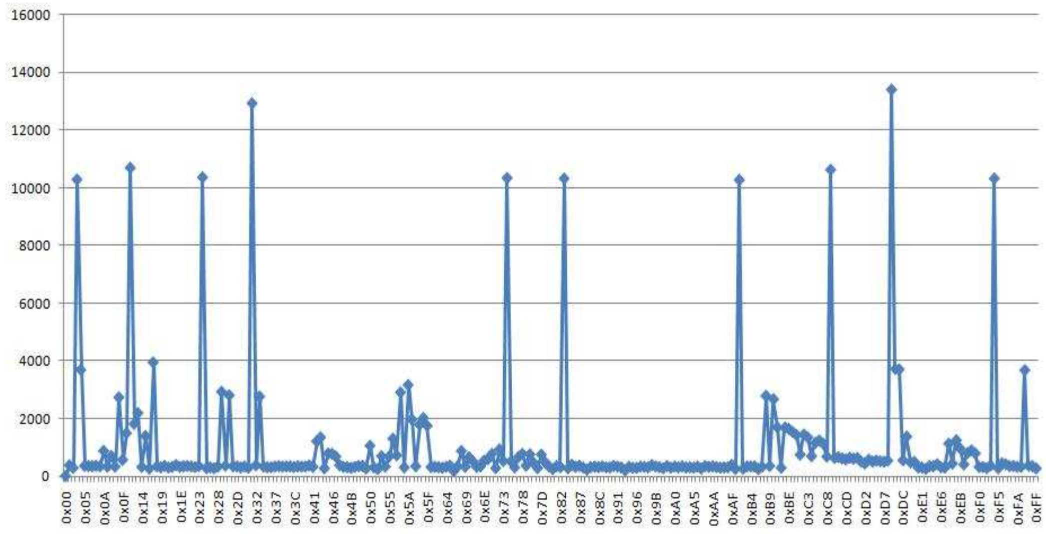


Figure 6.8: Shikata Byte Spectrum for Set 10000_2

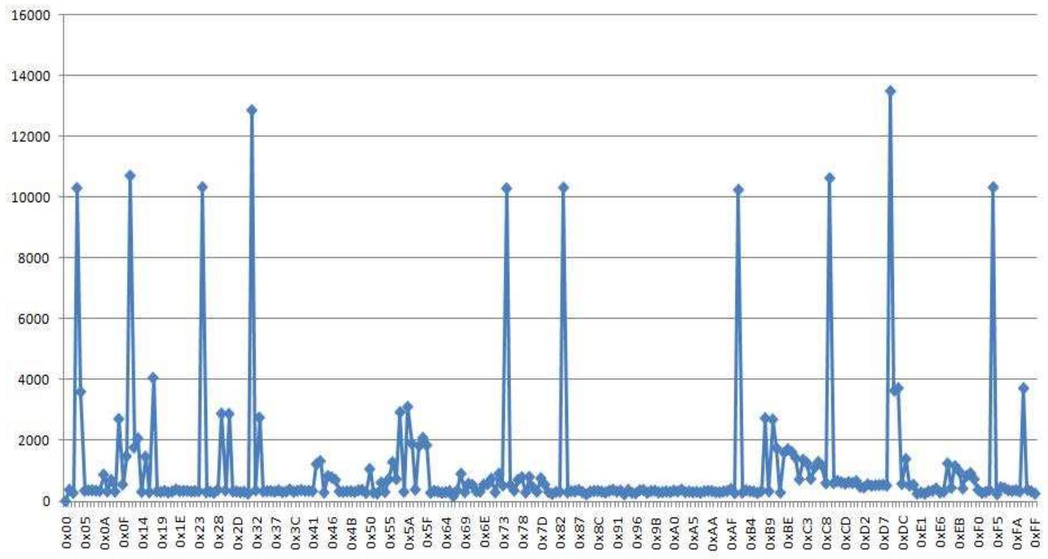
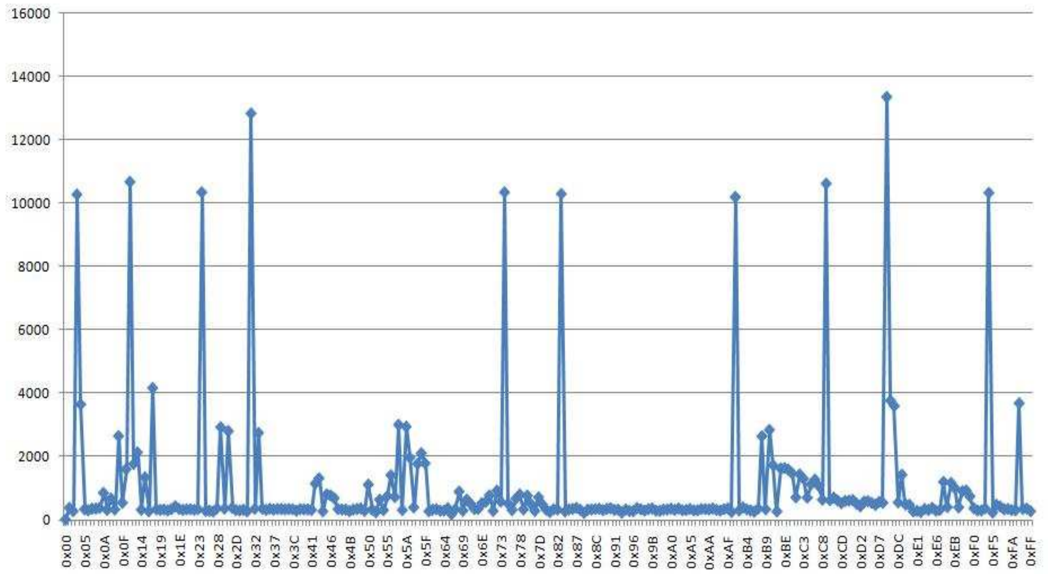


Figure 6.9: Shikata Byte Spectrum for Set 10000_3



The spectrums illustrated in Figure 6.1 to Figure 6.9 are very similar in their shapes, which shows that there is a pattern in terms of byte occurrences for Shikata Ga Nai. As expected, the whole byte spectrum is explored (with the exception of the null byte). Let us consider the most repeated bytes as the bytes that are greater or equal than the population number in terms of number of occurrences. One can notice that the same most repeated bytes are present across the various sets. Let see in the following whether the observations made for the Shikata Ga Nai are still valid for ADMmutate. The byte occurence of each set for the ADMmutate polymorphic engine are illustrated from Figure 6.10 to Figure 6.18 .

Figure 6.10: ADMmutate Byte Spectrum for Set 1000_1

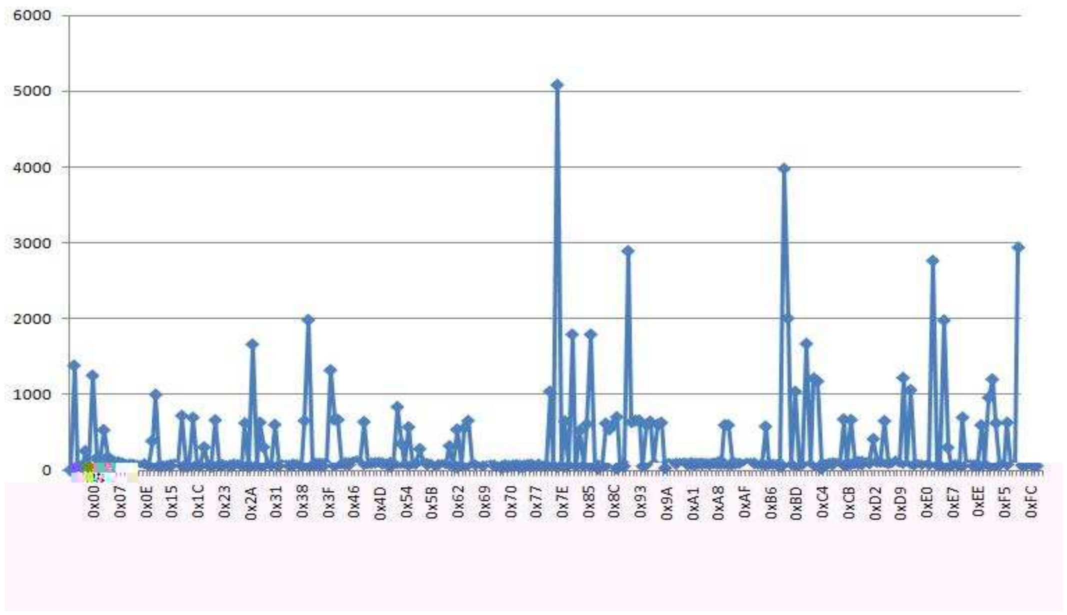


Figure 6.11: ADMmutate Byte Spectrum for Set 1000_2

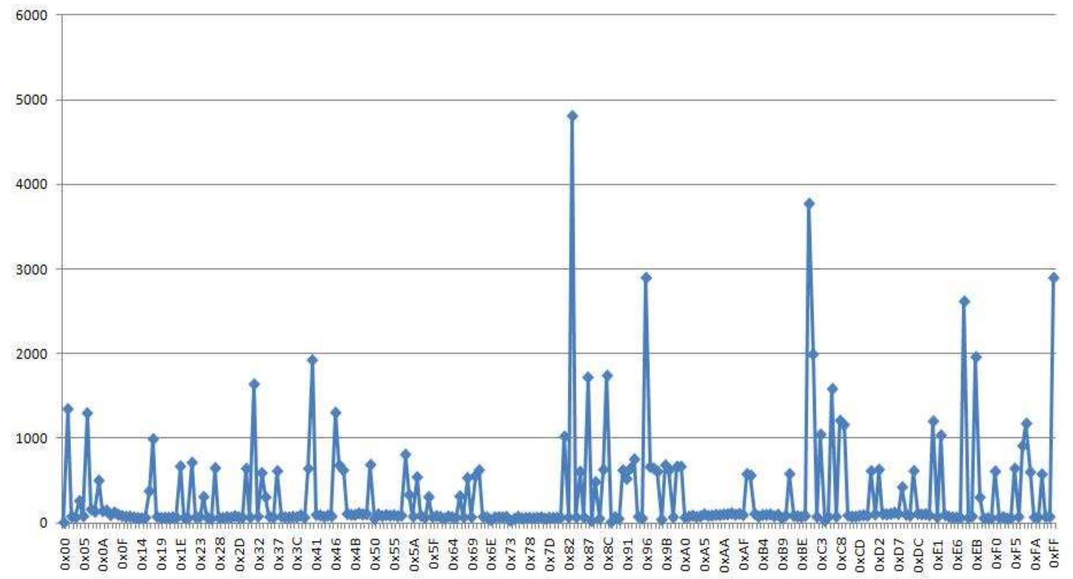


Figure 6.12: ADMmutate Byte Spectrum for Set 1000_3

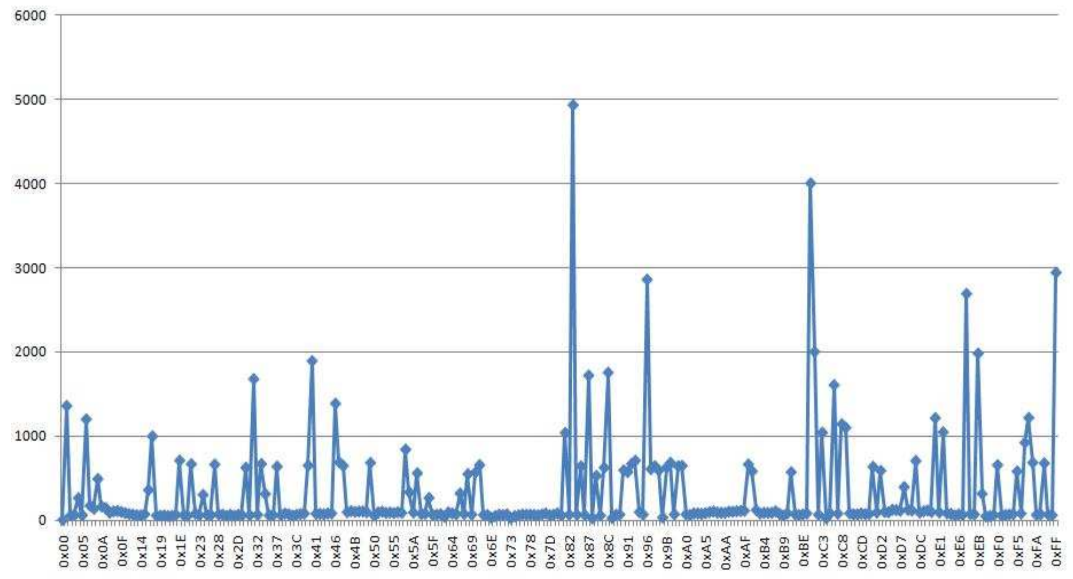


Figure 6.13: ADMmutate Byte Spectrum for Set 5000_1

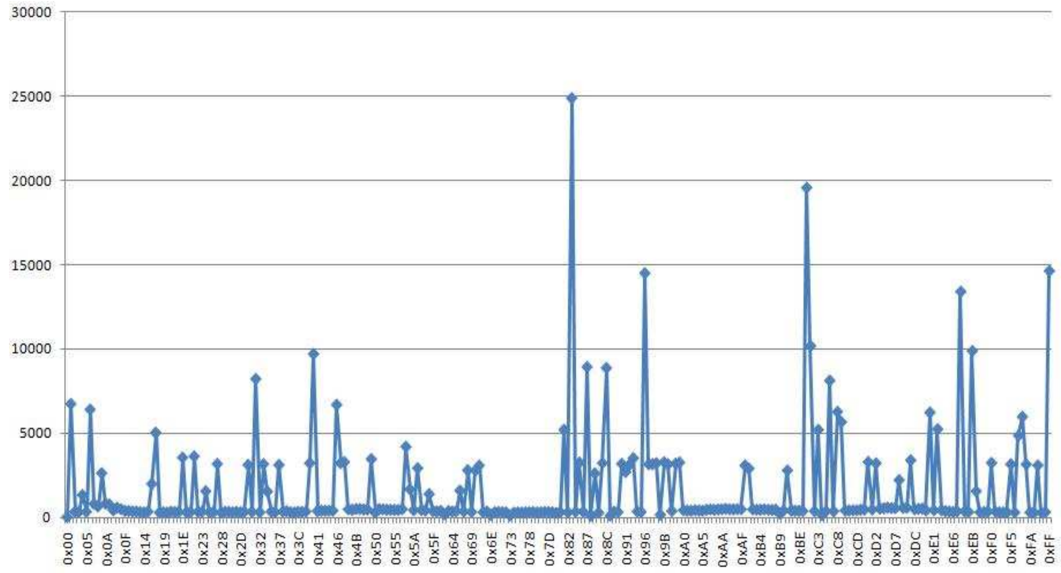


Figure 6.14: ADMmutate Byte Spectrum for Set 5000_2

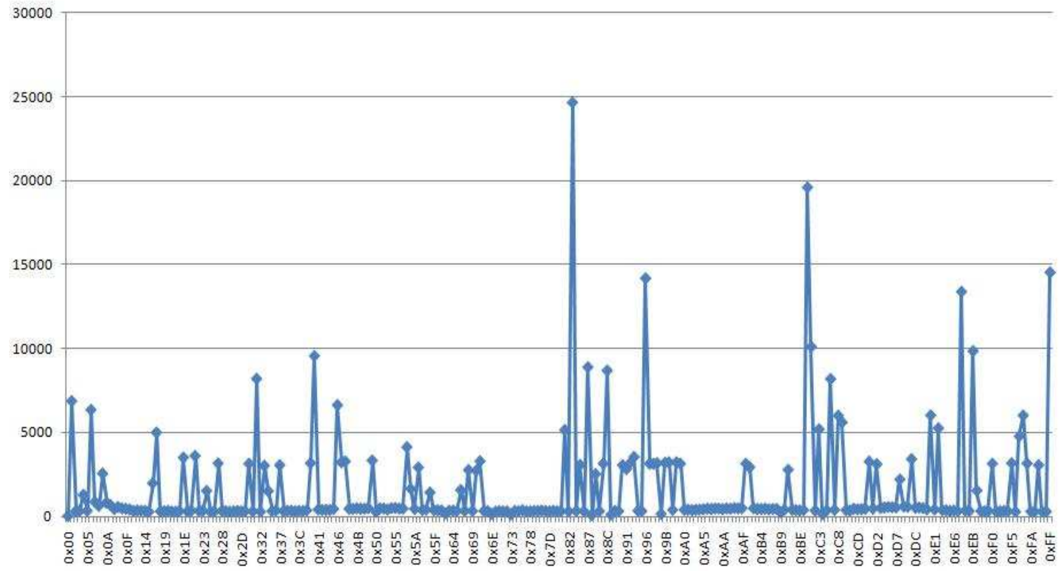


Figure 6.15: ADMmutate Byte Spectrum for Set 5000_3

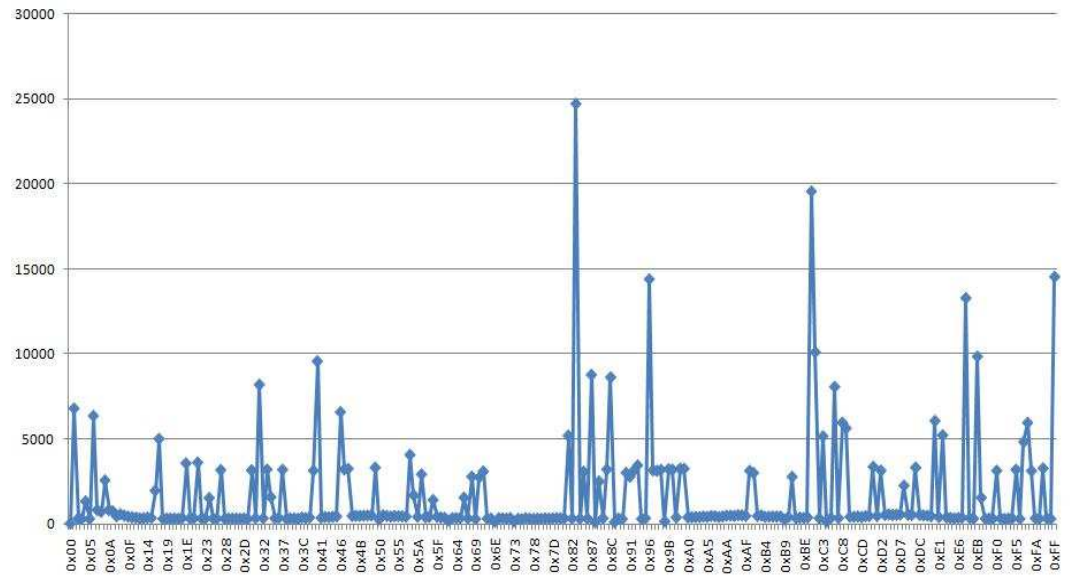


Figure 6.16: ADMmutate Byte Spectrum for Set 10000_1

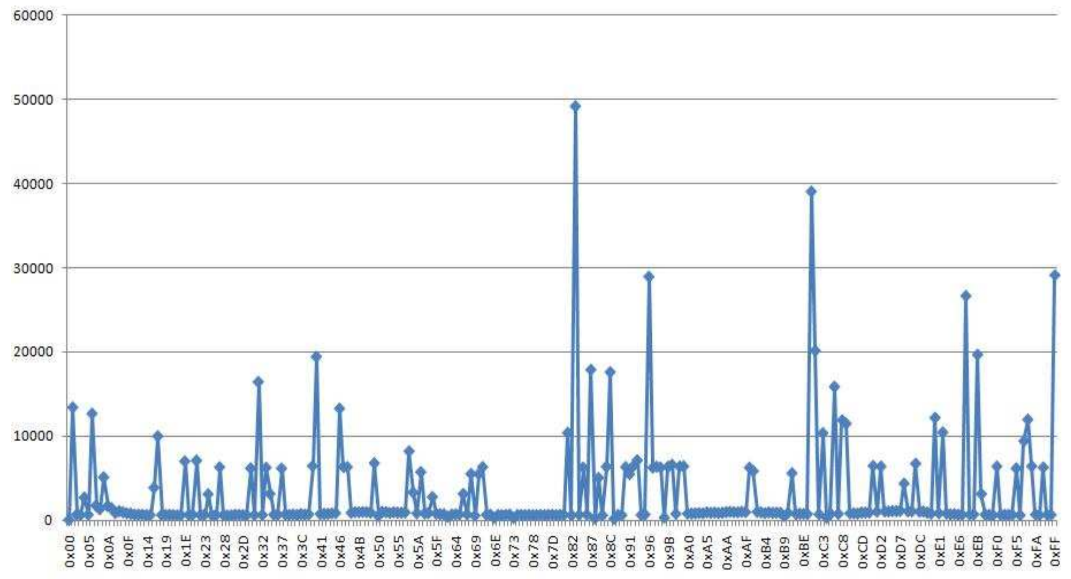


Figure 6.17: ADMmutate Byte Spectrum for Set 10000_2

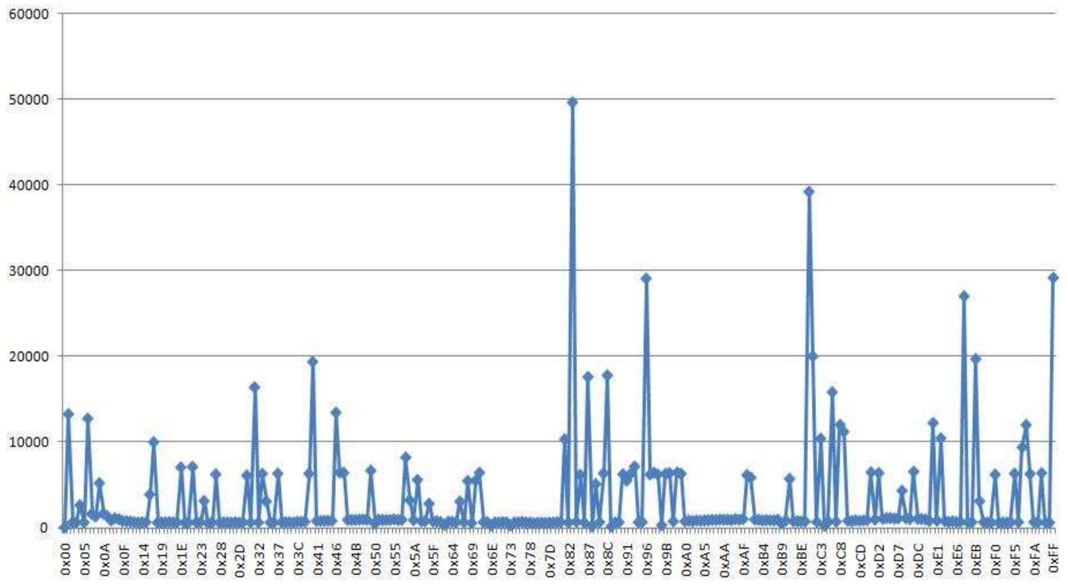
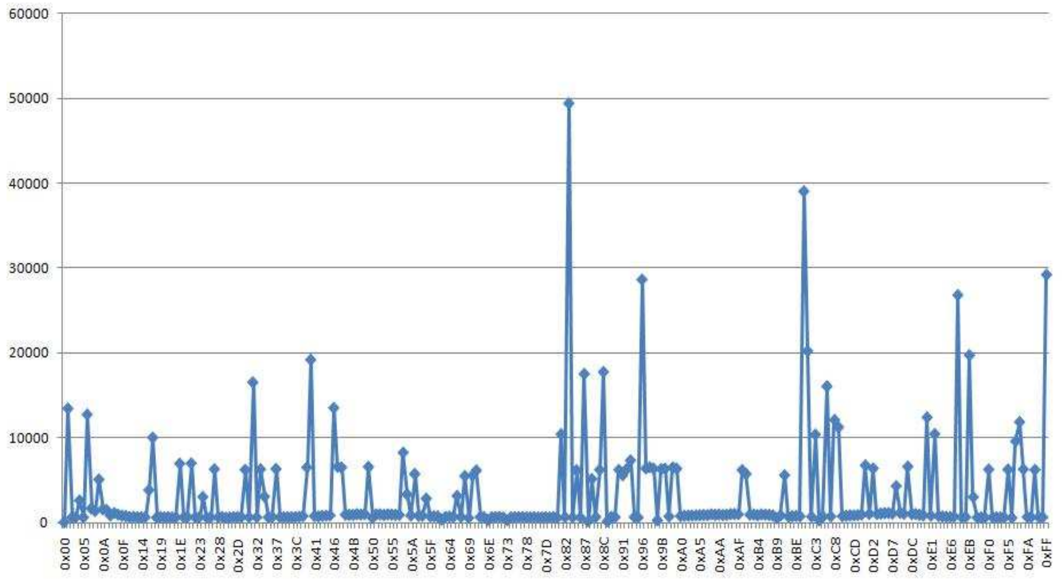


Figure 6.18: ADMmutate Byte Spectrum for Set 10000_3



Again, the same observations for Shikata Ga Nai are also verified for ADMmutate. We have an identical shape for all sets' byte spectrums. The most repeated bytes are constant across the various sets. As for the spectrums for Tapion, the byte occurence of each set are illustrated from Figure 6.19 to Figure 6.27 .

Figure 6.19: Tapion Byte Spectrum for Set 1000_1

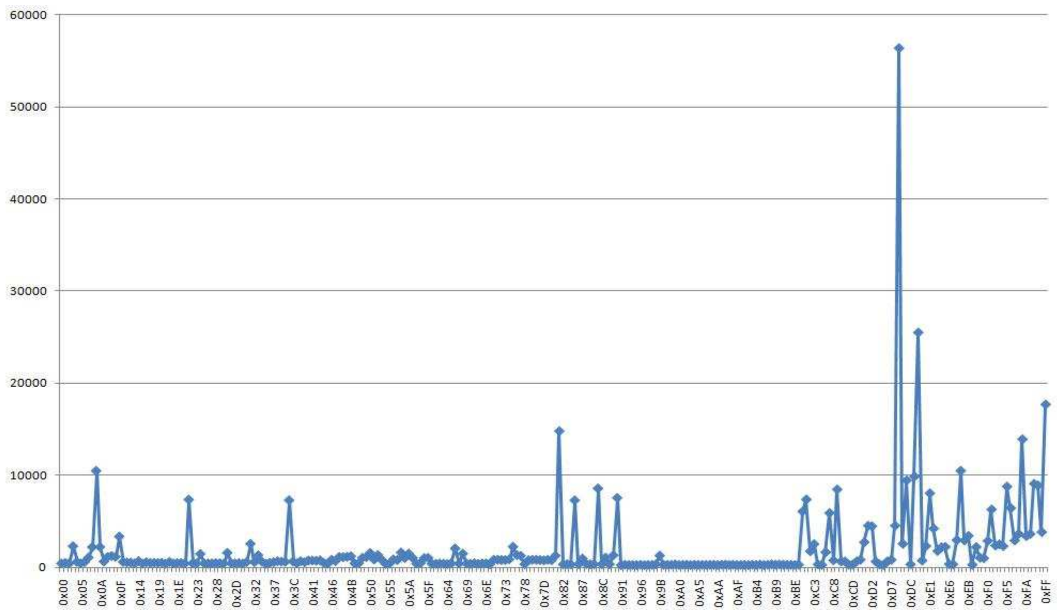


Figure 6.20: Tapion Byte Spectrum for Set 1000_2

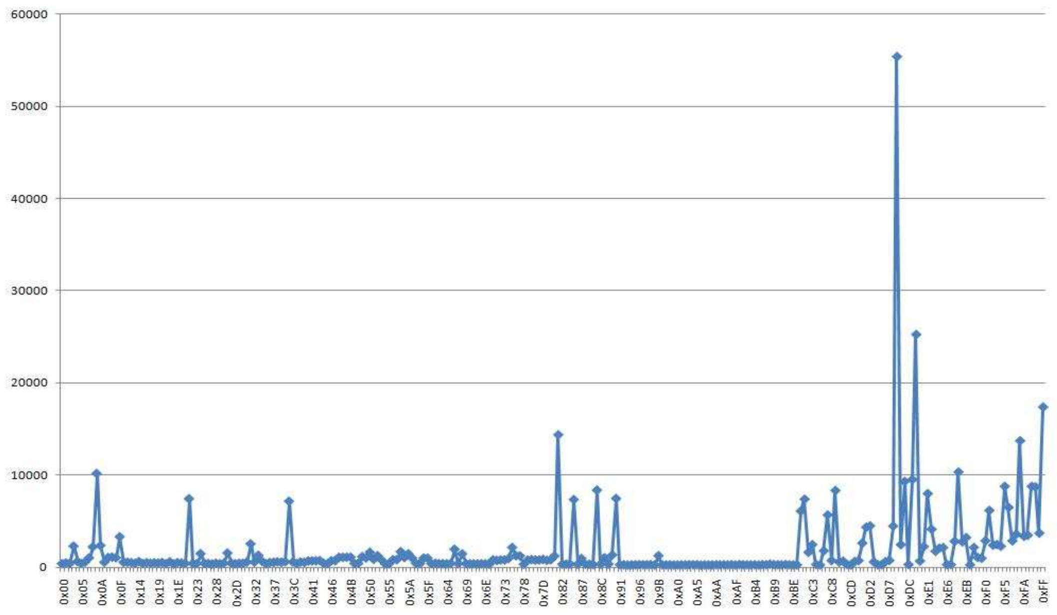


Figure 6.21: Tapion Byte Spectrum for Set 1000_3

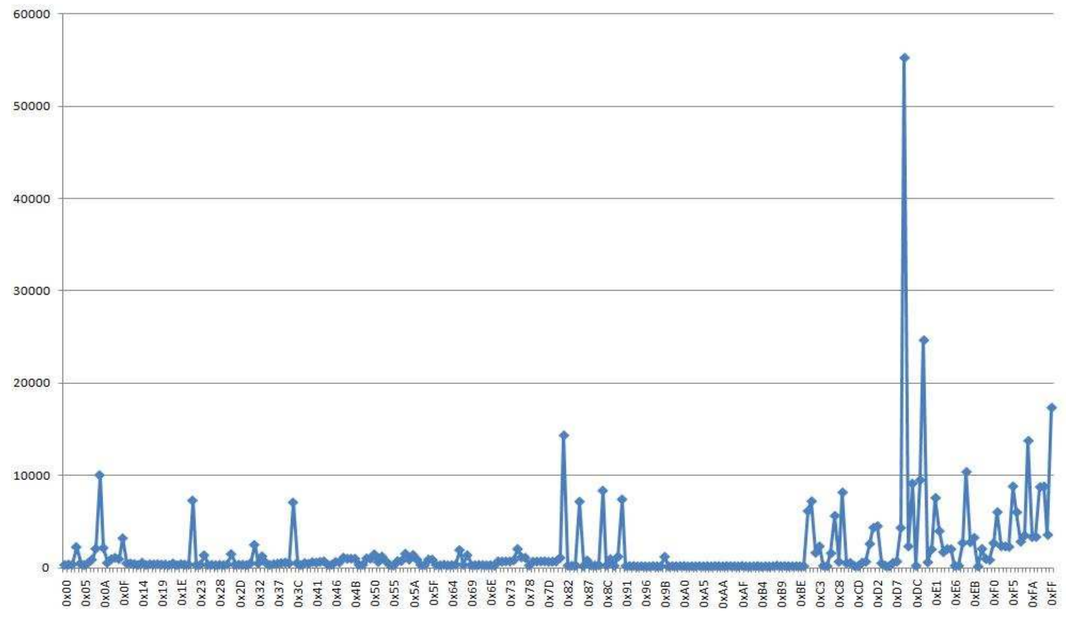


Figure 6.22: Tapion Byte Spectrum for Set 5000_1

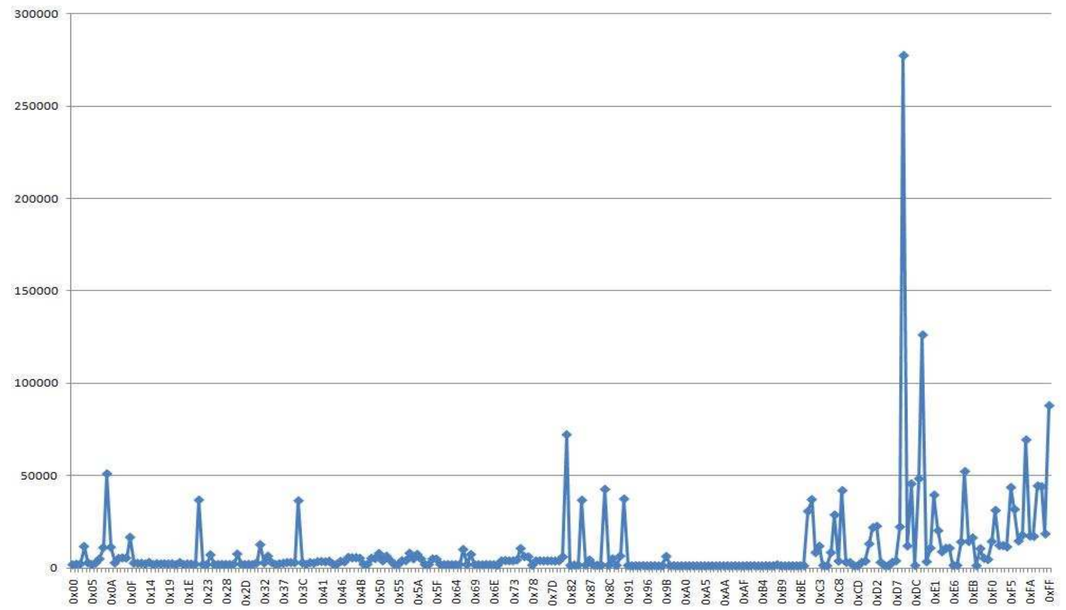


Figure 6.23: Tapion Byte Spectrum for Set 5000_2

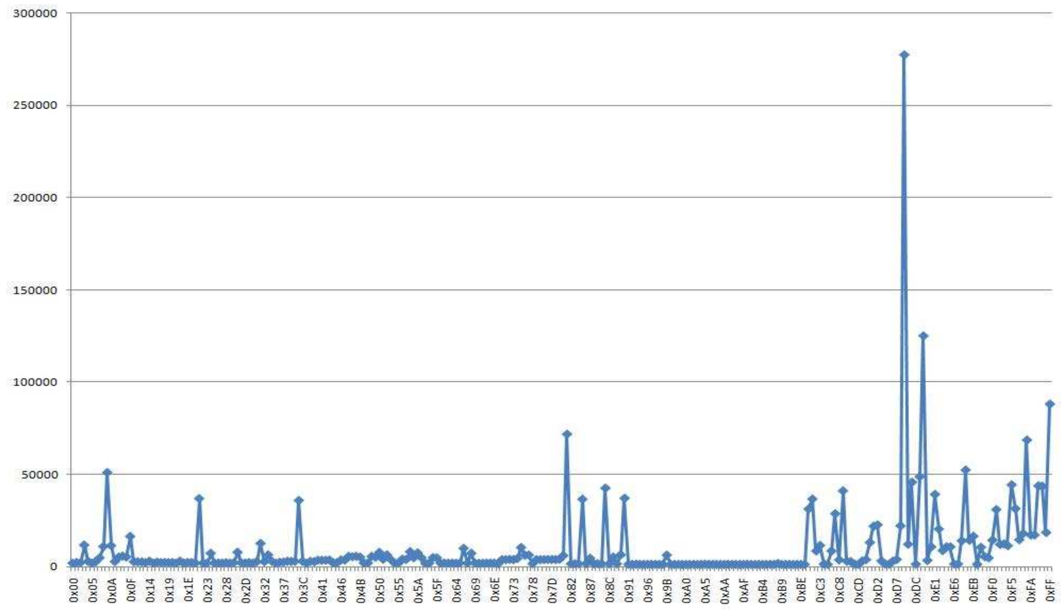


Figure 6.24: Tapion Byte Spectrum for Set 5000_3

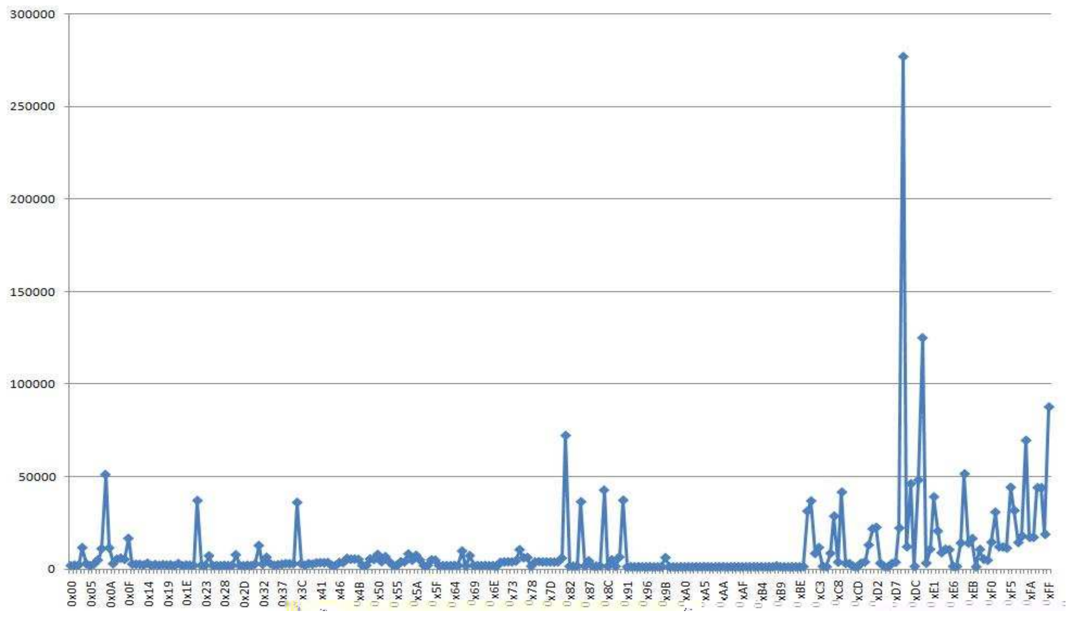


Figure 6.25: Tapion Byte Spectrum for Set 10000_1

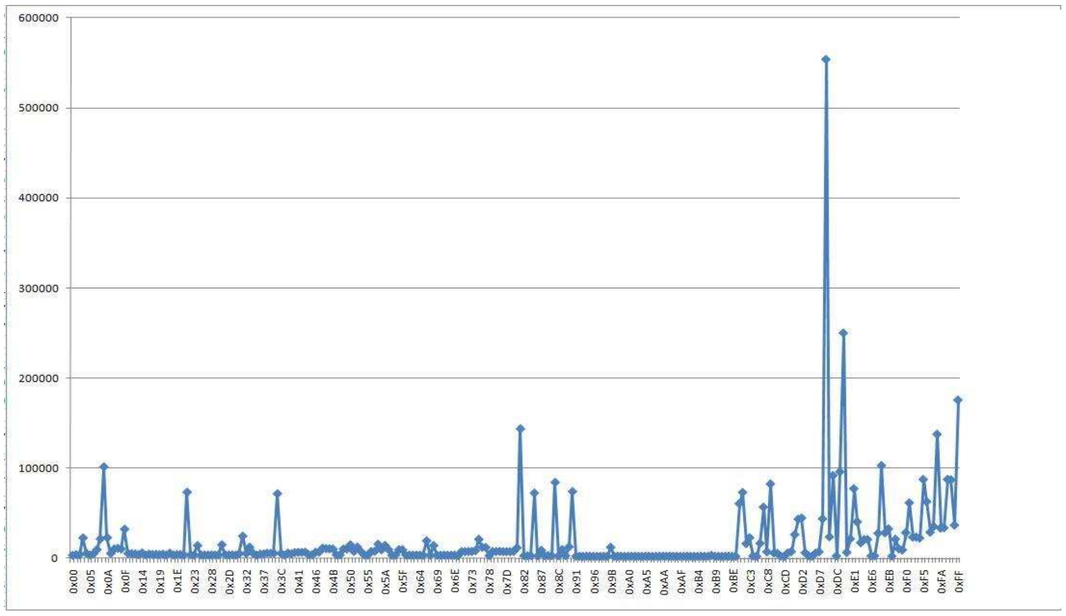


Figure 6.26: Tapion Byte Spectrum for Set 10000_2

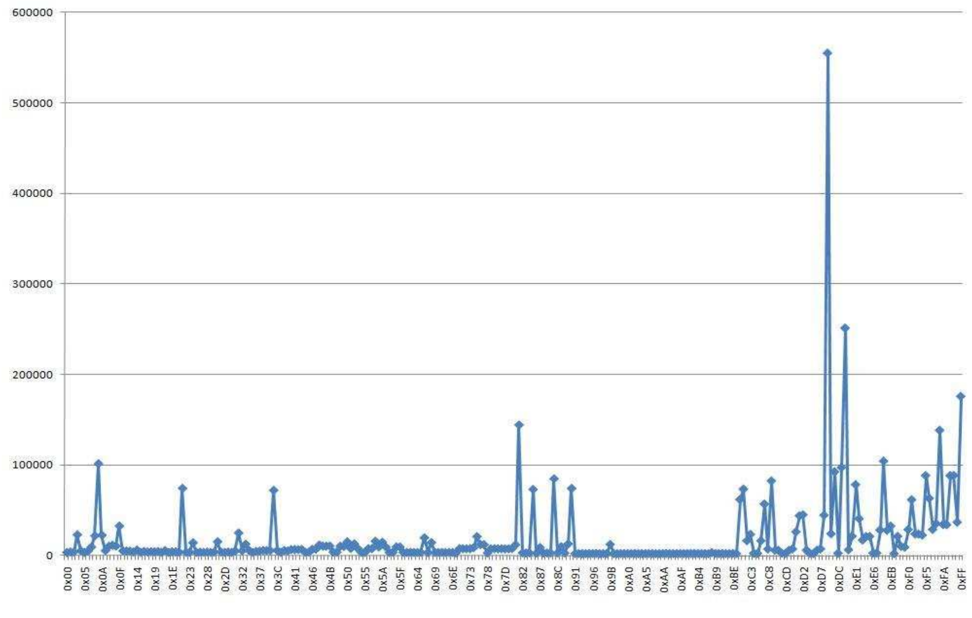
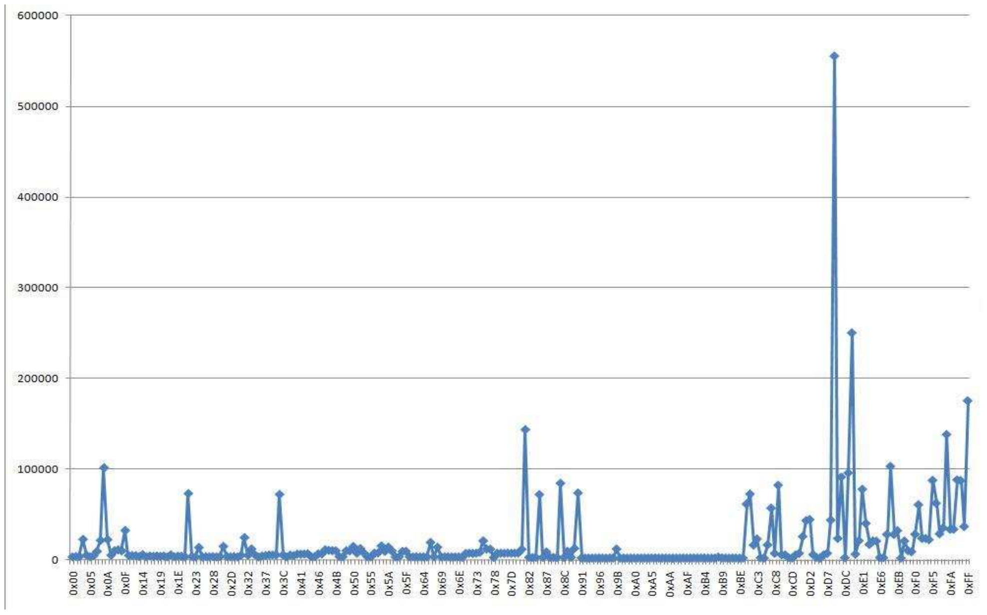


Figure 6.27: Tapion Byte Spectrum for Set 10000_3



We can observe that Tapion does not deviate from the previous observations. Given these results, we can conclude that polymorphic engines, indeed, explore the whole byte spectrum. However, they exhibit a similar pattern in terms of byte occurrences. Moreover, the same most repeated bytes are present across sets of different population size. Thus, we can use this valuable information as an important parameter in the analysis of byte composition.

6.3 Interpretation

While observing the number of occurrences of bytes for each polymorphic engine, one can notice that the most repeated bytes are constant across the various sets. Some of them are found in their respective shape signatures, however, others are not. The most important issue here is to investigate what this valuable information can be useful for and how it can help in understanding or modeling polymorphic engines. So far, we have studied the shape signatures and their bit position analysis. Now, we need another way to study polymorphic engines in order to integrate byte occurrences. The most important observation that one can make about those repeated bytes for each engine is that most of them are not *NOP* instructions (being single argument or multi argument). Based on that, it seems that an analysis of those engines in terms of composition could reveal interesting information about polymorphic engines.

6.4 Analysis

Let us consider a decoder as a zone of high polymorphism, we can define three main elements characterizing those decoders:

- Nop Area
- Byte Spectrum
- Other Bytes

By *Nop Area*, we mean the nop elements being single or multi byte arguments. A single byte argument NOP consists of a single x86 no-operation instruction whereas a multi byte argument NOP is a no-operation instruction that can take arguments. In this case, the *Nop Area* includes the no-operation instruction plus its argument. The arguments do not necessarily belong to the category of x86 no-operation instructions . As an example, the following sequence 0×3541424344 is considered a *Nop Area* as well as the single instruction 0×41. The *Byte Spectrum* consists of the bytes that characterize the polymorphic engines. The process of extracting those bytes is detailled in section 6.5. Finally, by *Other Bytes* we mean the bytes that are neither part of *Nop Area* nor the *Byte Spectrum*. Based on these elements, some logical assumptions that characterize a decoder in terms of composition have to be defined.

The first assumption to be made is that a decoder should have in excess (in terms of byte occurences) either the *Nop Area* or the *Byte Spectrum* and never the *Other Bytes*. Another important assumption is that the *Byte Spectrum* must not be neither very low nor null. The threshold for what we consider low is addressed in Sections 6.8 and 6.9 .The *Nop Area*, in terms of composition, should be recognizable, from a regular *NOP* section that might preceed the decoder. In other words, the decoder must not be composed of *NOPs* exclusively. The aim of this composition analysis is to check whether we can differentiate the decoders, in their compositions, from a regular *NOP* section and from any random sequence of bytes. In summary, our assumptions are as follows:

1. *Nops Area* or *Byte Spectrum* should be the most present elements in terms of composition.

2. *Byte Spectrum* must not be null.

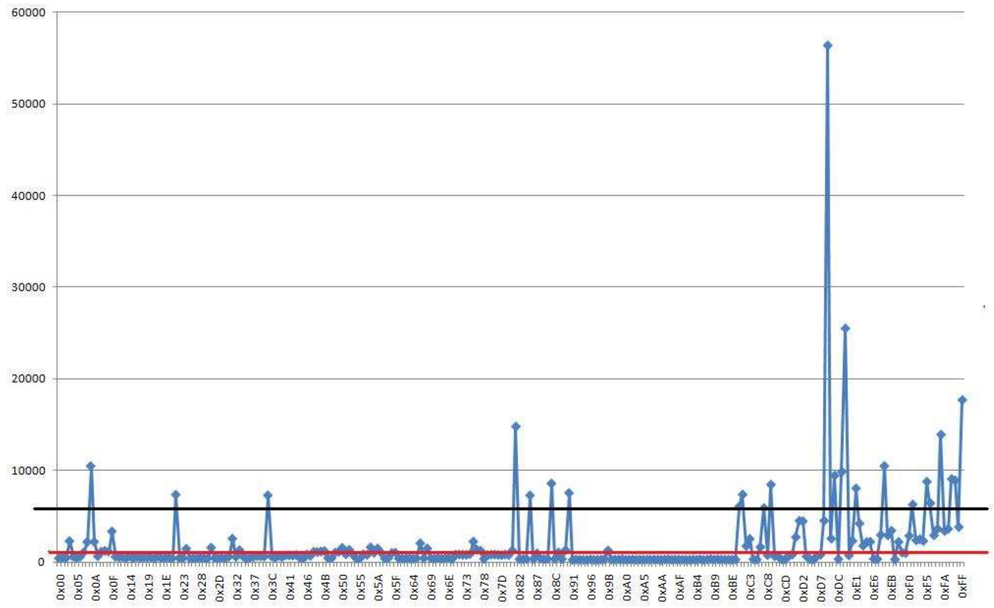
3. *Byte Spectrum* should have an acceptable amount (the threshold issue is addressed in sections 6.8 and 6.9).

4. *Nop Area* should be recognizable from a regular NOP section.

6.5 Byte Spectrum Extraction

The byte spectrum elements correspond to the most repeated bytes for a polymorphic decoder. In terms of occurences, their number is greater or equal than the population size. Byte spectrum figures have shown that those bytes remain the same across sets of population for each polymorphic engines. Consequently, analyzing one spectrum for a specific set is enough. This statement is reinforced by the fact that the shapes of byte occurrences are the same for all engines. For Shikate Ga Nai and ADMmutate, we see that the number of bytes exceeding the population size is relatively small compared to other bytes. Moreover, among these same most repeated bytes, the differences in terms of occurrences between bytes are relatively small. However, the case of Tapion is interesting since we can make for it the opposite claim; the number of most repeated bytes is large and the gaps between the number of bytes execeeding the population size and the other bytes are wide. This is mainly due to the ability of Tapion to generate a lot of padding. However, some bytes clearly distinguish themselves from others by having a number of occurences that is relatively important. By analyzing Tapion spectrum, presented in Figure 6.28, we can distinguish many levels that exceed the population number.

Figure 6.28: Tapion Byte Spectrum



The red line in Figure 6.28 shows the population size and the black line the distinguishable bytes in terms of occurrences.The aim is to extract those meaningful most repeated bytes. This problem is not encountered for Shikata Ga Nai and ADMmutate, which spectrums are illustrated in Figure 6.29 and Figure 6.30 respectively.

Figure 6.29: Shikata Ga Nai Byte Spectrum

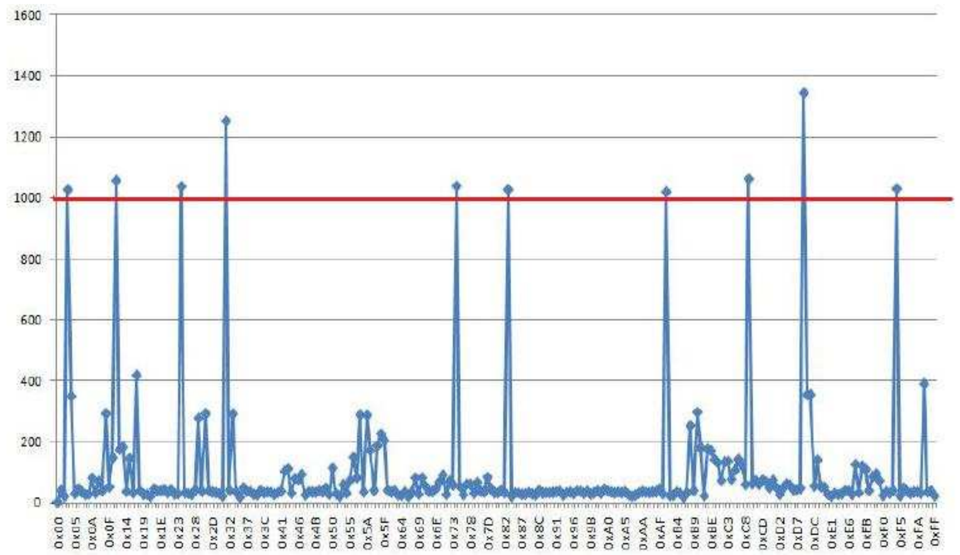
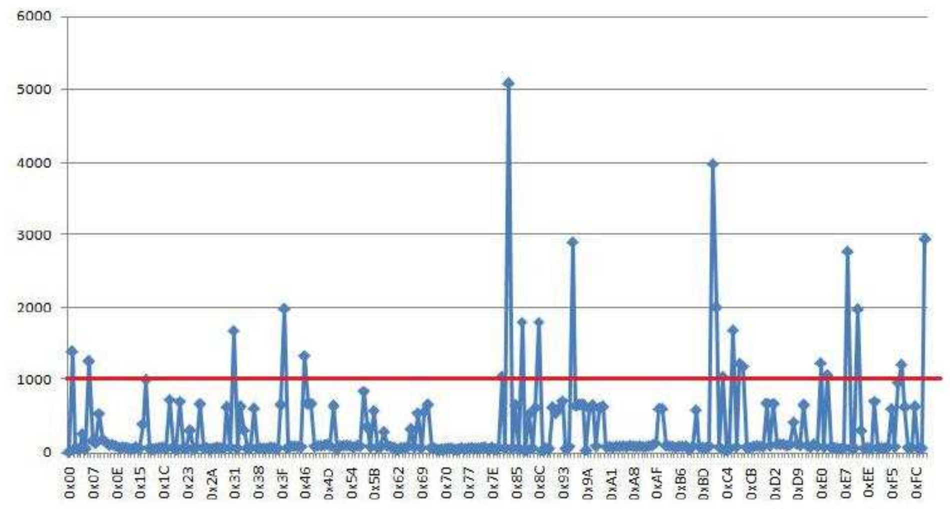


Figure 6.30: ADMmutate Byte Spectrum



Therein, the number of bytes exceeding the population size (above the red line) is small and the differences in terms of byte occurrences between those most repeated bytes are very small. The following algorithm deals with that by extracting the most meaningful bytes:

1. Select the bytes exceeding the population size (the population size corresponds to the number of decoders).

2. Select, among those bytes, the minimum occurrence (denoted by Min) and the maximum occurrence (denoted by Max).

3. Compute $\frac{Min}{Max}$:

   - If $\frac{Min}{Max} < 0.2$ then select only bytes that have their occurrence being at least 10% of Max.

   - Otherwise take all the bytes exceeding the population size.

The threshold of 0.2 allows us to recognize engines that are able to generate a lot of padding. If an engine is having the ratio $\frac{Min}{Max} < 0.2$, then we know that the engine is generating a lot of padding. Thus we need to apply a filter that permits the extraction of the most meaningful bytes. This is done by selecting bytes with an occurence of at least 10% of Max. From the tests conducted, the values chosen seem to be adequate for selecting the most meaningful bytes. By doing so, we are able to extract the bytes for each polymorphic engines' decoders shown in Tables 6.1, 6.2 and 6.3 for respectively Shikata Ga Nai, ADMmutate and Tapion.

Table 6.1: Shikata Ga Nai Meaningful Most Repeated Bytes

Shikata Ga Nai
03
11
24
31
74
83
$b1$
$c9$
$d9$
$f4$

Table 6.2: ADMmutate Meaningful Most Repeated Bytes

ADMmutate	
01	$c1$
06	$c6$
17	$c8$
31	$c9$
40	$e0$
46	$e2$
81	$e8$
83	eb
87	$f8$
$8c$	ff
96	
$c0$	

Table 6.3: Tapion Meaningful Most Repeated Bytes

Tapion	
09	dd
21	de
$3b$	$e1$
81	$e9$
$8b$	$f1$
$c0$	$f5$
$c1$	$f6$
$c7$	fc
$c9$	fd
$d9$	$f9$
db	ff

6.6 Byte Spectrum Analysis Approach

In order to analyze the decoders in terms of composition, we inspect the decoder's structure in terms of bytes. The main elements used (*Nops Area*, *Byte Spectrum* and

Other Bytes) are the same as defined in previous section 6.4. For each specific polymorphic engine, the algorithm performs the following steps for each sets of population:

1. Separate the *Byte Spectrum* from possible *NOP* elements.

2. Identify the *Nops Area* of the decoder by counting the number of bytes being *NOP* elements.

3. Identify the *Byte Spectrum* by counting the corresponding bytes.

4. Count the *Other Bytes*.

5. Compute the composition rates:

   - $Nops\ Area = \dfrac{\sum Nop\ elements}{\sum Bytes}$.
   - $Byte\ Spectrum = \dfrac{\sum Byte\ Spectrum\ elements}{\sum Bytes}$
   - $Other\ Bytes = \dfrac{\sum Other\ Bytes\ elements}{\sum Bytes}$

6. Record the compositions.

7. After analyzing all the population members, compute average compositions and standard deviation for each composition.

The composition rates will help us to better understand the bytes' structure of a polymorphic decoder. After analyzing all the sets of population for a specific polymorphic engine, we average the results to get a final composition. If the final compositions verify our previous assumptions, then we can effectively develop a model for recognizing decoders of polymorphic engines in terms of byte composition.

6.7 Results

The byte spectrum analysis gives us the following results. For *Shikata Ga Nai*, Table 6.4 illustrates the composition results for each of the previously described elements:

Table 6.4: Composition Results for Shikata Ga Nai

Sets	Metrics	Nops Area	Byte Spectrum	Other Bytes
1000_1	composition	0.59	0.183	0.22
	standard_dev	0.065	0.04	0.056
1000_2	composition	0.586	0.185	0.227
	standard_dev	0.068	0.04	0.06
1000_3	composition	0.588	0.184	0.227
	standard_dev	0.066	0.04	0.06
5000_1	composition	0.59	0.184	0.225
	standard_dev	0.065	0.04	0.057
5000_2	composition	0.59	0.184	0.2247
	standard_dev	0.06	0.04	0.057
5000_3	composition	0.59	0.184	0.224
	standard_dev	0.065	0.04	0.056
10000_1	composition	0.59	0.184	0.224
	standard_dev	0.065	0.04	0.056
10000_2	composition	0.5916	0.184	0.224
	standard_dev	0.066	0.04	0.057
10000_3	composition	0.59	0.184	0.224
	standard_dev	0.066	0.04	0.057

For *ADMmutate*, Table 6.5 illustrates the composition results for each of the previously described elements:

Table 6.5: Composition Results for ADMmutate

Sets	Metrics	Nops Area	Byte Spectrum	Other Bytes
1000_1	composition	0.545	0.313	0.1407
	standard_dev	0.06	0.051	0.044
1000_2	composition	0.553	0.308	0.14
	standard_dev	0.06	0.053	0.045
1000_3	composition	0.5505	0.309	0.14
	standard_dev	0.061	0.052	0.043
5000_1	composition	0.548	0.309	0.142
	standard_dev	0.061	0.053	0.045
5000_2	composition	0.548	0.31	0.141
	standard_dev	0.061	0.05	0.045
5000_3	composition	0.549	0.308	0.1415
	standard_dev	0.061	0.0532	0.045
10000_1	composition	0.555	0.309	0.141
	standard_dev	0.061	0.0529	0.045
10000_2	composition	0.55	0.303	0.1408
	standard_dev	0.061	0.052	0.045
10000_3	composition	0.55	0.309	0.1402
	standard_dev	0.061	0.052	0.047

For *Tapion*, Table 6.6 illustrates the composition results for each of the previously described elements:

Table 6.6: Composition Results for Tapion

Sets	Metrics	Nops Area	Byte Spectrum	Other Bytes
1000_1	composition	0.372	0.4086	0.22
	standard_dev	0.029	0.0258	0.023
1000_2	composition	0.37	0.4098	0.22
	standard_dev	0.029	0.025	0.0226
1000_3	composition	0.369	0.4095	0.22
	standard_dev	0.029	0.0276	0.0234
5000_1	composition	0.37	0.409	0.22
	standard_dev	0.029	0.026	0.023
5000_2	composition	0.369	0.41	0.22
	standard_dev	0.028	0.0257	0.0232
5000_3	composition	0.3711	0.409	0.219
	standard_dev	0.029	0.0262	0.0232
10000_1	composition	0.3703	0.409	0.22
	standard_dev	0.029	0.026	0.023
10000_2	composition	0.3709	0.409	0.219
	standard_dev	0.029	0.026	0.02276
10000_3	composition	0.3704	0.409	0.219
	standard_dev	0.029	0.0259	0.023

The average composition results for each engine are summarized in the following Table 6.7:

Table 6.7: Average Composition Results for the Engines

Engine	Metrics	Nops Area	Byte Spectrum	Other Bytes
Shikata Ga Nai	composition	0.589	0.184	0.224
	standard_dev	0.065	0.026	0.057
ADMmutate	composition	0.549	0.309	0.1407
	standard_dev	0.061	0.052	0.0448
Tapion	composition	0.37	0.409	0.219
	standard_dev	0.028	0.026	0.023

6.8 Interpretation

The first observation that we can make is that the standard-deviations are very low for all engines across all sets of population generated. Thus, somehow we can assess the reliability of the results. Our first assumption, which states that either the *Nops Area* or *Byte Spectrum* should be the elements in excess (in terms of composition) is verified. We can see that even when the *Nops Area* elements are in excess, we can differentiate it from a regular *NOP* section. A *NOP* section is exclusively composed of *NOPS* and thus have a composition of 100%. We can also observe that the amount of *Byte Spectrum* is never null and that its amount is never below 15%. In other words, this amount is never low (by low we mean being less than 5%). However, we still need to determine what we consider as an acceptable amount of *Byte Spectrum*. Accordingly, the minimum threshold for *Byte Spectrum* has to be defined in order to be able to differentiate it from any random sequence of bytes. The best way to do this is to repeat the experiment, for each polymorphic engines' *Byte Spectrum* elements, against regular network traffic, which provides us with enough randomness for byte sequences. We analyze the composition of the network dumps against the three *Byte Spectrum* retrieved from each polymorphic engine. It is important to mention the fact that we analyze the composition of the data part of network packets, thus stripping off the headers.

6.9 Network Traffic Results

The set up for network traffic analysis is the following: we record the traffic and generate three sets of data. During the traffic generation, we diversify our activity. We mean by that consulting various web sites, cheking emails, connecting to SSH, FTP accounts, etc. The aim is to generate network traffic with enough randomness. The traffic is recorded using a regular sniffer and saved into a file. Then, we strip off the network headers keeping only the data part. The composition results, for each network traffic, against the *Byte Spectrum* elements retrieved from each polymorphic engine are summed up in the following tables. Table 6.8, Table 6.9 and Table 6.10 illustrate the composition results of network traffics against *Byte Spectrum* elements of respectively Shikata Ga Nai, ADMmutate and Tapion.

Table 6.8: Composition for Network Dumps Against Shikata Ga Nai Byte Spectrum

Network dumps	Metrics	Nops Area	Byte Spectrum	Other Bytes
network_dump_1	composition	0.474	0.026	0.499
	standard_dev	0.092	0.014	0.09
network_dump_2	composition	0.5	0.02	0.473
	standard_dev	0.1	0.013	0.1
network_dump_3	composition	0.57	0.016	0.41
	standard_dev	0.091	0.014	0.093

Table 6.9: Composition for Network Dumps Against ADMmutate Byte Spectrum

Network dumps	Metrics	Nops Area	Byte Spectrum	Other Bytes
network_dump_1	composition	0.47	0.0835	0.44
	standard_dev	0.091	0.036	0.073
network_dump_2	composition	0.5	0.083	0.415
	standard_dev	0.1	0.045	0.07
network_dump_3	composition	0.57	0.038	0.387
	standard_dev	0.09	0.023	0.088

Table 6.10: Composition for Network Dumps Against Tapion Byte Spectrum

Network dumps	Metrics	Nops Area	Byte Spectrum	Other Bytes
network_dump_1	composition	0.474	0.0612	0.46
	standard_dev	0.09	0.029	0.079
network_dump_2	composition	0.5	0.05	0.447
	standard_dev	0.1	0.034	0.083
network_dump_3	composition	0.57	0.037	0.38
	standard_dev	0.09	0.022	0.09

The average composition results, for each network traffic, against the *Byte Spectrum* elements retrieved from each polymorphic engine are summarized in the following Table 6.11:

Table 6.11: Average Composition Results Against Engines' Spectrums

Engine	Metrics	Nops Area	Byte Spectrum	Other Bytes
Shikata Ga Nai	composition	0.51	0.062	0.462
	standard_dev	0.094	0.013	0.094
ADMmutate	composition	0.51	0.068	0.414
	standard_dev	0.093	0.034	0.077
Tapion	composition	0.51	0.0494	0.429
	standard_dev	0.093	0.085	0.084

6.10 Interpretation

We observe that the standard-deviations are very low (being less than 0.1), thus we can assess the reliability of the results. An important observation that we can make is that the *Byte Spectrum* elements are very low, which is a good sign since it can help us differentiate polymorphic decoders' compositions from any random sequence of bytes. We can also observe that the *Byte Spectrum* elements that we have extracted are really characterizing each of their respective polymorphic engine. Now we need to define the minimum threshold for the *Byte Spectrum* element. Let R be $\frac{\textit{Byte Spectrum}}{\textit{Nops Area}}$ ratio. The latter is computed for all the engine *Byte Spectrum* against the network traffics. The same is done for the average composition for each polymorphic engine. The corresponding results are summarized in Table 6.12 and Table 6.13 .

Table 6.12: Ratios for Network Dumps Against Polymorphic Engines Byte Spectrums

Engine	Ratio
Shikata Ga Nai	0.12
ADMmutate	0.133
Tapion	0.097

Table 6.13: Ratios for Polymorphic Engines Decoders' Composition

Engine	Ratio
Shikata Ga Nai	0.31
ADMmutate	0.56
Tapion	1.1

Regardless of the polymorphic engine analyzed, we always have a reasonable amount of *Byte Spectrum* elements present in the composition. This can be observed in the ratios. Thus, we can state that a polymorphic decoder should have its ration R such that: $R > 0.15$ which is equivalent to $\frac{\textit{Byte Spectrum}}{\textit{Nops Area}} > 0.15$.

6.11 Rules of Composition

Once the minimum threshold for the *Byte Spectrum* elements is computed, we can derive characteristics, in terms of byte composition, that allow us to differentiate polymorphic decoders from any other sequence of bytes:

1. *Nops Area* or *Byte Spectrum* should be the most present elements (in terms of composition)

2. *Other Bytes* could never be the most frequent elements

3. *Nops Area* < 0.9

4. *Byte Spectrum* $\neq 0$

5. $\frac{\textit{Byte Spectrum}}{\textit{Nops Area}} > 0.15$

In the following section, we verify the rules by testing them against all the sets of population of the polymorphic engines.

6.12 Verification

In this section, we check for each polymorphic engine whether the rules hold. This is done for the purpose of validating our previous conclusions. Table 6.14 to Table 6.16 show the correlation rate for respectively *Shikata Ga Nai*, *ADMmutate* and *Tapion* .

Table 6.14: Shikata Ga Nai Correlation for Composition

Sets	Rate of correlation
1000_1	97.2%
1000_2	97.8%
1000_3	97%
5000_1	97.82%
5000_2	97.6%
5000_3	97.66%
10000_1	97.51%
10000_2	97.78%
10000_3	97.3%

Table 6.15: ADMmutate Correlation for Composition

Sets	Rate of correlation
1000_1	99.89%
1000_2	99.89%
1000_3	100%
5000_1	99.7%
5000_2	99.79%
5000_3	99.75%
10000_1	99.79%
10000_2	99.71%
10000_3	99.79%

Table 6.16: Tapion Correlation for Composition

Sets	Rate of correlation
1000_1	100%
1000_2	100%
1000_3	100%
5000_1	100%
5000_2	100%
5000_3	100%
10000_1	100%
10000_2	100%
10000_3	100%

The average composition correlation rates are presented in Table 6.17 .

Table 6.17: Average Correlation for Composition

Engine	Rate of correlation
Shikata Ga Nai	97.52%
ADMmutate	99.81%
Tapion	100%

One can notice that there are high percentages of correlation. This confirms that the composition rules hold and that they can be used to characterize a decoder's polymorphic engine. However, in order to assess the validity of the results, we need to test these rules against the network dumps for the three *Byte Spectrum*. If we get low percentages of correlation, then we can confirm the fact that we can differentiate polymorphic engines' decoders from any random sequence of bytes. The obtained results are shown in Table 6.18 .

Table 6.18: Composition Correlation for Network Dumps against Byte Spectrums

Network dumps	Shikata Ga Nai	ADM	Tapion
network_dump_1	0.003%	19%	4.1%
network_dump_2	0.002%	22%	1.8%
network_dump_3	0.007%	2.6%	2%

We can observe that the rates of correlation are very low. In other words, it is possible to distinguish a polymorphic engine, in terms of byte composition since the network dumps do not highly correlate with our rules of composition. This confirms the validity of the rules for composition.

6.13 Conclusion

Our experiments have shown that indeed polymorphic engine decoders could be effectively characterized in terms of byte composition. In addition, we showed that those decoders can be mapped by effectively predicting the areas that are subject to high mutations as well as areas that correspond to the shape signatures. Thus, we are in a position to states that our attempt of modeling polymorphic engines is very promising.

Chapter 7

Conclusion

The main issue, when dealing with polymorphic shellcode, is to come up with a model that is able to express a polymorphic behavior. We know, from [10,11,12,13] that those engines leave artefacts even in their mutation processes due, mainly, to consistency constraints. We also know that they are able to exhibit high degree of polymorphism. Having such information in mind, we were able to develop a new concept of signature called *Shape Signatures*. Our model detect the invariant parts, due to consistency constraints, as well as mutated part due to polymorphism. In other words, instead of trying to extract traditional signatures for which we know the number of signatures will be huge, we look for a "mould" or higher model that can express all those possible instances that could be generated. This combination expresses the characteristics or abilities of polymorphic engines in a much better way compared to traditional signature-based models who fail against them. We implemented a genetic algorithm that takes in its fitness function the new approach to extract those shape signatures. The testing was done against the most powerful polymorphic engines. At the end, we are able to extract very reliable signatures.

In an attempt of modeling polymorphic engines, we studied the bit positions of the newly extracted signatures. The results were that we are able to effectively map the decoder by predicting and locating precisely the areas subject to high mutation and areas that correspond to shape signatures. The conclusions are that polymorphic engines' shape signatures do not explore the whole decoder's dimensions (in terms of bit positions) but rather shift in a defined area of the decoder. We also looked at the composition of the studied polymorphic engines in terms of bytes and showed that each one of them exhibit a specific byte composition that uniquely characterizes it. In the experiments, we were able to extract constant patterns, in terms of composition, against very large sets of decoders. Our modeling demonstrate that polymorphic decoders do not look as random as they are.

We believe that the methods and mechanisms that were developed could truly help modeling and better understanding how polymorphic decoders behave. We still believe that the modeling could be enhanced by trying to look for shape signatures that represent the whole decoder. By this, we mean, instead of extracting shape signatures for small portions of the decoders, we could look for one shape signature that characterize the whole decoder. Further work could be done on the Byte Spectrum or most repeated bytes extracted. This is especially the case when it comes to analyzing their byte position (for the most repeated bytes). For instance, one can observe that the byte EB for ADMmutate decoders is always starting the decoder. For Shikata Ga Nai, we found that at position 17,20 and 23 of the decoder, bytes $31, 03, 83$ are always alternating at those specific positions. Consequently, a mechanism for analyzing the bit positions of the most repeated bytes could be developed. Furthermore improvement could be made by integrating all the concepts developed in this thesis into a framework of detection taking into consideration the shape signatures, the bits position analysis, and the composition analysis. In addition to that, the work could be enhanced drastically by taking into account the effect of each used instructions. So far, in the analysis, we have taken a high-level approach of polymorphic engines by only looking at their bytes without relating to their implications in terms of what job they do. We believe that by doing so, we can develop a framework, using advanced artificial intelligence techniques, to better enhance modeling. Our concept of shape signatures, analysis of the bits positions and the composition analysis are an initial contribution towards better improvement in modeling polymorphic decoders. According to the experiments we realized, it is really possible to model these engines since our approach is really showing some promising results. Since polymorphic engine mutation techniques are becoming very powerful, we think the main issue when it comes to detection is finding a way to better express their polymorphic nature. The concept of shape signature is an initial contribution towards developing more adequate ways but we truly believe that there is still of lot of room for improvement regarding how we can express polymorphism into a form of detection.

Bibliography

[1] A. Pasupulati, J. Coit, K. Levitt, F. Wu, S.H. Li, J.C. Fan, K.P. Buttercup. On network-based detection of polymorphic buffer overflow vulnerabilities. In *Network Operation and Management Symposium*. IEEE/IFIP, 2004.

[2] L-C Wuu, C-R Dow, K-H Chen, T-J Liu, H-L Huang. A Polymorphic Shellcode Detection Mechanism in the Network. In *Proceedings of the* 2^{nd} *international conference on Scalable information systems, ACM International Conference Proceeding Series*, Vol.304 . ACM, 2007.

[3] Y. Song, M. E. Locasto, A. Stavrou, A.D. Keromytis, S.J. Stolfo. On the infeasibility of Modeling Polymorphic shellcode for Signature Detection. In *Proceedings of the* 14^{th} *ACM conference on Computer and communications security, Conference on Computer and Communications Security*. ACM, 2007.

[4] M. Polychronakis, E.P Markatos, and K.G Anagnostakis. Network-Level Polymorphic Shellcode Detection Using Emulation. In *Proceedings of the Third Conference on Detection of Intrusions and Malware & Vulnerability Assesment*. DIMVA, July 2006.

[5] M. Polychronakis, K.G. Anagnostakis, E.P. Markatos. Real-World Polymorphic Attack Detection using Network-level Emulation. In *Proceedings of the* 4^{th} *annual workshop on Cyber security and information intelligence research:developing strategies to meet the cyber security and information intelligence challenges ahead*, Vol.288 . CSIIRW, 2008.

[6] U. Payer, P. Teufl, M. Lamberger. Hybrid Engine for polymorphic shellcode detection. In *Proceedings of the Conference Detection of Intrusions and Malware & Vulnerabiltity Assessment*, pages 19-31. DIMVA, July 2005.

[7] W. Lanjia, D. HaiXin & L.I. Xing. Dynamic emulation based modeling and detection of polymorphic shellcode at the network level. In *Science in China Series F: Information Sciences*. Science in China Press, co-published with Springer-Verlag GmbH, 2008.

[8] P. Bania, Evading network-level emulation. Available at http://www.packetstormsecurity.org/papers/bypass/pbania-evading-nemu2009.pdf (accessed on 2009/08/06).

[9] A. Koleniskov, W. Lee, P. Fogla, M. Sharif, R. Perdisci. Advanced Polymorphic Worms: EvadingIDS by Blending with Normal Traffic. Technical report, Georgia Tech College of Computing, 2004.

[10] H-Ah Kim, B. Karp. Autograph,Toward Automated,Distributed Worm Signature Detection. In *Proceedings of the* 13^{th} *conference on USENIX Security Symposium*, Vol.13 . USENIX, 2004.

[11] C. Kreibich and J. Crowcroft. Honeycomb-Creating Intrusion Detection Signatures Using Honeypots. In *ACM SIGCOMM Computer Communication Review*, 2004.

[12] J. Newsome, B. Karp and D. Song. Polygraph:Automatically Generating Signatures for Polymorphic Worms. In *Proceedings of the IEEE Symposium on Security and Privacy*. ACM, 2005.

[13] Z. Li, M. Sanghi, Y. Chen, M. Yang, K. and B. Chavez. Fast signature generation for zero-day polymorphic worms with provable attack-resilience. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 2006.

[14] ADMmutate polymorphic engine. Available at http://www.ktwo.ca/security.html (accessed on 2009/06/11).

[15] The Metasploit framework project. Available at http://www.metasploit.com (accessed on 2008/04/06).

[16] S.N. Sivanandam and S.N. Deepa. Introduction to Genetic Algorithms. Springer-Verlag Berlin Heidelberg, 2008.

[17] T. Toth, C. Kruegel. Accurate Buffer Overflow Detection via Abstract Payload Execution. LNCS 2516, pp.274-291. RAID ,2002.

[18] Polymorphic Shell codes vs. Application IDSs. NGSEC White Paper available at http://liwuw nesec.com (accessed on 2009/07/22).

[19] NIDSfindshellcode. Available at http://www.ngsec.com/downloads/misc/NIDSfindshellcode.tgz (accessed on 2009/07/22).

[20] Snort, the open-source Network Intrusion Detection System. Available at http://www.snort.org/ (accessed on 2008/03/15).

[21] Multi-argument no-operation instructions. Available at http://ecl-labs.org/papers/ecl-poly.txt (accessed on 2009/05/23).

[22] SEH exploitation. Available at http://freeworld.thc.org/download.php_?tp̄&fP̄ractical-SEHexploitation.pdf (accessed on 2010/01/05).

[23] Address Space Layout Randomization. Available at http://pax.grsecurity.net/docs/aslr.txt (accessed on 2008/02/15).

[24] Address Space Layout Randomization exploitation. Available at http://www-users.rwth-aachen.de/Tilo.Mueller/ASLRpaper.pdf (accessed on 2008/04/15).

[25] Chinchani R., Berg E. A fast static analysis approach to detect exploit code inside network flows. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection*, Springer-Verlag, pages 284308. RAID, 2005.

[26] Kruegel C., Kirda E, Mutz D, et al. Polymorphic worm detection using structural information of executables. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection*, Springer-Verlag. RAID, 2005.

[27] Tapion polymorphic engine. Available at http://pb.specialised.info/all/tapion/ (accessed on 2009/07/05).

[28] J. Holland. Adaptation in Natural and Arti cial Systems. MIT Press, Cambridge, Massachusetts, second edition, 1992.

[29] D. E. Goldberg, K. Deb, H. Kargupta, H. George. Rapid Accurate Optimization of Difficult Problems Using Fast Messy Genetic Algorithms. In *Proceedings of The Fifth International Conference On Genetic Algorithms. Morgan Kaufmann Publishers*, pages 56-64, 1993.

[30] Levenshtein, V. I. . Binary codes capable of correcting deletions, insertions and reversals. Soviet Physics-Doklady 10, pages 707-710, 1996.

[31] M. Talbi, M. Mejri, A. Bouhoula. Specification and evaluation of polymorphic shellcode properties using a new temporal logic. In *Journal of Computer Virology*, Springer Paris, Vol.6, 2010.

[32] M. Polychronakis, K.G. Anagnostakis, E.P. Markatos. An Empirical Study of Real World Polymorphic Code Injection Attacks. Available at http://www.usenix.org/event/leet09/tech/full_papers/polychronakis/polychronakis_html/ (accessed on 2009/05/15).

[33] K. Borders, A. Prakash, and M. Zielinski. Spector: Automatically analyzing shell code. In *Proceedings of the Annual Computer Security Applications Conference*, pages 501-514. ACSAC, 2007.

[34] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iFRAMEs point to us. In *Proceedings of the 17th USENIX Security Symposium*, pages 1-16. USENIX, 2008.

[35] David J. Day, Zhengxu Zhao, Minhua Ma. Detecting Return-to-libc Buffer Overflow Attacks Using Network Intrusion Detection Systems. In *Fourth International Conference on Digital Society*, pages 172-177. ICDS, 2010.

[36] I. Kim, D. Kim, B. Kim, Y. Choi, S. Yoon, J. Oh, J. Jang. An architecture of unknown attack detection system against zero-day worm. In *Proceedings of the 8th conference on Applied computer science*, Venice. ACM, 2008.

[37] K. Tatara, Y. Hori, K. Sakurai. Polymorphic Worm Detection by Analyzing Maximum Length of Instruction Sequence in Network Packets. In *International Conference on Availability, Reliability and Security*, pages 972-977. ARES, 2009.

[38] US-CERT Vulnerability Notes Database. Available at http://wwww.kb.cert.org/vulns (accessed on 2010/07/07).

[39] H. Etoh and K. Yoda. Protecting from Stack-Smashing Attacks. Available at http://www.trl.ibm.com/projects/security/ssp/main.html (accessed on 2010/06/06).

[40] Q. Zhang, D. S. Reeves, P. Ning, and S. P. Lyer. Analyzing network traffic to detect self-decrypting exploit code. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, pages 4-12. ASIACCS, 2007.

[41] CLET polymorphic engine. Polymorphic Shellcode Engine Using Spectrum Analysis available at http://www.phrack.org/issues.html?issue$\bar{6}$1&id=9 (accessed on 2009/05/15).

[42] Michal Piotrowski. How to create polymorphic shellcode available at http://hakin9.org/article-html/9374-how-to-create-polymorphic-shellcode (accessed on 2009/03/05).

[43] A. Papadogiannakis, M. Polychronakis, E.P. Markatos. Improving the accuracy of network intrusion detection systems under load using selective packet discarding. In *Proceedings of the Third European Workshop on System Security, EUROSEC '10*. ACM, 2010.

[44] Advanced Shellcode Techniques. Available at http://projectshellcode.com/aggregator/categories/2 (accessed on 2010/01/05).

[45] Michal Piotrowski. Optimization des shellcodes sous linux. Available at www.linux-pour-lesnuls.com/shellcode.pdf (accessed on 2009/04/15).

[46] Jonathan Salwan. How to create polymorphic shellcode. Available at http://www.packetstormsecurity.org/papers/shellcode/how-to-create-polymorphic-shellcode.txt (accessed on 2009/05/01).

[47] Daniele Mazzocchio. Self modifying shellcode. Available at http://www.kernel-panic.it/security/shellcode/Linux_BSD_Shellcode.pdf (accessed on 2009/07/15).

[48] Phantasmal Phantasmagoria. On polymorphic evasion. Available at http://www.securityfocus.com/archive/1/377339 (accessed 2009/07/15).

[49] M.V. Gundy, D. Balzarotti, G. Vigna. Catch Me, If You Can: Evading Network Signatures with Web-based Polymorphic Worms. In *Proceedings of the First USENIX Workshop on Offensive Technologies (WOOT)*. USENIX, 2007.

[50] SANS. SANS Top 20. Available at http://www.sans.org/top-cyber-security-risks/?ref=top20 (accessed on 2010/04/05).

[51] MPack. Mpack. Available at http://www.pandasecurity.com/homeusers/security-info/tools/reports/ (accessed on 2010/04/15).

[52] J.R Crandall, Z. Su, S.F. Wu, F.T. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*. ACM, 2005.

[53] P. Akritidis, E.P. Markatos, M. Polychronakis, K. Anagnostakis. Stride: Polymorphic sled detection through instruction sequence analysis. In 20th *IFIP International Information Security Conference*. IFIP, 2005.

[54] Longest Common Sequence. LCS. Available at http://en.wikipedia.org/wiki/Longest_common_subsequence_problem (accessed on 2009/07/07).

[55] Smith-Waterman Algorithm. Smith-Waterman Algorithm. Available at http://en.wikipedia.org/wiki/Smith-Waterman_algorithm (accessed on 2009/07/15).