12-2009

Statistical Tools for Linking Engine-Generated Malware to Its Engine

Edna Chelangat Milgo
Columbus State University

STATISTICAL TOOLS FOR LINKING ENGINE-GENERATED MALWARE TO ITS ENGINE

Edna Chelangat Milgo

Columbus State University

The College of Business and Computer Science

The Graduate Program in Applied Computer Science

Statistical Tools for Linking Engine-generated Malware to its Engine

A Thesis in

Applied Computer Science

by

Edna Chelangat Milgo

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

December 2009

I have submitted this thesis in partial fulfillment of the requirements for the degree of Master of Science

DECEMBER 1, 2009	[illegible]
Date	Edna Chelangat Milgo

We approve the thesis of Edna Chelangat Milgo as presented here.

December 1, 2009	[illegible]
Date	Mohamed R. Chouchane Assistant Professor of Computer Science, Thesis Advisor
12-01-2009	[illegible]
Date	Edward L. Bosworth, Associate Professor of Computer Science
12/01/2009	[illegible]
Date	Jianhua Yang, Associate Professor of Computer Science
12/01/2009	[illegible]
Date	Lei Li, Associate Professor of Computer Information Systems Management

ABSTRACT

Malware-generating engines challenge typical malware analysts by requiring them to quickly extract, and upload to their customers' machines, a signature for each of a possibly vast number of never-before-seen malware instances that an engine can generate in a short amount of time. In this thesis we propose and evaluate two methods for linking variants of engine-generated malware to its engine. The proposed methods use the *n*-gram frequency vector (NFV) of the opcode mnemonics of an engine-generated malware instance as a feature vector for the instance. An NFV is a tuple that maps *n*-grams with their frequencies. The in-formation contained within the NFV of an engine-generated malware instance is then used to attribute the instance to the engine. The first method implements a Bayesian-like classifier that uses 1-gram frequency vectors of programs as feature vectors. This method was successfully evaluated on a sample of benign programs and one of malicious programs from the `W32.Simile` family of self-mutating malware. The second method, which is an extension of the first method, uses optimized 2-gram frequency vectors as feature vectors and classifies malware by computing its proximity to the average of the NFVs of instances known to have been generated by a known engine. The second method was successfully evaluated on four malware-generating engines: `W32.Simile`, `W32.Evol`, `W32.NGCVK`, and `W32.VCL`. The evaluation yielded a set of four 17-tuples of doubles as signatures for each of the engines, and achieved a 95% discrimination accuracy between a sample of benign programs and samples of malware instances that were generated by these engines. Accuracies of 94.8% were achieved for engine signatures of size 6, 8 and, 14 doubles. We also used four *k*-nn classifiers which, unlike the second method, require the time-consuming task of creating and storing one signature per known malware instance, to countercheck the accuracies achieved by the second method. This work is inspired by successful methods for attributing natural language texts to their respective authors. The proposed methods may be viewed as filtering (or decision support) tools that malware detectors may use to determine whether extensive engine-specific program analyses such as emulation and control flow analysis are needed on a suspect program.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEGMENTS

I owe my deepest gratitude to my thesis advisor, Dr. Mohamed R. Chouchane, for introducing me to the field of malware and statistical applications and his guidance and patience throughout the thesis design and implementation. Great thanks to Dr. Edward Bosworth, Dr. Jianhua Yang and Dr. Lei Li for being on my thesis committee and for their help in improving my oral presentation and thesis document. My heartfelt appreciation goes to the TSYS School of Computer Science chairperson, Dr. Wayne Summers, together with all the faculty who molded me in various academic areas and supported me while attending various academic conferences. Great thanks to my former academic advisor, Dr. Yan Bai, for laying the foundation for my graduate school research. Great thanks to Ms. Yasmine Kandissounon and Ms. Jessica Turner who supplied the `W32.NGVCK` and `W32.VCL` variants and the original `W32.Simile` variant.

I am indebted to Ms. Dorothy Cheruiyot for her wise advice and for helping me realize my dream of going to graduate school. I am also grateful to my parents and my eight siblings who have always been there for me. Finally my heartfelt thanks to everyone whose name I did not mention here but in one way or another helped me to successfully complete this thesis.

Chapter 1

Introduction

This chapter presents the general problem of malware detection. We first give a definition for the different types of malware, and then we give an overview of the traditional methods for detecting malware and how engine-generated malware challenges current detectors. Finally we enumerate the methods proposed in this thesis to improve the accuracy and speed of malware detectors at attributing engine-generated malware to its engine.

1.1 A Brief Look at Malware

Malware, short for malicious software, is a program designed to potentially disrupt the normal functioning of a system in which it executes [8, 9]. The term computer virus is commonly used to mean malware, but the definition of malware expands to include other forms of malicious programs. These programs include computer worms, Trojan Horses, adware and spyware.

1.1.1 Classification

A computer *virus* is malware that, when executed, tries to attach itself to other files without the knowledge of the user [2, 9]. *Brain* is one example of an early computer virus [22]. A computer *worm* is standalone malware (i.e., it does not need to attach itself to other files) that is able to spread across a network [2, 29]. A common example of this type of malware is the 2003 SQL Slammer worm which spread fast and affected millions of computers in a short period of time. In addition to causing damage to the infected system, worms also consume a lot of bandwidth. A *Trojan Horse* is malware that appears benign but may perform malicious actions that could allow unauthorized access to the system on which it was installed. *Adware* is malware that generates unsolicited ads usually through pop ups. *Spyware* is malware that attempts to capture personal information and transmit it to the attacker. Detailed definitions of these and other classes of malware are given by Ször in [30] and by Aycock in [2].

Malware has evolved through various stages in an attempt to evade detection and be able to execute its malicious payload. *Armored viruses* are those viruses which use *code obfuscation* to make their code harder to reverse engineer, without altering its functionality [30]. Figure 1.1 shows an example of an obfuscating transformation, where a predicate is crafted so that its value is always 'true' and so that the obfuscator is able to insert an unreachable set of code fragments.

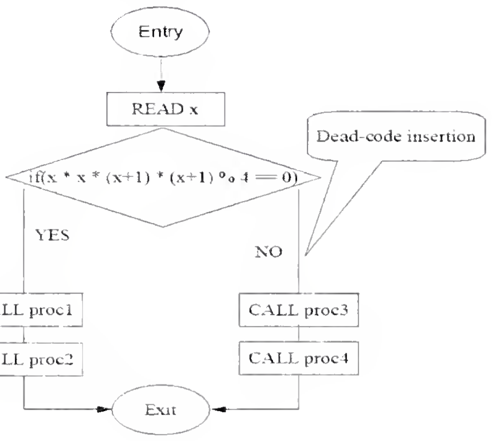


Figure 1.1: Code obfuscation: The predicate may be crafted so that the 'YES' path will always be taken. The 'NO' path is dead code that has been inserted for the sole purpose of hardening static analysis.

Encryption of a malware's binary is another form of obfuscation where the intent is to harden disassembly or make it impossible. Malware that encrypts its own code is often called *polymorphic malware* [30]. Kaspersky Lab determined that almost all of today's malware are polymorphic, making it difficult to develop detection and disinfection procedures within an acceptable time frame [14].

A *morphing engine* is a program whose purpose is to change the appearance or the semantics of the malware. Malware which carries its own engine is called *metamorphic malware* [5, 8, 24]. A breakdown of the structure of a metamorphic malware is given by Walenstein *et al.* [3, 34]. Malware which invokes a remote engine with the malware code as input in order for the engine to transform its appearance or semantics is called *morphing malware* [7]. (See Figure 1.2).

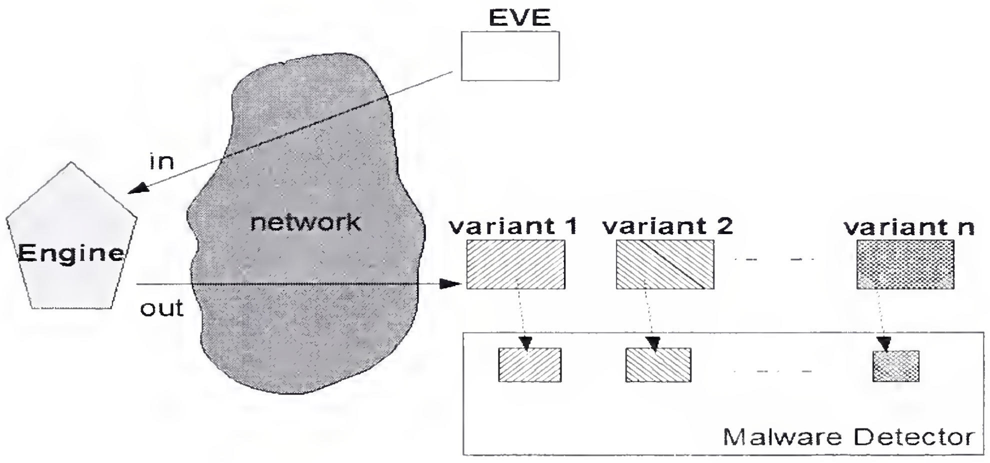


Figure 1.2: Morphing Engine: On input a malware instance, the engine is able to generate arbitrarily many variants of that instance.

Metamorphic malware is henceforth viewed as a subclass of morphing malware. A *variant* of a morphing malware is a program that was generated by a malware-morphing engine [2, 8]. This work specifically targets malware, of any of the above classes, that is generated by a machine, not by a human.

1.1.2 Current State of Threat

The last several years marked the start of a new period characterized by a rapid increase in the number of malicious programs. Figure 1.3 shows the sharp increase in the number of malware variants that was observed over the last few years by *F-Secure*, an anti-virus company.

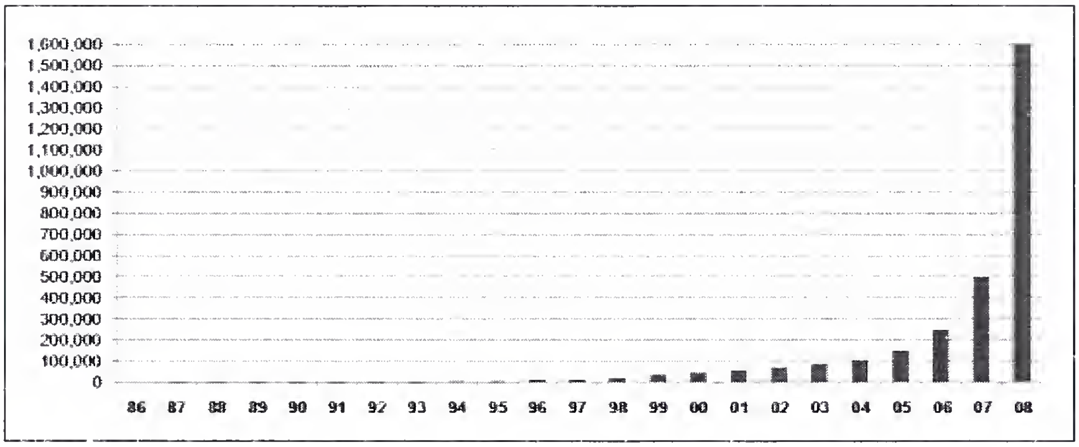


Figure 1.3: Malware growth per year, the number of malicious programs tripled in 2008. Source F-Secure, 2008 Q4 [13].

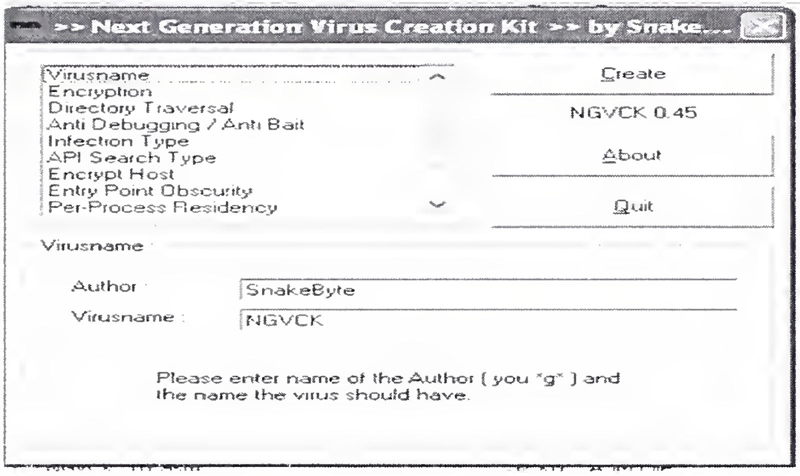
According to Kaspersky *Lab* [14] there is separation of the design tasks by the malware authors, where different people participate at the different stages of malware design, and later combine the work hence making it complex. Microsoft Security Intelligence Report determined that there was a large increase in the number of worms detected across the world in the first half of 2009. It also reports that computers in corporate environments were more likely to be infected by worms than home computers. According to Microsoft Security Intelligence Report, malware creators often release variants for a family which are typically used in an effort to avoid being detected by security software. Another factor that has contributed to a fast increase in the level of malware threats is the availability of underground funding for malware hosting and distribution, which made malware more accessible to cyber criminals [25].

There were approximately of 5.5 million malware instances analyzed in 2007, which on average involved analyzing between 15,000 and 20,000 new malware instances each day. This is more than four times the average number of malware instances analyzed per day, compared to 2006 [31]. Another reason that has contributed to this rapid increase is the hiring of professionals by malware shops. These professionals also do quality assurance to ensure the complexity of the software [4]. Malware authors automate the creation and spread of malware to ensure that the malware spreads fast and to challenge manual malware analysis [4].

Anti-virus companies, in an attempt to cap the rapid growth in volume, have come up with generic detectors to detect a wider range of "related" malware instances. This approach, though faster than using one signature per malware instance, has resulted in high rates of false positives, where benign programs were mistaken for malicious. An example of a false positive is when Kaspersky Lab misclassified Windows Explorer as malicious in December 2007. Team-Cymru, a malware research firm, submitted 1000 samples of malware to 32 different commercial antivirus firms. Out of these 1000 samples, 630 samples went unnoticed [10].

1.1.3 Engine-generated Malware

Engine-generated malware is that which has been generated by a machine, perhaps after being given some user input. Malware generating engines include the so-called "virus generation toolkits" that are often available for free download by anyone over the Internet, as well as morphing engines. We continue our discussion of morphing engines to further illustrate the concept of a malware-generating engine.



Author : SnakeByte
Virusname : NGVCK

Please enter name of the Author (you *g*) and the name the virus should have.

Figure 1.4 : The Next Generation Virus Creation Kit.

Morphing engines use source-to-source transformations such as code substitution (e.g., code expansion and code compression), garbage insertion, code permutation, and register renaming to modify a malware instance's appearance without necessarily preserving its functionality [34]. Code expansion (see Figure 1.5 for an example) is a transformation where one code segment is replaced with a larger code segment. An indirect call instruction, for instance, can be morphed by computing the sum S of two variables on the program's stack, which are known beforehand, and then jumping to the location pointed to by S. Code compression (see Figure 1.5 for an example) is a transformation where a code segment is replaced with a smaller code segment. Code compression also aims at eliminating garbage instructions from the code. Code permutation (see Figure 1.5 for an example) makes use of direct jump instructions to reorder the lines of code while maintaining the control flow of the program being transformed. Garbage insertion (see Figure 1.5 for an example) adds do-nothing instructions to the malware code.

Examples of existing metamorphic malware include `W32.Simile`, `W95.Zmist` and `W32.Evol`. Examples of malware generation toolkits include `W32.NGCVK` and `W32.VCL` [11, 27, 29]. The "very user-friendly" interface of `NGVCK` is shown in Figure 1.4.

1.1.4 Malware Detection

Malware detectors (also referred to as anti-virus scanners) are programs whose main task is to search their host computing system for the presence or absence of malware [5]. They may also be invoked manually or automatically to determine whether a given program is malware.

Since the size of malware code may range from just a few lines of code to a large number of lines of code, including multiple procedure calls, malware detectors typically use malware signatures (also called malware definitions) to detect malware. A malware signature is a sequence of bytes and/or patterns that occur in malware's binary and that can be used to uniquely identify the binary. A malware *signature* should be specific to the identified malware or malware family so as to avoid false positives and false negatives.

Anti-virus analysts need to have an up-to-date knowledge of the current vulnerabilities and exploits which could be targeted by malware authors [29], this way a signature can be custom-built to recognize patterns in the malware code which indicate that the code may be able to exploit a given vulnerability. They sometimes rely on submissions by individuals and organizations of new malware samples for which signatures may not yet have been extracted by the anti-virus analysts. Ször describes a number of approaches used by anti-virus analysts to analyze malware, extract a signature for it, and then use the signature for detection purposes [29]. Malware detection is discussed in more detail in Chapter 2.

1.2 Challenges

In his pioneering work on computer viruses, Cohen formally proves that no malware detector can possibly be constructed that can perfectly detect all the current and future malware instances in a finite amount of time [9]. One other, more practical, challenge that anti-virus analysts face is the need to provide a signature in a timely manner while keeping up with the rate at which never-before-seen malware instances show up in the wild, since malware signature databases must be updated frequently to add signatures for newly released malware [33]. One condition for a malware detector to be efficient is that the malware signature database be as small as possible in order to reduce the time taken to update it and scan it. Also, if a malware signature database is large and requires an inordinate amount of space and time to search and update, a user might be tempted to ignore requests by the malware detector to update its database or to scan the computer for the presence of malware.

In this thesis we propose a method for efficiently attributing engine-generated malware to its engine, which we hope will address the challenge of having to store one signature for each of a possibly vast number of malware instances that a given engine may be able to autonomously produce in a short amount of time.

Code Expansion

```
mov eax,16                  mov eax,10
                            add eax,6
push eax                    push eax
                            push eax
mov eax,32              →   mov eax,32
mov eax,1024                mov eax,1024
mov edx,32                  mov edx,36
                            sub edx,4
push eax                    push eax
```

Code Compression

```
mov[esi+4],6     mov[esi+4],9
add[esi+4],3
push eax      →  mov [ebp+8],eax
mov eax,ecx
mov[ebp+8],eax
pop eax
```

Code Permutation

```
push ebp                    jmp L1:
mov ebp,esp                 L2: mov edi,dw ptr[ebp+08]
mov edi,dw ptr[ebp+08]→jmp L3:
test esi,esi                L1: mov ebp,esp
mov edi,dw ptr[ebp+0C]      jmp L2:
or edi,edi                  L3: test esi,esi
xor edx,edx                 mov edi,dw ptr[ebp+0C]
                            or edi,edi
                            xor edx,edx
```

Garbage Insertion

```
push ebp         push ebp
                 nop
              →  nop
mov ebp,esp      mov ebp,esp
pop esp          pop esp
```

Figure 1.5: Morphing Transformations: A selection of the source-to-source transformations used by morphing engines to change the appearance of malware in-stances.

1.3 Our Contributions

This thesis makes the following contributions.

- We propose and evaluate a method for discriminating between variants of the `W32.Simile` metamorphic malware and a sample of benign program. The method uses an optimized instruction frequency vector of a program as a feature vector for the program. The method is successfully evaluated, with a success rate of 100%, for optimized instruction frequency vectors composed of only 4 real numbers.

- We propose and evaluate a method for attributing malware generated by the `W32.VCL` engine, the `W32.NGVCK` engine, and the engines of `W32.Simile` and `W32.Evol`, to the engines. The method uses as feature vector for a program the (possibly optimized) n-gram frequency vector of the program's opcode mnemonics, for some positive integer n. The method is evaluated for $n = 2$, with a success rate of 95% for optimized bigram frequency vectors composed of only 17 doubles. An accuracy of 94.8% was achieved for 6, 8 and 14 doubles. The method asks that only the average of the optimized bigram frequency vectors of known variants be used as a signature for the engine that generated the variants.

The proposed methods only require the malware detector to disassemble a suspect program before feeding it into a classifier that implements the proposed methods.

1.4 Impact of Our Contributions

The proposed methods are expected to improve those detectors that use static and dynamic program analyses to extract malware signatures in the following ways:

- Both methods were time efficient. A full experimental evaluation of the methods (involving 500 different subjects) completed in 4 minutes. This is a relatively good time, given the fact that the program was running on a Java virtual machine.

- The second method allows the detector to be more space efficient. The space required to store the best performing signature is the space needed to store 17 doubles.
- The detection accuracies of both methods, especially that of the second method, suggest that the large body of work in forensic linguistics may be worth tapping into. The proposed methods where inspired by works which have been successfully used by forensic linguists to attribute natural language texts to their corresponding authors.

1.5 Organization of this Thesis

Chapter 2 gives a detailed statement of the problem of detecting engine-generated malware. Existing approaches for detecting engine-generated malware are investigated and their contributions contrasted with those proposed in this thesis to attribute engine-generated malware to its engine. Chapter 3 describes and evaluates the first detection method proposed in this thesis. This method uses a program's instruction frequency vector to link the program to a known malware-generating engine. Chapter 4 describes and evaluates the second method that this thesis proposes to attribute engine-generated malware to its engine. This method uses a program's *n*-gram frequency vector, for some positive integer *n*, to attribute the program to its engine. Chapter 5 concludes this thesis and outlines directions for further work.

Chapter 2

The Detection Problem of Engine-generated Malware

In this chapter, we visit some of the existing methods for detecting engine-generated malware. Each of these methods falls under one or more of the (1) generic model, (2) the normalization model, and (3) the engine signature model for detecting engine-generated malware. We then provide a synopsis of our proposed detection methods, which fall under the engine signature detection model of engine-generated malware.

2.1 Existing Detection Methods

In order to eventually detect a malware signature, a malware detector can start by statically analyzing a suspect program, dynamically analyzing it, or both.

- **Static analysis.** A suspect program is statically analyzed by going through one or more of the normal steps in the program analysis pipeline. This is usually done by disassembling a suspect program, extracting its control flow graph, and then searching the control flow graph for sub-graphs that may signal a malicious intent [8, 11, 21]. Static program analysis is inherently hard and may be challenged by code obfuscation (see Figure 1.1). Figure 2.1 gives a high-level view of the malware analysis pipeline.

- **Dynamic analysis.** Malware detectors may also run a suspect program in a virtual environment that simulates some computing platform. Malware emulation is the use of a virtual machine to run a suspect program and monitor its behavior. An emulator may execute a suspect program a given number of times with different inputs, but the emulation process could become lengthy in a situation where the program has to be emulated in a real-time environment or if a malware instance attempts to test the patience of the emulator by entering a do-nothing loop that will runs for a random amount of time [30].

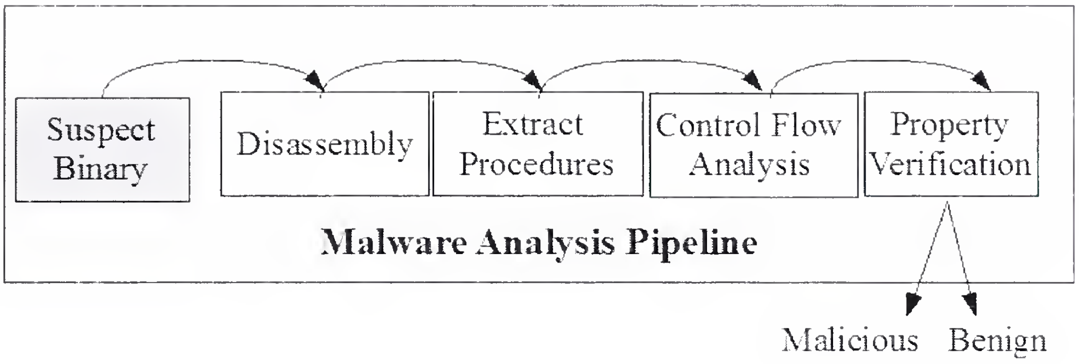


Figure 2.1: Malware Analysis Pipeline

Program analysis tools have enabled malware detectors to fairly accurately discriminate between malicious programs and non malicious ones [7, 23], and any effort made by a malware detector to detect malware will require a certain amount of program analysis. The remainder of this section will describe those detection models (which also require that a certain amount of program analysis be applied by the detector to suspect programs) that were specifically designed to detect members of known malware families, including engine-generated malware.

2.1.1 Generic Detection

The generic detection model is that which requires a malware analyst to analyze a suspect program by looking for those patterns that the malware analyst expects all of the variants of that suspect program to have. This model has initially been devised to detect variants of malware that have been manually generated by slightly altering an existing malware instance, usually to fix a bug within the instance, or to modify the instance's malicious behavior. This model is certainly applicable to those engine-generated malware instances, provided the engine behaves in a manner similar to that of a human, by just altering some "non-essential" fragments of the malware code to fix a bug, or by slightly modifying the malware's malicious behavior. Due to the high level of complexity, for a human's point of view, of assembly language programs, humans typically prefer making slight alterations to malware code (which is often written in assembly), over major overhauls.

Since it relies on the assumption that malware variants are no more than slight modifications of the other variants, generic detection is not a good match for the problem of detecting malware whose appearance and/or semantics may be drastically changed by morphing engines. Engines, after all, are much better than humans (think compilers) at making conservative decisions about the control flow of a program, and at deciding that the insertion or deletion of a given set of code fragments from an assembly language program will modify the input/output behavior of the program in unintended ways.

Geometrical detectors use the effects caused by a viral infection to the size of an infected file (e.g. an attachment to a system file or a sudden increase in the file size). Since system files have known sizes, a change in their sizes may be interpreted as an infection. This method effectively detects variants of the `W95/ZMist` metamorphic virus but is ineffective in the detection of non parasitic malware [29].

The generic detection model has actually not been successful at dealing with the variant generation problem, even for those malware variants that were manually generated by slightly modifying existing malware instances. *AVTest Labs*, a German consulting company that specializes in computer security, has reported in 2008 unacceptably high levels of false positives and false negatives generated by commercial malware detectors that use the generic detection model for detecting member of malware families. Their main concern was that there was no systematic, scientifically testable way of verifying whether a given generic signature, extracted in a lab by a human analyst, perhaps relying on a set of "hunches" and with the help of a set of reverse engineering tools, would produce a single generic signature that is representative of a large enough (say, 90% or above) portion of a malware family.

2.1.2 Normalization

Normalizers are programs that take as input a suspect program, simplify it in some fashion (e.g., by eliminating garbage instructions), and then analyze the output of the normalizer to determine whether its output is similar to that of a known malware. Normalizers aim to reduce the size of the signature space needed to store a signature for each instance of engine-generated malware by assuming that multiple instances would be "simplified" by the normalizer into a small set of normal forms [32]. Since the normalizer construction problem is known to be unsolvable in general [32], existing methods for normalizing members of a given malware family may not always scale the problem of normalizing the members of any malware family. Figure 2.2 gives a pictorial representation of the normalization model for detecting malware.

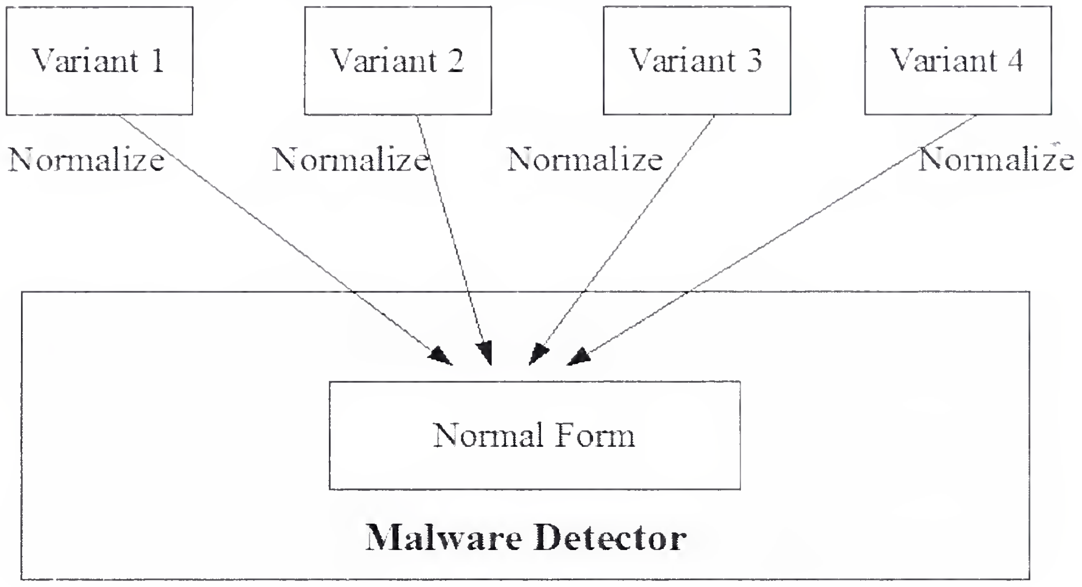


Figure 2.2: Normalization: A procedure is constructed that efficiently reduces each malware instance to a normal form. Instances that belong to a given malware family are expected to be reducible to the same normal form.

2.1.3 Engine Signatures

The engine signature model for detecting malware [5, 6] asks that forensic evidence be extracted from a suspect program that can be used to link the suspect program to a known malware-generating engine. This model was inspired by the success that forensic linguists had in attributing, to their authors, documents written in some human language such as English. Probabilistic and statistical methods been suggested to attribute morphed malware variants to the morphing engine that generated them. Chouchane *et al.* [7] suggested that Markov chains theory be used to model the morphing process of malware variants, and then construct a filter for engine-generated malware that uses the morpher transition matrix as a signature for the morpher.

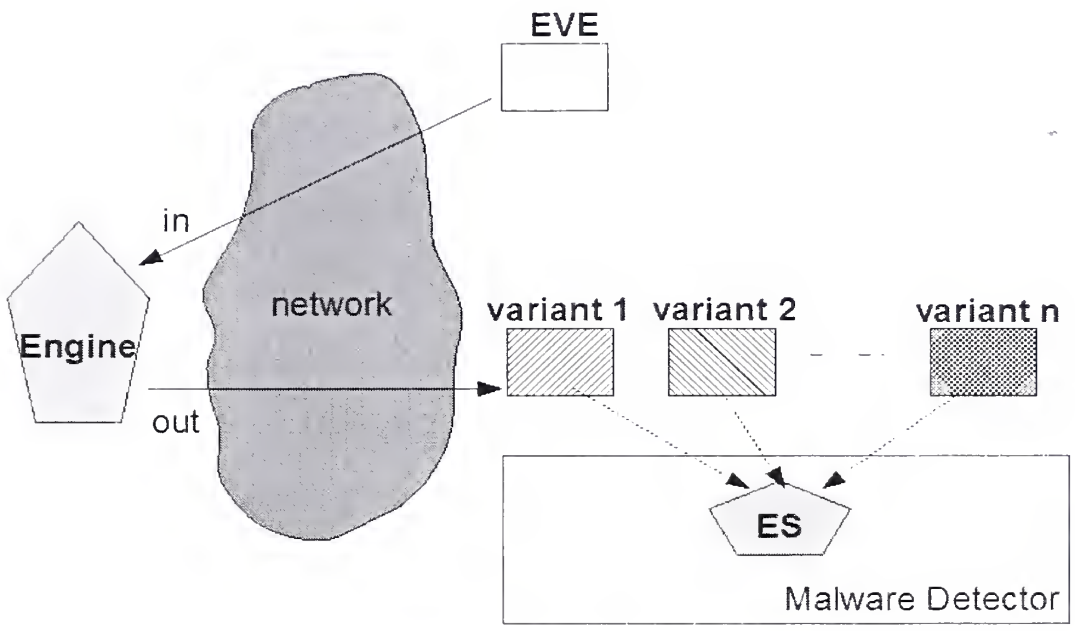


Figure 2.3: Engine Signature: Instead of storing and managing a signature for each malware variant. A single signature, that of the engine that generated the variants, is used to detect the variants.

Other statistical approaches use as feature vectors a code fragment`s sequence of system calls that is known to be executed by members of a given malware family [26]. These methods work well when the system calls are not obfuscated but, since they make the assumption that the system calls used as a signature will not be changed by the morpher, they are not of much help in detecting engine-generated malware where the engine may or may not preserve the malware`s functionality.

2.2 IFVs and NFVs as Feature Vectors

The proposed detection methods fall under the engine signature detection model of engine-generated malware; they provide fast, approximate tools for attributing engine-generated malware to the engine. The following metrics are used to construct a program's feature vector:

1 A program's instruction frequency vector: This is the vector each of whose components holds the frequency, in the program, of a given opcode mnemonic in the instruction set architecture of the machine on which the program's binary is runnable. A program's instruction frequency vector is analogous to the word frequency vector of a document written in some human language. Word frequency vectors have been shown to contain enough information about a document's human author to enable a forensic linguist to tell what human author, in a given set of authors, has written the document whose word frequency vector is being examined [18].

2 A program's *n*-gram frequency vector: This is the vector each of whose components holds the frequency, in the program, of a given *n*-gram, for some fixed positive integer *n*, of opcode mnemonics that occur in the instruction set architecture of the machine on which the program's binary is runnable. A program's *n*-gram frequency vector contains more information about the program than the program's instruction frequency vector, since an *n*-gram frequency vector also capture the rate at which the program's author tend to append a given opcode mnemonic to any given sequence of *n*-1 opcode mnemonics. Forensic linguists have conducted

a successful empirical evaluation of an authorship attribution method that used word *n*-gram frequency vectors to attribute English documents to their authors [18].

Optimization choices had to be made by us to reduce the size of a program's instruction frequency vector and *n*-gram frequency vector, as well as to allow us to conduct our extensive experimental evaluations, which involved 500 different programs of various sizes, so that they terminate within a reasonable amount of time. Subsequent chapters elaborate on each of the above methods, describe the experiments that we have conducted to evaluate them, and discuss the outcome of each of these experiments.

Chapter 3

Using IFVs to Detect Engine-generated Malware

This chapter describes the first method that we have successfully tried to attribute engine-generated malware to the engine that produced it. This method uses a statistic about a suspect program, namely its instruction frequency vector (IFV), to determine whether the program is engine-generated malware that has been generated by a known malware generating engine. This method was successfully used to discriminate between variants of the `W32.Simile` malware and benign programs.

3.1 Motivation

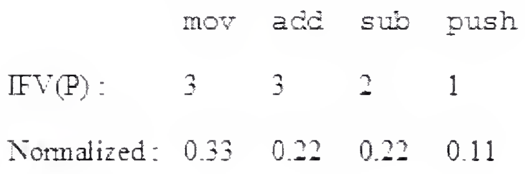
The detection method described below only requires the malware's binary be disassembled, saving the malware detector the trouble of having to run potentially time consuming program analyses on a suspect binary. Since the IFV of a program does not change should the code permutation transformation be applied to the program, the detection accuracy is insensitive to the code permutation transformation that a morphing engine may use, in addition to other source-to-source transformations, to change the appearance of its input malware variant. The method also reduces the space normally needed to store a signature for each variant by only requiring that the IFV's of a sample of malware instances known to have been generated by a given engine be used to determine whether a suspect program has been generated by that engine.

3.2 Approach

An *Instruction Frequency Vector* (IFV) is a vector that maps opcode mnemonics with their frequencies in an assembly language program. The frequencies of these opcode mnemonics are recorded as entries in the IFV. IFVs are normalized so that the relative frequencies of the opcode mnemonics are considered instead of the absolute ones. A program's normalized IFV is then used as a feature vector for the program. Consider the following program *P*, reduced to its sequence of opcode mnemonics.

```
P :   mov,add,mov,add,sub,push,add,mov,sub
```

Figure 3.1 shows the IFV of *P*. We will henceforth use the acronym IFV to refer to a program's normalized instruction frequency vector.



	mov	add	sub	push
IFV(P) :	3	3	2	1
Normalized :	0.33	0.22	0.22	0.11

Figure 3.1: Computing the Instruction Frequency Vector of P.

We measure the distance between two IFVs using the following distance measure.

$$d(IFV_x, IFV_y) = \sqrt{\sum_{i=0}^{n-1} \left((IFV_x)_i - (IFV_y)_i\right)^2},$$

where IFV_x and IFV_y are instruction frequency vectors of program x and program y, respectively.

Let L(X; S) denote the likelihood that a suspect program *X* is `W32.Simile` given that a number of `W32.Simile` instances are in the vicinity of *X*. *L(X; S)* is expressed as follows:

$$L(X,S) = \frac{|s \,\mathcal{E}\, Sim : d(IFV_x, IFV_s) \leq \mathcal{E}\,|}{Number\ of\ trainers\ within\ \mathcal{E}\ of\ X}$$

Let *L(X; B)* denote the likelihood that a suspect program *X* is benign given that a number of benign instances are in the vicinity of *X*. *L(X; B)* is expressed as follows:

$$L(X,B) = \frac{|s \,\mathcal{E}\, Bgn : d(IFV_x, IFV_b) \leq \mathcal{E}\,|}{Number\ of\ trainers\ within\ \mathcal{E}\ of\ X}$$

A discriminator between benign programs and malware generated by `W32.Simile`'s engine can then be designed to operate as follows:

1. Take as input the IFV of a suspect program *X*.
2. Choose a threshold ε > 0.
3. Find the IFV's of existing benign and `W32.Simile` samples (trainers) that are within of the IFV of the suspect program.
4. Compute the number of `W32.Simile` trainers within of *X*.
5. Compute the number of benign trainers within of *X*.
6. The family that has the highest number of trainers within ε of X is declared to be *X*'s family; if there is a tie, choose one at random.

3.2.1 Evaluation

We downloaded `W32.Simile`'s *eve* program from *vx.netlux.org* [15]. We then extracted the opcode mnemonics from the *eve* and performed the code substitution (expansion and compression), code permutation, and garbage insertion transformations. These transformations simulated those described by The Mental Driller, the author of `W32.Simile`, on how he implemented `W32.Simile`'s morphing engine [16]. This option (i.e., implementing a simulator for the engine) is more efficient and secure than actually running the malware and waiting for it to mutate, which it may or may not do on any given run. One hundred variants of `W32.Simile`'s eve were generated using the simulator.

We collected 100 benign programs. These benign programs were downloaded from *download.cnet.com* [12] and *sourceforge.net* [28]. These programs were disassembled using *Ollydbg* [17] and the IFV for each program was generated.

We grouped all of the benign and malicious IFVs into training and testing sets. Each training set consisted of 50 instances from each sample and the remaining instances were used for testing.

We then evaluated the six-step approach described in the previous section using 40 classifiers, by varying the threshold from 0.1 to 2.0, with an increment of 0.05. IFVs were further optimized by considering only the RI most frequent instructions across the collected samples, for some small positive integer RI, to construct the IFVs.

We report our experimental results for RI=4 and RI=5. Figure 3.2 and Figure 3.3 show the experimental results, in terms of accuracy, false positives, and false negatives, for RI=4 and RI=5, respectively.

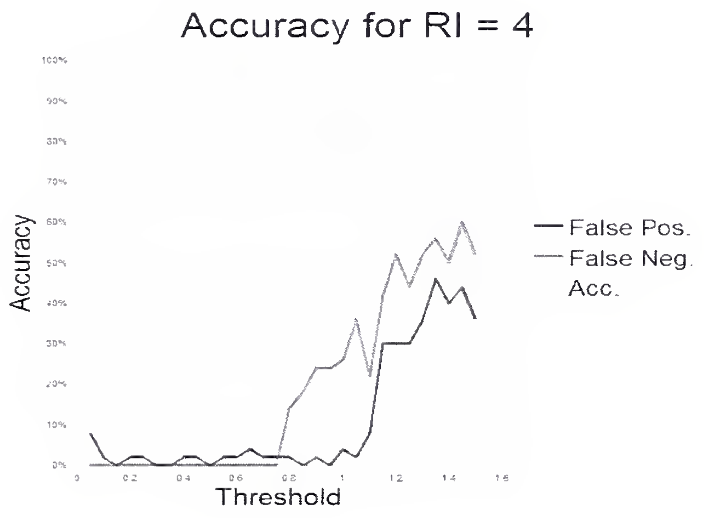


Figure 3.2 Evaluation results of the IFV classifier for RI=4 and $0.1 \leq \varepsilon \leq 2.0$

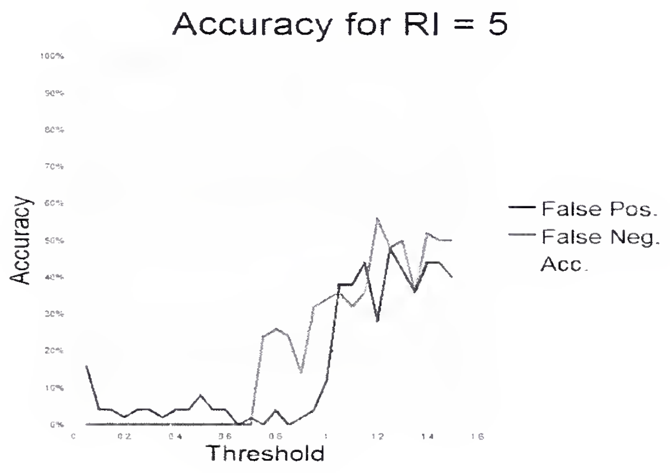


Figure 3.3 : Evaluation results of the IFV classifier for RI=5 and $0.1 \leq \varepsilon \leq 2.0$

3.2.2 Discussion

Some of the classifiers achieved 100% discrimination accuracy between the sample of benign instances and that of the `W32.Simile` instances. This level of accuracy was achieved for RI=4 and $\varepsilon = 0.5$, as well as for RI=5 and $\varepsilon = 0.7$. This is a rather promising result since the number of instructions used to compute the feature vectors is small, hence saving space and computation time. As the threshold approached 2, the classifiers' decision making became not much different than random guessing. This is due to the fact that the Euclidian norm of any (normalized) IFV is less than or equal to 1, placing any IFV within a threshold of no more than 2 of any other IFV.

One limitation of this approach is that it has to visit all of the samples while computing the distance from a suspect program to each member of each sample to identify those members that are within ε of the suspect program.

The next chapter describes and successfully evaluates a more general method for attributing malware to its engine, without the need to store the feature vectors of all of those malware instances that are each known to have been generated by a specific engine.

Chapter 4

Using NFVs to Detect Engine-generated Malware

In this chapter, we describe and then evaluate an approach that uses *n*-grams to attribute engine-generated malware to its engine. In our experiments we evaluate the approach for *n*=2. For *n*=1, the approach is similar to the approach described in Chapter 3. For *n*=2, the approach is similar to that taken by Abou-Assaleh *et al.* [1] to separate malicious programs from benign ones, but not to attribute malware to a known malware-generating engine.

4.1 Motivation

We were motivated to implement this *n*-gram method by the success of a work on attributing human text to its author [18]. With this method, only one signature would need to be computed and maintained to detect members of a given family of malware known to have been generated by a known engine. This approach requires no program analysis beyond disassembly except for malware that is generated in the form of an assembly language program.

4.2 Approach

We use the term *n*-gram to refer to a sequence of *n* opcode mnemonics. For instance, a 2-gram is a sequence of two opcode mnemonics. An *n-gram Frequency Vector* (NFV) is a tuple that maps *n*-grams with their frequencies in a given sequence of *n* or more opcode mnemonics. Consider the following program *P*, reduced to its sequence of opcode mnemonics.

P : `mov,mov,add,mov,add,sub,push,add,mov,sub`

Considering only the 2-grams that are composed of the 3 most frequent opcode mnemonics in *P*, the NFV of *P* is computed as shown in Figure 4.1. The NFV components for the "relevant" bigrams `submov, subadd, and subsub` are not displayed in the figure, since these bigrams do not occur in *P*.

	Movmov	movadd	movsub	addmov	addadd	addsub
NFV(P) :	(1	2	1	2	0	1)
Normalized :	(0.143	0.286	0.143	0.287 0	0	0.143)

Figure 4.1: Computing the bigram frequency vector of P, using only P's three most frequent opcode mnemonics.

A program's NFV is treated as a feature vector (signature) for the program. The signature for a family $(P_i)1 \leq i \leq l$ of programs is computed as follows:

$$Family\ Signature = \frac{\sum_{i=1}^{l} NFV(P_i)}{l},$$

where the + operation on two NFVs is the outcome of the component-wise addition of the components of the operands.

Our distance measure on NFVs is given by:

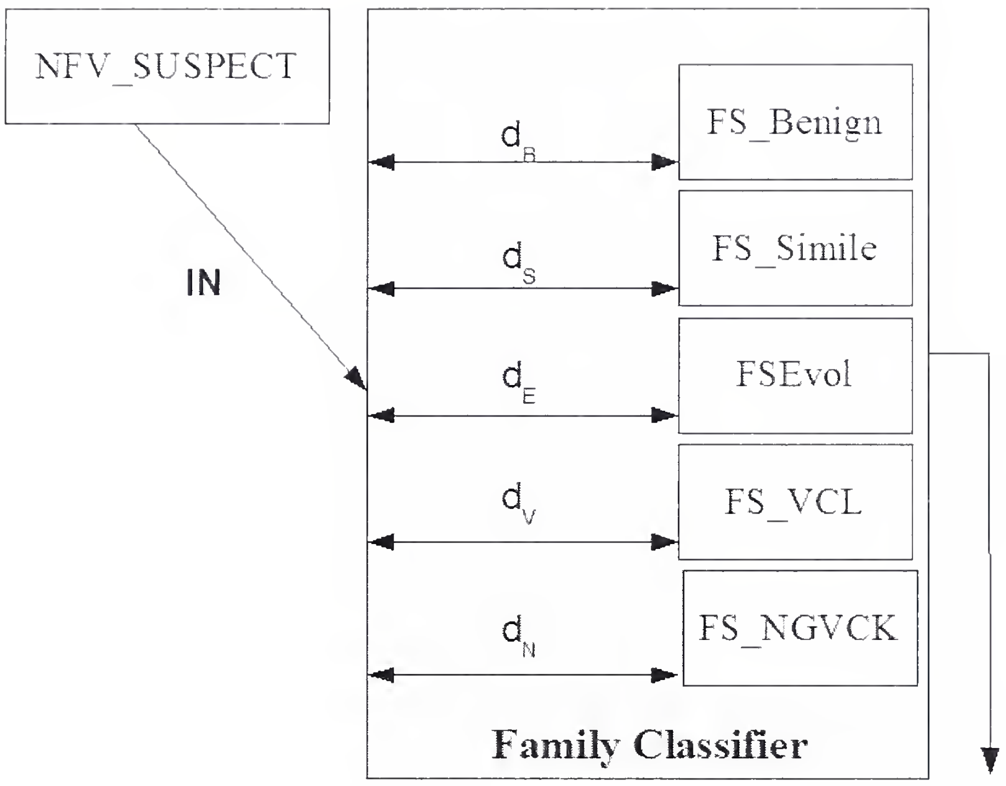
$$d(NFV_x, NFV_y) = \sum_{i=1}^{m^n} \left(\frac{(NFV_x)_i - (NFV_y)_i}{\frac{(NFV_x)_i + (NFV_y)_i}{2}} \right)^2$$

which can also be exprcssed as

$$d(NFV_x, NFV_y) = \sum_{i=1}^{m^n} \left(\frac{2 \cdot (NFV_x)_i - (NFV_y)_i}{(NFV_x)_i + (NFV_y)_i} \right)^2$$

where m is the number of uniquc opcode mnemonics that are considered "relevant" by thc detector. $d(NFV_x, NFV_y)$ computes the dissimilarity betwecn NFV_x and NFV_y. For identical strings of opcode mnemonics, this dissimilarity is 0.

Detection method 1 (Proposed method). In order to determine to which family, in a given set of engine-generated malware families, a suspect program belongs, wc designed a classifier that computes the distance *d* between the NFV of the program and the signature of each family. (The NFV may not necessarily be taken whole; the detector may opt to choose whatever *n*-gram it deems "relevant" to the NFV.) The classifier compares the label of the family that is closest (according to *d*) to the suspect program with the label of the actual family of the suspect program. If the labels are different, then we increment the mismatched counter. If the label of the suspect program is "benign" and that of the closest family is "malicious", then we increment the false positive counter. If the label of the suspect program is "malicious" and that of the closest family is "benign", then we increment the false negative counter. We then use the misclassification rate, false positive rate, and false negative rate to evaluate the classifier.



$$\mathrm{IF}(\min(d_B,d_S,d_E,d_V,d_N) == d_B) \rightarrow \textbf{OUT} = \text{Benign}$$
$$\mathrm{IF}(\min(d_B,d_S,d_E,d_V,d_N) == d_S) \rightarrow \textbf{OUT} = \text{W32.Simile}$$
$$\mathrm{IF}(\min(d_B,d_S,d_E,d_V,d_N) == d_E) \rightarrow \textbf{OUT} = \text{W32.Evol}$$
$$\mathrm{IF}(\min(d_B,d_S,d_E,d_V,d_N) == d_V) \rightarrow \textbf{OUT} = \text{VCL}$$
$$\mathrm{IF}(\min(d_B,d_S,d_E,d_V,d_N) == d_N) \rightarrow \textbf{OUT} = \text{NGVCK}$$

Figure 4.2: Engine Signatures: The signatures of the engines are used to attribute malware instances to known engines.

(k-nn). We also used a *k*-nearest neighbor classifier (*k*-nn). A *k*-nn classifier is an instance based classifier that has been shown to be powerful enough for most classification problems [20, 35]. Given a training set *T* of malicious programs, the NFVs of all the programs in *T* are labeled and then stored in a set *S*. The distances between a suspect program's NFV and each of the NFVs in *S* are computed; the *k* nearest ones, for an *a-priori* chosen *k*, are then selected to vote. The family, be it one of benign programs or one of engine-generated malware, that has the majority of the votes is declared by the classifier to be that of the suspect program. Where there are ties a winner is selected at random.

4.2.1 Evaluation

We collected 100 instances of each of the following families: `W32.Simile`, `W32.Evol`, `W32.VCL`, `W32.NGVCK` and benigns.

The proposed approach was evaluated by using as feature vectors for the collected programs the 2-gram frequency vectors of the programs. These frequency vectors were not used whole to evaluate the classifiers, instead we experimented with two strategies for choosing the most relevant bigrams for the collected families of programs.

1. **RI.** Consider only those bigrams that are composed of any two of the RI most frequent opcode mnemonics across the collected programs, for some small positive integer RI.
2. **RB.** Consider only those RB most frequent bigrams across the collected programs, for some small positive integer RB.

We divided each family into a training set of size 90 and a testing set of size 10. Our first classifier (implementing the proposed method) used the average of each family's NFVs as the family signature. Our second classifier (implementing *k*-nn) was evaluated for $k = 1$ to 20. For each of the classifiers we performed a 10-fold cross validation [19]. We used a new testing set each time. Letting A_i denote the classification accuracy for each of the ten runs of the 10-fold cross validation, we took the average of the A_i's as a performance measure for the classifiers. $Cross\ Accuracy = \frac{A_1+A_2+\ldots A_{10}}{10}$

Figure 4.3 shows the achieved classification accuracies (using the RI strategy) of the proposed classifier, as well as the *k*-nn classifier for *k*=1, 5, 10, 15, and 20.

	FS	**1-nn**	**5-nn**	**10-nn**	**15-nn**	**20-nn**
RI = 1	0.200	0.200	0.200	0.200	0.200	0.200
RI = 2	0.252	0.400	0.400	0.400	0.400	0.400
RI = 3	0.850	0.958	0.936	0.936	0.928	0.930
RI = 4	0.852	0.996	0.988	0.988	0.982	0.978
RI = 5	0.698	0.992	0.976	0.964	0.964	0.958
RI = 6	0.768	0.988	0.976	0.964	0.956	0.948
RI = 7	0.834	0.994	0.992	0.970	0.964	0.954
RI = 8	0.824	1.000	1.000	0.996	0.974	0.964
RI = 9	0.852	0.996	0.994	0.992	0.978	0.964
RI = 10	0.818	0.996	0.996	0.990	0.988	0.976

Figure 4.3 : Accuracies of the proposed classifier and those of k-nn. (RI feature selection strategy).

Figure 4.4 shows the achieved classification accuracies (using the RB strategy) of the proposed classifier, as well as the *k*-nn classifier for *k*=1, 5, 10, 15, and 20.

4.2.2 Discussion

By taking the most frequent bigrams as the most relevant ones (i.e., as features), the proposed approach attained an accuracy of 95% using as family signature a 17-tuple of real numbers. The signatures that enabled us to obtain this high level of accuracy are:

W32.Simile's Engine signature = (0.190, 0.030, 0.155, 0.048, 0.043, 0.057, 0.063,0.020, 0.076, 0.022, 0.0, 0.041, 0.109, 0.0, 0.122, 0.022, 0.0)

W32.Evol's Engine signature = (0.074, 0.026, 0.006, 0.326, 0.208, 0.014, 0.024, 0.073, 0.043, 0.048, 0.0, 0.071, 0.042, 0.0, 0.026, 0.019, 0.0)

W32.VCL's Engine signature = (0.111, 0.238, 0.142, 0.027, 0.076, 0.063, 0.063,0.033, 0.009, 0.018, 0.018, 0.054, 0.042, 0.0, 0.040, 0.052, 0.013)

W32.NGVCK's Engine signature = (0.132, 0.113, 0.106, 0.048, 0.203, 0.018, 0.055, 0.038, 0.022, 0.017, 0.070, 0.122, 0.007, 0.0, 0.007, 0.020, 0.017)

Benign's "Engine signature" = (0.165, 0.173, 0.091, 0.061, 0.052, 0.060, 0.052, 0.046, 0.060, 0.028, 0.019, 0.043, 0.024, 0.029, 0.02, 0.031, 0.029)

	FS	**1-nn**	**5-nn**	**10-nn**	**15-nn**	**20-nn**
RB = 3	0.650	0.846	0.844	0.822	0.830	0.824
RB = 4	0.884	0.968	0.940	0.924	0.926	0.922
RB = 5	0.940	0.990	0.978	0.978	0.974	0.964
RB = 6	0.948	0.988	0.988	0.974	0.964	0.962
RB = 7	0.946	0.988	0.986	0.972	0.970	0.962
RB = 8	0.948	0.984	0.982	0.972	0.958	0.956
RB = 9	0.932	0.988	0.986	0.980	0.968	0.966
RB = 10	0.940	0.990	0.980	0.978	0.970	0.966
RB = 11	0.882	0.996	0.976	0.960	0.954	0.940
RB = 12	0.910	0.996	0.980	0.972	0.956	0.944
RB = 13	0.926	0.998	0.984	0.976	0.968	0.962
RB = 14	0.948	1.000	0.984	0.968	0.954	0.952
RB = 15	0.936	1.000	0.988	0.972	0.958	0.948
RB = 16	0.938	1.000	0.986	0.972	0.962	0.958
RB = 17	0.950	1.000	0.992	0.980	0.966	0.966
RB = 18	0.926	1.000	0.994	0.978	0.964	0.964
RB = 19	0.916	1.000	0.996	0.978	0.960	0.962
RB = 20	0.940	1.000	0.988	0.962	0.954	0.958

Figure 4.4: Accuracies of the proposed classifier and those of k-nn. (RB feature selection strategy).

These signatures are small compared to those that were generated using as features the bigrams composed of the most relevant instructions available across the samples. The *1*-nn classifier was 100% accurate for RB= 14 to 20.

The overall performance of the proposed method can be compared to that of k-nn by examining the following measures:

- **Accuracy.** The proposed method achieved a 95% accuracy for RB=17, which is a high level of accuracy. *1*-nn and *5*-nn achieved 100% accuracy for RB =14 to 20, and for RI =5, respectively.
- **Time efficiency.** The proposed method is, in general, more time efficient than *k*-nn: the proposed method creates a family signature once and uses it to recognize new or never-before-seen programs that belong to that family. *k*-nn must visit all of the training instances each time it has to attribute a suspect program to a known engine by finding the suspect's nearest neighbors.
- **Space efficiency.** The proposed method is more space efficient than *k*-nn. The proposed method stores only one family signature, as an array of doubles, to represent all of the training instances. *k*-nn has to load all of the training instance, perhaps one at a time, in order to classify a suspect program.

Chapter 5
Conclusions and Directions for Further Work

5.1 Research Outcomes

We proposed a fast filter for engine-generated malware instances using *n*-gram Instruction Frequency Vector (NFV) as a feature vector to attribute a malware instance to the engine which generated it. The filtering process is optimized by computing the frequencies of only the "most relevant" *n*-grams, for some measure of relevance (e.g., most frequent *n*-grams).

Our first filtering method used the *1*-gram frequency vectors of malware instances, to attribute the instances to the engine that generated them. The approach was successful in discriminating variants of the `W32.Simile` metamorphic malware from benign programs.

Our second filtering method used the *2*-gram frequency vectors of engine-generated malware instances, to attribute each instance to the engine which generated it. This approach was successful in filtering malware generated by `W32.Simile`'s engine, `W32.Evol`'s engine, `W32.NGVCK`, and `W32.VCL`. Our results indicated that a small engine signature can be created using only the most frequent instructions or the most frequent bigrams across all the instances that the detector has on hand. A signature of only 17 doubles gave us an accuracy of 95%. The feature selection strategy was also shown to be important, since the RB strategy discussed in Chapter 4, is more space efficient than the RI strategy.

The proposed approaches have been successfully used in attributing natural language texts to their human authors [18]. By analogy to the fact that human authors tend to have distinct writing patterns, the engines generating the malware can also have distinct features that can be computed from the malware.

5.2 Directions for Future Work

In the future, we will expand our study to improve the feature selection strategies and experiment with different classifiers. We will experiment with larger numbers of malware instances and different families of malware. We will also do a more extensive examination of the existing body of knowledge in forensic linguistics to see if more methods from this field could be applied to malware detection. We will also see if byte *n*-gram frequency vectors would give us a level of accuracy that is at least as high as the opcode NFV used in the experiments. With byte NFVs the detector would not even be required to disassemble suspect programs.

Bibliography

[1] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan. "N-gram-based detection of new malicious code". Computer Software and Applications Conference, Annual International, Vol 2, pages 41–42, 2004.

[2] J. D. Aycock. "Computer Viruses and Malware". 2005.

[3] D. Bruschi, L. Martignoni, and M. Monga. "Using code normalization for fighting self mutating malware". In Proceedings of International Symposium on Secure Software Engineering,IEEE, 2006.

[4] E. T. center. http://www.eset.com/. ASET, 2008.

[5] M. R. Chouchane. "Approximate Detection of Machine-morphed Variants of Malicious Programs". PhD thesis, University of Louisiana at Lafayette, 2008.

[6] M. R. Chouchane and A. Lakhotia. "Using engine signature to detect metamorphic malware". Proceedings of the 4th ACM workshop on Recurring malcode, pages 73–78, 2006.

[7] M. R. Chouchane, A. Walenstein, and A. Lakhotia. "Using markov chains to filter machine-morphed variants of malicious programs". 3rd International Conference on Malicious and Unwanted Software, 2008, 2008.

[8] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. Bryant. "Semantics-aware malware detection". Symposium on Security and Privacy,IEEE, pages 32–46, 2005.

[9] F. Cohen. "Computational aspects of computer viruses". Computers and Security, 1989.

[10] T. Cymru. http://www.teamcymru org/. Team Cymru, 2008.

[11] L. M Danilo Bruschi and M. Monga. "Detecting self-mutating malware using control flow graph matching". Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA),IEEE, 2006.

[12] C Download.com. http://download.cnet.com/windows/, November 2008.

[13] F-Secure. http://www.fsecure.com. FSecure, 2008.

[14] http://usa.kaspersky.com/threats/. Kaspersky lab: Threats. October 2009.

[15] http://vx.netlux.org/. Vx heavens: Collections.

[16] http://vx.netlux.org/lib/vmd01.html. "How I made metaphor and what I've learnt".

[17] http://www.ollydbg.de/. Ollydbg v2.0.

[18] V. Keselj, F. Peng, N. Cercone, and C. Thomas. "N-gram-based author profiles for authorship attribution". Pacific Association for Computational Linguistics, 2003.

[19] R. Kohavi. "A study of cross-validation and bootstrap for accuracy estimation and model selection". International Joint Conference on Artificial Intelligence (IJCAI), 1995.

[20] J. Z. Kolter and M. A. Maloof. "Learning to detect and classify malicious executables in the wild". Journal of Machine Learning Research 7 (2006) 2721-2744, 2006.

[21] A. Lakhotia and M. Mohammed. "Imposing order on program statements to assist anti-virus scanners". Proceedings of the 11th Working Conference on Reverse Engineering, 2004.

[22] I. McAfee. "A Brief History of Malware: An Educational Note for Service Providers". 2005.

[23] S. J. Mihai Christodorescu. "Static analysis of executables to detect malicious patterns". Proceedings of the 12th USENIX Security Symposium, pages 169–186, 2003.

[24] F. Perriot, P. Ször, and P. Ferrie. Striking similarites: "Win32/simile and metamorphic virus code". Symantec Security Response, 2003.

[25] M. PressPass. Microsoft security intelligence report(sir) vol 6. Microsoft Security Intelligence Report, pages 1–15, 2009.

[26] D. S. R. Qinghua Zhang. Metaaware: "Identifying metamorphic malware". Computer Security Applications Conference, 2007. ACSAC 2007, pages 411–420, 2007.

[27] M. R.Chouchane, A. Walenstein, and A. Lakhotia. "Statistical signatures for fast filtering of instruction-substituting metamorphic malware". Proceedings of the 2007 ACM workshop on Recurring malcode, pages 31–37, 2007.

[28] SourceForge. http://sourceforge.net/, November 2008.

[29] P. Ször. "The Art of Computer Virus Research and Defense". 2005.

[30] P. Ször and P. Ferrie. "Hunting for metamorphic". Proceedings of the 2001 Virus Bulletin Conference, pages 123–144, 2001.

[31] A. Test. http://www.av-test.org/. AV Test Lab, 2008.

[32] A. Walenstein, R. Mathur, M. R. Chouchane, and A. Lakhotia. "Normalizing metamorphic malware using term rewriting". International Workshop on Source Code Analysis and Manipulation, IEEE, 2006.

[33] A. Walenstein, R. Mathur, M. R. Chouchane, and A. Lakhotia. "The design space of metamorphic malware". 2nd International Conference on I-Warfare and Security (ICIW), 2007.

[34] W. Wong and M. Stamp. "Hunting for metamorphic engines". Springer Journal in Computer Virology, pages 211--229, 2006.

[35] D. Z. H. Z. Yang Song, Jian Huang and C. L. Giles. "*k*-nn: Informative *k*-nearest neighbor pattern classification". Proceedings of the 11th European conference on Principles and Practice of Knowledge Discovery in Databases, 2007.