A Framework for Metamorphic Malware Analysis and Real-Time Detection

by

Shahid Alam
BSc., University of Engineering and Technology Lahore
MSc., Wayne State University
MASc., Carleton University

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science
University of Victoria

© Shahid Alam, 2014
University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

A Framework for Metamorphic Malware Analysis and Real-Time Detection

by

Shahid Alam
BSc., University of Engineering and Technology Lahore
MSc., Wayne State University
MASc., Carleton University

Supervisory Committee

Dr. Robert Nigel Horspool, Supervisor
(Department of Computer Science, University of Victoria)

Dr. Issa Traore, Co-Supervisor
(Department of Electrical and Computer Engineering, University of Victoria)

Dr. Ibrahim Sogukpinar, Outside Member
(Department of Computer Engineering, Gebze Institute of Technology)

Dr. Yvonne Coady, Department Member
(Department of Computer Science, University of Victoria)

ABSTRACT

Metamorphism is a technique that mutates the binary code using different obfuscations. It is difficult to write a new metamorphic malware and in general malware writers reuse old malware. To hide detection the malware writers change the obfuscations (syntax) more than the behavior (semantic) of such a new malware. On this assumption and motivation, this thesis presents a new framework named *MARD* for Metamorphic Malware Analysis and Real-Time Detection. We also introduce a new intermediate language named MAIL (Malware Analysis Intermediate Language). Each MAIL statement is assigned a pattern that can be used to annotate a control flow graph for pattern matching to analyse and detect metamorphic malware. MARD uses MAIL to achieve platform independence, automation and optimizations for metamorphic malware analysis and detection. As part of the new framework, to build a behavioral signature and detect metamorphic malware in real-time, we propose two novel techniques, named *ACFG* (Annotated Control Flow Graph) and *SWOD-CFWeight* (Sliding Window of Difference and Control Flow Weight). Unlike other techniques, *ACFG* provides a faster matching of CFGs, without compromising detection accuracy; it can handle malware with smaller CFGs, and contains more information and hence provides more accuracy than a CFG. *SWOD-CFWeight* mitigates and addresses key issues in current techniques, related to the change of the frequencies of opcodes, such as the use of different compilers, compiler optimizations, operating systems and obfuscations. The size of *SWOD* can change, which gives anti-malware tool developers the ability to select appropriate parameter values to further optimize malware detection. *CFWeight* captures the control flow semantics of a program to an extent that helps detect metamorphic malware in real-time. Experimental evaluation of the two proposed techniques, using an existing dataset, achieved detection rates in the range 94% – 99.6% and false positive rates in the range 0.93% – 12.44%. Compared to *ACFG*, *SWOD-CFWeight* significantly improves the detection time, and is suitable to be used where the time for malware detection is more important as in real-time (practical) anti-malware applications.

Contents

Supervisory Committee **ii**

Abstract **iii**

Table of Contents **iv**

List of Tables **vii**

List of Figures **viii**

Acknowledgements **x**

Dedication **xi**

1 Introduction and Motivation **1**

1.1 Malware 1

1.2 Hidden Malware 2

1.3 Obfuscations 3

1.3.1 Opcode Level 3

1.3.2 Control Flow Level 4

1.3.3 Self-Modifying Code 5

1.4 Real-Time Detection 7

1.5 Problem Statement 7

1.6 Contributions 8

1.6.1 Obtained Performance Improvements 10

1.7 Organization of the Thesis 10

2 Literature Review **12**

2.1 Metamorphic Malware Detection Systems 12

2.1.1 Control Flow Analysis 12

2.1.2 Information Flow Analysis 14
2.1.3 Opcode-Based Analysis 16
2.1.4 Summary 20
2.2 Intermediate Languages 22
2.2.1 Why a New Language for Malware Analysis? 24

3 MAIL (Malware Analysis Intermediate Language) 27
3.1 Why an Intermediate Language for Malware Analysis? 28
3.2 Binary Analysis for Malware Detection 31
3.2.1 More Examples of Obfuscation 33
3.3 Design of MAIL 36
3.3.1 MAIL Statements 38
3.3.2 MAIL Library 40
3.3.3 MAIL Patterns for Annotation 40
3.4 Conclusion 42

4 MARD (Metamorphic Malware Analysis and Real-Time Detection) 43
4.1 Model 43
4.2 Design 44
4.3 Characteristics 45
4.4 Components of MARD 47
4.5 Conclusion 48

5 ACFG (Annotated Control Flow Graph) 49
5.1 Definitions 49
5.2 ACFG For Metamorphic Malware Detection 50
5.2.1 Subgraph Matching 51
5.2.2 Pattern Matching 52
5.3 Runtime Optimization with Parallelization 53
5.4 Runtime Optimization with ACFG Reduction 56
5.5 Summary 58

6 SWOD-CFWeight (Sliding Window of Difference and Control Flow Weight) 60
6.1 Motivations and Overview 60

6.2 Statistical Analysis of MAIL Pattern Distributions for Metamorphic Malware . . . 63
6.2.1 Dataset . . . 63
6.2.2 MAIL Pattern Distributions . . . 64
6.3 Metamorphic Malware Detection Model . . . 68
6.3.1 Sliding Windows of Difference . . . 68
6.3.2 Control Flow Weight and MAIL Program Signature . . . 73
6.3.3 Signature Matching and Malware Detection . . . 74
6.3.4 Complexity Analysis . . . 75
6.4 Summary . . . 76

7 Evaluation, Analysis and Comparison **78**
7.1 Performance Metrics . . . 78
7.2 Performance of *ACFG* . . . 79
7.2.1 Dataset Based on ACFGs . . . 79
7.2.2 Empirical Study . . . 80
7.2.3 Comparison with Others . . . 82
7.3 Performance of *SWOD-CFWeight* . . . 83
7.3.1 Empirical Study . . . 83
7.3.2 Performance Results of *SWOD-CFWeight* and Comparison with *ACFG* . . . 83
7.3.3 Comparison with Others . . . 87

8 Conclusion and Future Work **89**
8.1 Discussion . . . 89
8.1.1 Static Analysis . . . 89
8.1.2 Dynamic Analysis . . . 90
8.2 Summary of Contributions . . . 91
8.3 Future Work . . . 93

A MAIL Grammar **94**

B One of the Reports Generated by MARD **98**

Bibliography **105**

List of Tables

Table 2.1 Summary of The metamorphic malware analysis and detection systems discussed in Section 2.1 21
Table 2.2 Summary of the intermediate languages developed for malware analysis and detection discussed in Section 2.2 and there comparison with MAIL 25

Table 5.1 Runtime improvement after parallelizing the *Subgraph Matching* component (using different number of threads) 55

Table 6.1 An example, comparing the change in frequency of Opcodes with the change in frequency of MAIL Pattern *ASSIGN*, of a Windows program *sort.exe* compiled with different level of optimizations. 62
Table 6.2 Dataset distribution based on the size of each program sample . 64
Table 6.3 *Class* distribution of the 1020 metamorphic malware samples . . 65

Table 7.1 Dataset distribution based on the number of Annotated Control Flow Graphs (ACFGs) for each program sample 80
Table 7.2 Dataset distribution based on the size (number of nodes) for each Annotated Control Flow Graph (ACFG) after normalization and shrinking 81
Table 7.3 Malware detection results for smaller dataset. 82
Table 7.4 Malware detection results for larger dataset. 82
Table 7.5 Summary and comparison with ACFG of the metamorphic malware analysis and detection systems discussed in Chapter 2 . . . 84
Table 7.6 Malware detection results for *SWOD-CFWeight* and comparison with *ACFG* 85
Table 7.7 Comparison of *SWOD-CFWeight* with the malware detection techniques discussed in Chapter 2 88

List of Figures

Figure 3.1 The CFG and the Source Code in C++ of the Function in Listing 3.1 34
(a) The CFG 34
(b) The Source Code 34

Figure 4.1 High Level Overview of MARD 45

Figure 5.1 An example of subgraph matching. The graph in Figure (a) is matched as a subgraph of the graph in Figure (b). 52
(a) A malware sample 52
(b) The malware embedded inside a benign program 52
Figure 5.2 Example of *pattern matching* of two *isomorphic* ACFGs. The ACFG in (a) is *isomorphic* to the subgraph (blocks 0 - 3) of the ACFG in (b). 53
Figure 5.3 Example of ACFG shrinking. ACFG X is not shrinkable. ACFG Y with 6 blocks is shrinked to ACFG Z with 4 blocks. 56
(a) ACFG X 56
(b) ACFG Y 56
(c) ACFG Z 56
Figure 5.4 Example of an ACFG, of one of the functions of one of the samples of the MWOR class of malware, before and after shrinking. The ACFG has been reduced from 92 nodes to 47 nodes. 57
(a) ACFG X 57
(b) ACFG Y 57
Figure 5.5 Example of an ACFG, of one of the functions of one of the samples of the MWOR class of malware, before and after shrinking. The ACFG has been reduced from 484 nodes to 145 nodes. . . 58
(a) ACFG X 58
(b) ACFG Y 58

Figure 5.6 Example of an ACFG, of one of the functions of the Windows disk free space utility program *df.exe*, before and after shrinking. The ACFG has been reduced from 894 nodes to 283 nodes. . . 59
(a) ACFG X 59
(b) ACFG Y 59

Figure 6.1 MAIL Patterns distributions based on the percentage of the MAIL Patterns in each sample in the dataset 66
(a) MAIL Pattern distributions for benign samples 66
(b) MAIL Pattern distributions for malware samples 66
Figure 6.2 Superimposing three of the MAIL Patterns distributions from Figures 6.1(a) and 6.1(b). 67
Figure 6.3 Sliding Window of Difference ($SWOD_{j1}$) as defined in *Definition 10.* $HWOD_{j1} = \{V_{j1}, V_{j2}, V_{j3}, \ldots V_{jn}\}$, where V_{j1}, V_{j2}, V_{j3}, . . . V_{jn}, are the *VWODs*. 70
Figure 6.4 Sliding Window of Differences (*SWODs*) for the MAIL Pattern *ASSIGN*. 71
Figure 6.5 Malware detection using MAIL program signatures. 75

ACKNOWLEDGEMENTS

I would like to thank Dr. Issa, Dr. Ibrahim and Dr. Nigel for their contributions throughout the course of my PhD studies in the form of contents, resources, commitment and support. I would especially like to mention the support and encouragement provided by Dr. Nigel that allowed me to pursue and develop my own ideas, without which it would have been impossible to finish my PhD. Consistent hard work and thorough discussions with Dr. Issa greatly helped me to gain knowledge and further insight in this field of research and in turn improved the dissertation. He helped me to keep focused, which was very difficult to maintain when there were so many diverting paths to delve into. Profound feedback and comments from Dr. Ibrahim provided practical directions, and his discerning knowledge of the subject helped to further polish and fine tune my ideas. I would also like to thank Dr. Yvonne Coady, department of Computer Science and the external examiner Dr. Habib Hamam, University of Moncton, for making my dissertation complete.

Numerous other people deserve to be mentioned for their advice and support during my PhD studies. I was inspired and developed a passion for teaching while working with Dr. LillAnne Jackson, Bette Bultena, Bill Gorman, Victoria Li and other fellow teaching assistants. Wendy Beggs and other staff members of the department of Computer Science office were always there to help and answer any question that I had about my studies, courses, university and the department.

Special thanks go to my family, my parents, wife and twins for their understanding, support and help, which enabled me to accomplish this.

Shahid Alam, Victoria, BC, Canada

DEDICATION

To my late father Khan Alam,
mother Naseem Akhtar,
wife Aminah Shahid,
and twins
Shayaan Alam
and
Samrah Shahid.

Chapter 1

Introduction and Motivation

End point security is often the last defense against a security threat. An end point can be a desktop, a server, a laptop, a kiosk or a mobile device that connects to a network (Internet). Recent statistics by the International Telecommunications Union [50] show that the number of Internet users (i.e. people connecting to the Internet using these end points) in the world have increased from 20% in 2006 to 40% (almost 2.7 billion in total) in 2013. A study carried out by Symantec on the impacts of cybercrime reports that worldwide losses due to malware attacks and phishing between July 2011 and July 2012 were $110 billion [88]. According to the 2011 Symantec Internet security threat report [89], there was an 81% increase in malware attacks over 2010, corresponding to 403 million new malware infections created, a 41% increase over 2010. In 2012 there was a 42% increase in the malware attacks over 2011. Web-based attacks increased by 30% in 2012. With these increases and anticipated future increases, such end points pose a new security challenge [76]. The onus is on security professionals and researchers in industry and in academia to devise new methods and techniques for malware detection and protection.

1.1 Malware

A broad definition of malware, also called malicious code, is used in the literature that includes viruses, worms, spywares and trojans. Here we use one of the earliest definitions by Gary McGraw and Greg Morrisett [65]: *Malicious code is any code added, changed, or removed from a software system in order to intentionally cause harm or subvert the intended function of the system.* A malware carries out activities

such as setting up a back door for a bot, setting up a keyboard logger and stealing personal information etc.

Antimalware software detects and neutralizes the effects of a malware. There are two basic detection techniques [49]: anomaly-based and signature-based.

1. *Anomaly-based detection* technique uses the knowledge of the behavior of a normal program to decide if the program under inspection is malicious or not.

2. *Signature-based detection* technique uses the characteristics of a malicious program to decide if the program under inspection is malicious or not.

Each of the techniques can be performed statically (before the program executes), dynamically (during or after the program execution) or both statically and dynamically (hybrid).

Detecting whether a given program is a malware is an undecidable problem [22, 62]. Antimalware software detection techniques are limited by this theoretical result. Malware writers exploit this limitation to avoid detection.

In the early days, the malware writers were hobbyists but now the professionals have become part of this group because of the incentives attached to it, such as financial gains, intelligence gathering, and cyber warfare etc. One of the basic techniques used by a malware writer is obfuscation [61]. Such a technique obscure a code to make it difficult to understand, analyze and detect malware embedded in the code.

1.2 Hidden Malware

Initial obfuscators were simple and were detected by simple signature-based detectors. To counter these detectors the obfuscation techniques have evolved in sophistication and diversity [11, 23, 56, 61, 70]. Such techniques obscure a code to make it difficult to understand, analyze and detect malware embedded in the code. These techniques can be divided into three groups [70]: packing, polymorphism and metamorphism.

Packing is a technique where a malware is packed (compressed) to avoid detection. Unpacking needs to be done before the malware can be detected. Current antimalware tools normally use entropy analysis [70] to detect packing but to unpack a program they must know the packing algorithm used to pack the program. Packing

is also used by legitimate software companies to distribute and deploy their software. Therefore a packed program needs to be unpacked before a malware can be detected.

Polymorphism is an encryption technique that mutates the static binary code to avoid detection. When an infected program executes the malware is decrypted and written to memory for execution. With each run of the infected program a new version of the malware is encrypted and stored for the next run. This results in a different malware signature with each new run of the program. The changed malware keeps the same functionality, i.e. the opcode is semantically the same for each instance. It is possible for a signature-based technique to detect this similarity of signatures at runtime.

Metamorphism is a technique that mutates the dynamic binary code to avoid detection. It changes the opcodes with each run of the infected program and does not use any encryption or decryption. The malware never keeps the same sequence of opcodes in memory. This is also called *dynamic code obfuscation*. There are two kinds of metamorphic malware defined in [70] based on the channel of communication used: *Closed-world malware*, that do not rely on external communication and can generate the newly mutated code using either a binary transformer or a metalanguage. *Open-world malware*, that can communicate with other sites on the Internet and update themselves with new features.

1.3 Obfuscations

This Section discusses some of the mutations used in polymorphic and metamorphic malware. We discuss some more obfuscations in Chapter 3 when we describe binary analysis for malware detection.

1.3.1 Opcode Level

Instruction reordering: By changing the ordering of instructions with commutative or associative operators, the structure of the instructions can be changed. This reordering does not change the behavior of the program. As a simple example:

```
a = 10; b = 20;                    a = 10; b = 20;
x = a * b;   can be changed to:   x = b * a
```

```
original machine code    and       assembly:

c7 45 f4 0a 00 00 00     movl   [rbp-0xc], 0xa  ; a = 10
c7 45 f8 14 00 00 00     movl   [rbp-0x8], 0x14 ; b = 20
8b 45 f4                 mov    eax, [rbp-0xc]  ;
0f af 45 f8              imul   eax, [rbp-0x8]  ; a * b
89 45 fc                 mov    [rbp-0x4], eax  ; x = a * b

changed machine code     and       assembly:

c7 45 f4 0a 00 00 00     movl   [rbp-0xc], 0xa  ; a = 10
c7 45 f8 14 00 00 00     movl   [rbp-0x8], 0x14 ; b = 20
8b 45 f8                 mov    eax, [rbp-0x8]  ; (reordered)
0f af 45 f4              imul   eax, [rbp-0xc]  ; b * a (reordered)
89 45 fc                 mov    [rbp-0x4], eax  ; x = b * a
```

Because of the two reordered instructions the original and the changed machine codes have different signatures. Other instructions can also be reordered if no dependency exists between the instructions.

Dead code insertion: Dead code is a code that either does not execute or has no effect on the results of a program. Following is an example of dead code insertion:

```
mov   ebx, [ebp+4]
add   ebx, 0x0           ; dead code
nop                      ; dead code
jmp   ebx
```

Register renaming: To avoid detection registers are reassigned in a fragment of a binary code. This changes the byte sequence (signature) of the machine code. A signature-based detector will not be able to match the signature if it is searching for a specific register. An example of register renaming is given below (register *eax* is renamed to *edx*):

```
lea   eax, [RIP+0x203768]          lea   edx, [RIP+0x203768]
add   eax, 0x10                    add   edx, 0x10
jmp   eax                          jmp   edx
```

1.3.2 Control Flow Level

Order of instructions: To change the control flow of a program the order of instructions is changed in the program, keeping the order of execution the same by

using jump instructions. An example of such a code is given in Section 1.3.3.

Branch functions: A branch function is used [61] to obscure the flow of control in a program. The target of all or some of the unconditional branches in a program is replaced by the address of a branch function. The branch function makes sure the branch is correctly transferred to the right target for each branch.

Opaque predicates: These are the predicates (variables) whose values are either true or false, such as $y^2 - 1 \neq x^2$ for any integer values of y and x, and still needs to be evaluated at runtime. To break the control flow of a program, an opaque predicate is used [23, 61] to create an unconditional branch that looks like a conditional branch.

Jump tables: Compilers use jump tables to implement switch-case statements in a language [79]. Jump tables are also used in system and function calls in operating systems. To alter the control flow of a program, either one or the combination of the following is used: an artificial jump table can be created, artificial jumps can be added to the existing jump table or the target of a jump in the table can be changed to point to a malicious code.

Exception tables: Modern compilers use exception tables to implement exceptions in high level languages for better performance [13, 31]. An exception table contains information about the various operations required for exception processing, such as invoking the destructors, adjusting the stack and finding the address of the exception handler. A malware writer can manipulate an exception table in a binary file to replace the address of an exception handler with the address of his/her own written malicious exception handler. A more ambitious malware writer can create a new exception table pointing to his/her own written malicious exception handler. This malicious exception handler may steal user information or open a back door for a botnet.

1.3.3 Self-Modifying Code

Self-modifying code is a code that changes its own instructions at runtime. The purpose of changing the instructions at runtime can be benign or malicious.

An optimizing program may change its instructions to improve its performance.

For example, to improve the runtime of a program, the numbers of instructions of parts of the program that run most (> 70%) of the time are reduced. To avoid branch prediction [73] and exploit instruction level parallelism (ILP) [73], a conditional branch is changed to an unconditional branch during the program execution. A program is compressed before execution to save space and reduce bandwidth required for downloading the program (when relevant), and then decompressed during the execution.

A malware may change its instructions at runtime to hide code to prevent reverse engineering or to evade detection by anti-malware programs. Self-modifying code is mostly used by polymorphic and metamorphic malware but is also used in other malware, for instance, to carry buffer overflow attacks [28].

The following example depicts a snippet of a self modifying original and obfuscated (order of instructions changed) code:

```
       original assembly        changed to        obfuscated assembly

       mov   ebx, 0x402364                               mov   ebx, 0x402364
       add   ebx, 0x100                                  jmp   j2
       push  edx                                  loop:  mov   edx, [ebx]
loop:  mov   edx, [ebx]                                  mov   [ecx], edx
       mov   [ecx], edx                                  jmp   j3
       dec   ebx                                  j1:    jmp   j4
       inc   ecx                                  j2:    add   ebx, 0x100
       cmp   ebx, (0x402364+0x100)                       push  edx
       jne   loop                                        jmp   loop
       pop   edx                                  j3:    dec   ebx
                                                         inc   ecx
                                                         jmp   j1
                                                  j4:    cmp   ebx, (0x402364+0x100)
                                                         jne   loop
                                                         pop   edx
```

The above code modifies its instructions by copying data (that contains code) from the data section to the code section of the program. This snippet of code can be part of a malware or a benign program.

1.4 Real-Time Detection

To provide continuous protection to an end point a security software needs to be operated and threats need to be detected in real-time. Antimalware provide protection from malware in two ways:

1. They can provide real-time protection by detecting the malware before the software is installed. All the incoming network traffic is monitored and scanned for malware. Depending on the methods used this continuous monitoring and scanning slows down a computer considerably, which is not practical and desirable. This is one of the main reasons this type of protection is not very popular.

2. They can provide protection by detecting a malware during or after the software installation. A user can scan different files and parts of the computer as and when he/she desires. This type of protection is much easier to use and is more popular.

In this thesis our emphasis is on the second option.

1.5 Problem Statement

As is clear from the above discussion out of the three malware groups mentioned above, metamorphic malware are getting more complex and pose a special threat and new challenges to the end point security. Stealthy mutation techniques provided by metamorphism helps a malware evade detection by today's signature-based anti-malware programs. Such malware are very difficult to analyse and detect manually even with the help of tools.

The number of new malware are increasing significantly and we need to automate the process of malware analysis and detection. To address effectively the challenges posed by metamorphic malware, we need to develop new methods and techniques to analyze the behavior of a program and make a better detection decision with few false positives.

Current techniques [14, 36, 37, 38, 43, 58, 59, 75, 78, 86, 93, 94, 97] for detecting malware are compute intensive, have poor detection rates, cannot handle smaller size malware, and are not suitable for real-time detection.

Some of the recent techniques that use opcodes, such as [75, 78, 93], have the potential to be used for real-time metamorphic malware detection, but have the following issues. The frequencies of opcodes can change by using different compilers, compiler optimizations and operating systems. Obfuscations introduced by polymorphic and metamorphic malware can change the opcode distributions. Selecting too many features (patterns) results in a high detection rate but also increases the runtime.

It is difficult to write a new metamorphic malware [90] and in general malware writers reuse old malware. To hide detection the malware writers change the obfuscations (syntax) more than the behavior (semantic) of such a new metamorphic malware. If an unknown metamorphic malware uses all or some of the same class of behaviors as are used by the training dataset (set of old metamorphic malware) then it is possible to detect these types of malware. On this assumption and motivation, we develop new techniques in this thesis to build behavioral signatures and detect effectively known and unknown metamorphic malware in real-time.

1.6 Contributions

Following are the contributions of this thesis:

1. We propose a new intermediate language named **MAIL** (Malware Analysis Intermediate Language) for malware analysis that can enhance the detection of metamorphic malware. Almost all the malware use binaries, instructions that a computer can interpret and execute, to infiltrate a computer system. There are hundreds of different instructions in any assembly language. We need to reduce and simplify these instructions considerably to optimize the static analysis of any such assembly program for malware detection.

    (a) MAIL provides an abstract representation of an assembly program and hence the ability for a tool to automate malware analysis and detection.

    (b) By translating binaries compiled for different platforms to MAIL, a tool can achieve platform independence.

    (c) Each MAIL statement is annotated with patterns that can be used by a tool to optimize malware analysis and detection.

2. We propose a novel technique named **ACFG** (Annotated Control Flow Graph) that reduces the effects of obfuscations and provides efficient metamorphic malware detection. ACFG is built by annotating CFG of a binary program and is used for graph and pattern matching to analyse and detect metamorphic malware in real-time. We also optimize the runtime of malware detection through parallelization and ACFG reduction, maintaining the same accuracy (without ACFG reduction) for malware detection. An ACFG:

   (a) Captures the control flow semantics of a program.

   (b) Provides a faster matching of ACFGs compared to other such techniques, without compromising the accuracy.

   (c) Can handle malware with smaller CFGs compared to other such techniques.

   (d) Contains more information and hence provides better accuracy than a CFG.

3. We propose a novel technique named **SWOD-CFWeight** (Sliding Window of Difference and Control Flow Weight) that reduces the effects of obfuscations and provides real-time metamrophic malware detection.

   (a) SWOD is a window that represents differences in MAIL Patterns[1] distributions (instead of opcodes) and hence makes the analysis independent of different compilers, compilers optimizations, instruction set architectures and operating systems. This is a significant improvement compared to existing techniques that use opcodes for malware detection.

   (b) Size of SWOD can change, this property gives a user (anti-malware tool developers) the ability to select appropriate parameters for a dataset to further optimize malware detection.

   (c) Unlike the current techniques that use opcodes for metamorphic malware detection, CFWeight captures the control flow semantics of a program and includes this information to an extent that helps detect metamorphic malware in real-time.

[1] Patterns present in MAIL are a high level representation of opcodes and can be used in a similar manner.

4. We present a new framework named **MARD** for Metamorphic Malware Analysis and Real-Time Detection. MARD uses MAIL and implements the above two proposed techniques, and provides:
   (a) Automation
   (b) Platform independence
   (c) Optimizations for real-time performance
   (d) Modularity
5. We conduct experimental **evaluation** of the proposed techniques, using an existing dataset of 5305 metamorphic malware and benign samples. We also provide distribution of the samples based on size of the files, number of ACFGs per sample and size (number of nodes) of ACFGs of the samples.

1.6.1 Obtained Performance Improvements

In the experimental evaluation carried out in Chapter 7, the two proposed techniques achieve detection rates in the range 94% – 99.6%. When comparing the two proposed techniques, *ACFG* achieves a detection rate (DR) of 94% and a false positive rate (FPR) of 3.1%, whereas *SWOD-CFWeight* improves over *ACFG*, and achieves a DR of 99.08% and a FPR of 0.93%. Compared to *ACFG*, *SWOD-CFWeight* significantly improves the detection time, and is more suitable to be used where the time for malware detection is important as in real-time (practical) anti-malware applications.

When compared with other such recent techniques, using the best reported results, the two proposed techniques show superior results, and unlike others are fully automatic, support malware detection for 64 bit Windows (PE binaries) and Linux (ELF binaries) platforms and have the potential to be used in a real-time detector.

1.7 Organization of the Thesis

The rest of the thesis is organized as follows:

Chpater 2 discusses the previous research efforts for detecting malware. We cover these research efforts under two categories: Metamorphic malware detection systems and Intermediate languages for malware analysis and detection.

Chapter 3 describes in detail the design and components of the new intermediate language MAIL and illustrates how a binary program can be translated to MAIL.

Chapter 4 describes the new proposed framework MARD in detail.

Chapter 5 describes the novel technique ACFG and how it is used for efficient metamorphic malware analysis and detection. The Chapter also discusses how parallelization and ACFG reduction reduces the runtime of a malware detector.

Chapter 6 defines and develop the novel technique SWOD-CFWeight and shows how it can be implemented in a malware detector for real-time metamorphic malware analysis and detection.

Chapter 7 describes the experimental studies in detail to analyse the correctness and the efficiency of our techniques proposed in this thesis. The Chapter discusses the experiments, that were carried out to evaluate the performance of the framework MARD.

Chapter 8 concludes the thesis and discusses the future work that can be carried out based on the research presented in this thesis.

Chapter 2

Literature Review

This chapter discusses previous research into detecting malware. We cover these research efforts under two categories: (1) *Metamorphic malware detection systems.* We cover only academic research efforts that claim to or will extend their detector to detect metamorphic malware. Our emphasis is on the most recent advances and their potential for malware detection. We therefore only cover some of the major research efforts, starting from the year 2012. (2) *Intermediate languages.* We cover only those intermediate languages that are used either in commercial or academic malware analysis and detection systems. We do not cover intermediate languages that are only used for binary analysis or reverse engineering.

2.1 Metamorphic Malware Detection Systems

We divide these metamorphic malware detection systems into three groups based on the type of analysis used for malware detection.

2.1.1 Control Flow Analysis

The method described in [86] uses model checking to detect metamorphic malware. Model checking techniques check if a given model meets a given specification. A program is modelled and the malware behavior is specified using a mathematical notation. The behavior of a program is checked without executing the program.

According to the paper [86], previous such techniques did not model the program stack. The paper [86] used a pushdown system to build a model that takes into account the behavior of the stack. They use IDA Pro [35] and Jackstab [53] to build

a CFG (control flow graph) [1] for a program binary. The CFG contains information about the contents of registers and memory at each control point of the program. It is translated into a pushdown system. The pushdown system stores the control points and the stack of the program.

Model checking is time consuming and sometimes it can run out of memory as was the case with an earlier approach [87] of the same authors. Times reported in the paper range from a few seconds (for 10 instructions) to over 250 seconds (for 10000 instructions). Real-life applications are much bigger than the samples tested. Therefore we believe their system cannot be used as a real-time malware detector.

The technique described in [59] checks similarities between code graphs (called semantic signatures in the paper) to detect metamorphic malware. A code graph is generated from the call graph of a program that is build from the binary of the program. It is not clear from the paper how the call graph is built (e.g. what tools, disassembler are used) from the binary. Only system calls are extracted from the binary to build the call graph. The problem of checking if two graphs are isomorphic is NP-complete [42]. To reduce the size of the call graph, they separated these system calls into 128 groups (32 objects x 4 behaviors). This reduced the processing time but also impacted the accuracy of the detector.

The code graph is compared with the already generated code graphs of the known metamorphic malware samples. Assuming that the new malware samples are the obfuscated versions of existing known malware, if a similarity is found then the code is classified as malicious code. However, the paper neither mentions the performance overheads of generating code graphs from the binaries nor the performance overheads of comparing the two code graphs.

The technique described in [38] uses an API call-gram to detect malware. An API call-gram captures the sequence in which API calls are made in a program. First, a call graph is generated from the disassembled instructions of a binary program. This call graph is converted to a call-gram. The call-gram becomes the input to a pattern matching engine. They use WEKA [47], which performs binary classification using a set of pattern recognition and machine learning algorithms. However, the paper does not mention the performance overheads of the system implemented. The system designed is not fully automated and cannot be used as a real-time detector.

The method presented in [36] and [37] uses a CFG for visualizing the control structure and representing the semantic aspects of a program. They extended the CFG with the extracted API calls to have more information about the executable program. This extended CFG is called the API-CFG.

Their system consists of three components: a PE-file disassembler, an API-CFG generator and a classification module. They built the API-CFG as follows. First they disassemble a PE file using a third party disassembler. Then the unnecessary instructions that are not required for building the CFG are removed from the disassembled instructions. The instructions kept are: jumps, procedure calls, API calls and targets of jump instructions.

A feature vector is generated using the API-CFG which is a sparse graph and can be represented by a sparse matrix. They store all the nonzero entries in a feature vector. An algorithm is given in the paper for converting the API-CFG to a feature vector.

Different classifiers (Decision Stump, Sequential Minimal Optimization, Naive Bayes, Random Tree, Lazy K-Star and Random Forest) are used to process the data consisting of the feature vectors of the PE files, and then decide if a PE file contains malware or not. This learning model based on the classifiers is used by a decision module to decide if a PE file is a malware or not.

The implemented system is dependent on a third party closed source disassembler. The disassembler cannot disassemble more than one file at a time, so they used a script to automate disassembly of a set of files. The proposed system is unsuitable for use as a real-time malware detector. Furthermore, the proposed techniques cannot be used to detect metamorphic malware, but as mentioned in the paper, this option will be explored in the future.

2.1.2 Information Flow Analysis

A recent effort [97] uses dynamic taint analysis (DTA) to automatically detect if an unknown sample exhibits malicious behavior or not. The proposed design consists of four engines: taint engine, test engine, malware detection engine and malware analysis engine. The taint engine tracks the flow of information (all actions taken by the system are kept as taint graphs) of the whole system. This information is used to detect malware from unknown samples by comparing extracted information against a set of defined policies. To perform manual detection, their malware analysis engine

can be used to help a human analyst examine the taint graphs in detail.

As a proof of concept, they implemented a system called *Panorama* as a plugin of an emulator. *Panorama* is not fully automatic so a tool was written in Python to load and install the samples. The tool was able to handle 70% of the samples. The remaining samples were installed manually. Using automatic and manual analysis together, the detection rate for these samples became 100%.

Panorama is part of an emulator and all the samples were run inside the emulator. Running a sample/application in an emulator to detect malware has its own overheads. The paper does not provide more detailed performance (timings) results and overheads. Panorama needs human analysts to inspect its data to detect malware more accurately. Since it runs in an emulator and takes a considerable amount of time for detection, it cannot be used as a real-time malware detector.

The technique described in [58] uses value set analysis (VSA) [6] to detect metamorphic malware. Value set analysis is a static analysis technique that keeps track of the propagation and changes of values throughout an executable. They track only register and stack values for efficiency reasons. This is how their system works: First they disassemble the executable. Then they apply the value set analysis to approximate the possible values of each memory location for every instruction in the program. These values are matched against a reference list of value sets, generated from infected files. Based on the matching results, a similarity score is computed and used to detect or classify the malware. They use static analysis so all execution paths are analysed. The disassembler used and the performance overheads are not described in the paper, so we cannot comment on the real-time applicability of their implemented system.

The technique described in [43] also uses VSA for detecting metamorphic malware. Their technique is based on extending the idea of VSA proposed in [58]. They track the register values for each API (application programming interface) call in a dynamic analysis setting. The use of dynamic analysis may miss some of the execution paths in a program during the analysis. Malware binaries are run and traced inside a controlled environment to collect register values. Based on the matching, a similarity score is computed which is used for detecting or classifying the malware. Because of the dependency on a controlled environment for execution, the proposed approach cannot be used as real-time detector.

The performance overheads of both the techniques proposed in [58] and [43] are not specified. [58] is based on static analysis and [43] is based on dynamic analysis, it would have been interesting to see the difference in the performances of these two techniques.

A framework presented in [7] for polymorphic worm detection is worth mentioning here, because it has the potential to be used for metamorphic malware detection. It is a graph based classification framework of content based polymorphic worm signatures. It relies on using byte-pattern-based signatures to detect worm traffic. A vertex in the graph is a common invariant string found in the majority of different forms of polymorphic worms, extracted from flow pools as described in [48], and an edge in the graph represents the directed sequences of two vertices. A vertex score is calculated which is a probability of vertex appearing in a suspicious flow pool as opposed to an innocuous flow pool. On the basis of this score, vertices are differentiated as strong or weak. Edges which consist of a weak vertex and a strong vertex or two strong vertices are considered strong. The signature set is defined as a conjunction of strong vertices and strong directed edges. This signature scheme is called CCM (Conjunction of Combinational Motifs). If one of these signatures matches completely with a network flow, then a malware artifact (a worm) has been detected. The experimental results reported in the paper [7] outperform two other byte-pattern-based techniques for polymorphic worm detection.

2.1.3 Opcode-Based Analysis

The method described in [80] uses opcode sequences as a representation of executables for malware detection. Each opcode is given a weight based on its frequency of occurrence in malware or in benign executables. The authors use the *Term Frequency* [52] and the calculated weight to compute the *Weighted Term Frequency* (WTF). The calculated weight is the relevance (occurrence of a opcode in malware or benign executable) of the opcode. Four different machine learning classifiers are trained and tested using WTF to detect malware, including Decision Tree, Support Vector Machines, K-Nearest Neighbours and Bayesian Networks.

Their best results based on the detection rate are obtained using the *Decision Tree* classifier which also achieves the best malware detection time. Most of the

execution time (from 0 – 45 seconds per sample file in the dataset, depending on the opcode-sequence length used) spent by their malware detector is on feature (calculated weight) extraction. The authors did not include this time when computing the testing time of the classifiers, whereas we have included this time when computing the testing time, as listed in Chapter 7. Therefore, we cannot compare their malware detection timings with the timings of the techniques proposed in this thesis. By not including the feature extraction time, the testing time for the *Decision Tree* classifier used in [80] is almost 0.

The technique presented in [82], similar to [80], also uses opcode sequences to represent executables for malware detection. After extracting the opcode sequences they compute the *Term Frequency* and *Inverse Document Frequency* [77] for each opcode sequence or feature in each file. After reducing the number of features by using the document frequency measure (number of files in which the feature appeared) they applied eight commonly used classification algorithms for malware detection.

The work presented in [96] provides a good introduction to malware generation and detection, and served as a benchmark for comparison in several other studies [8, 32, 60, 78, 93] on metamorphic malware. They analysed and quantified (using a similarity score) the degree of metamorphism produced by different metamorphic malware generators, and proposed a hidden Markov model (HMM) for metamorphic malware detection. A HMM is trained using the assembly opcode sequences of the metamorphic malware files. The trained HMM represents the statistical properties of the malware family, and is used to determine if a suspect file belongs to the same family of malware or not.

The malware generators analysed in [96] are G2, MPCGEN, NGVCK and VCL32. Based on the results, NGVCK (also used in this thesis) outperforms other generators. VCL32 and MPCGEN have very similar morphing ability, and the malware programs generated by G2 (also used in this thesis) have a higher average similarity than the other three. Based on these results, we can conclude that malware programs generated by NGVCK are the most difficult to detect out of the four.

The method described in [93] uses the chi-squared (χ^2) test [95] to detect metamorphic malware. Their method is based on the observation that different compilers use different subsets of instructions, i.e. each compiler has its own subset of instructions

for generating code. The instructions that are common between the two compilers will appear with different frequencies. An estimator function can then estimate if a set of instructions is generated by a particular compiler. The same concept can be used to estimate whether sets of instructions were generated by a metamorphic malware generator.

Their estimator works as follows: First they generate a spectrum of an infected program. This spectrum contains information about the typical frequencies of the opcodes (instructions). These are the expected frequencies of the instructions in a particular metamorphic generator. To detect if a file contains a metamorphic malware artifact these expected frequencies are compared with the observed frequencies. A χ^2 statistical test as described in [39] is used to determine if there is a significant difference between the expected and the observed frequencies. If there is a significant difference then the file under test is considered to be benign. Their implementation uses IDA Pro [35], a closed source disassembler, to disassemble the executables and is not fully automatic.

The technique presented in [78] uses the similarity of executables based on opcode graphs for metamorphic malware detection. Opcodes are first extracted from the binary of a program. Then a weighted opcode graph is constructed. Each distinct opcode becomes a node in the graph, and each outgoing edge leads to the node for a successor opcode. Each edge is given a weight representing the frequency that control transfers to the successor opcode. This graph is directly compared, using matrices, with the graph of known malware. This comparison is based on a scoring function developed in the paper. If the similarity score of the comparison is below the threshold then malware is detected otherwise the program is considered to be benign. The threshold is computed using the scoring function based on the scoring differences between different kinds of files (benign, normal and metamorphic virus files).

The method described in [75] uses a histogram of instruction opcode frequencies to detect metamorphic malware. A histogram is built for each file and is compared against the already built histograms of malware samples to classify the file as malware or benign. The similarity between two histograms is measured using a distance metric called Minkowski-form distance [55]. The system implemented extracts opcodes from a binary file and uses MATLAB to generate a histogram of

these opcodes, and is not fully automatic.

The technique presented in [5] used hidden Markov models (HMMs) to capture how hand written assembly differs from compiled code and how benign code differs from malware code. This model is used to detect malware. HMMs are built for both benign and malware programs. For each program, the probability of observing the sequence of opcodes is determined for each of the HMMs. If the HMM reporting the highest probability is malware, the program is flagged as malware.

The technique presented in [83] presented an opcode-based similarity measure inspired by substitution cipher cryptanalysis [51] to detect metamorphic malware. They obtained promising results. A score is computed using an analog of Jackobsens algorithm [51] that measures the distance between the opcode sequence of a given program and the opcode statistics for a malware program. A small distance suggests that malware has been detected.

The method described in [94] uses bioinformatics sequence alignment methods to detect metamorphic malware. The basic idea used in the paper is to extract the structural and functional characteristics of a program from the machine opcodes. They are aligned into multiple sequences for comparison and detection. The authors assume that in metamorphic malware some of the machine opcode(s) are replaced by equivalent machine opcode(s) but a complete rewrite is impossible if the same functionality is being maintained.

First they disassemble a binary to extract the opcodes. These opcodes are then aligned using local, global and multiple sequence alignments. Three kinds of signatures, single, group and probabilistic, are generated from these alignments. These signatures are compared with the signatures of the already known malware. A higher similarity score means a malware is detected.

They conducted experiments using the three signatures mentioned above and obtained the following results. A single signature achieved a higher detection rate (91%) but a very high false positive rate (52%). A group signature achieved a low detection rate (72.2%) but a very low false positive rate (0.01%). A probabilistic signature achieved a low detection rate (71%) and a low false positive rate (7%). The paper does not provide any information about the performance overheads of the proposed system implemented in the paper. With such low accuracies, the prototype

system cannot be used as an effective real-time detector. Because of its low detection rate, we do not further compare this technique with the techniques proposed in this thesis.

Recently, [14] presented a technique that uses the frequencies of occurrence of instructions in the disassembled code to detect metamorphic malware. Their technique relies on the assumption that some instructions occur within the metamorphic malware many times. Based on this assumption they build an instruction occurrence matrix (IOM) for a program. The IOM associates each opcode with the number of instructions that use the opcode, but have at least 2 occurrences in the program. A χ^2 statistical test is used to select the opcodes. Different types of decision tree classifiers are used with the selected opcodes to distinguish malware from a benign program. The paper does not mention the (runtime) performance of the proposed technique.

There is nothing mentioned in the paper [14] on the testing data (specifically unknown data) used for validating the proposed technique. For example, how are known (training) and unknown (testing) datasets are distributed, to validate that the technique proposed can also detect unknown malware? Due to a lack of such testing described in the paper, we consider this technique to be incapable of detecting unknown malware, and we do not include this technique for further comparison with the techniques proposed in this thesis.

2.1.4 Summary

Table 2.1 gives a summary of all the malware detection systems discussed above. None of the prototype systems implemented can be used as a real-time detector. The systems that claim perfect detection rates do not validate such claims with large enough data sets. They need to perform experiments using more samples. Out of all the research efforts discussed above, **API-CFG**, **Call-Gram** and **VSA-2** show impressive results and have the potential to be used as real-time malware detectors. However, **API-CFG** does not yet support detection of metamorphic malware, **VSA-2** is using a controlled environment for detection, and **Call-Gram** is not fully automated and its performance overheads are not mentioned in the paper.

Table 2.1: Summary of The metamorphic malware analysis and detection systems discussed in Section 2.1

System	**Analysis Type**	**Detection Rate**	**False Positives**	**Data Set Size Benign/Malware**	**Real Time**	**Platform**
Model-Checking [86]	Static	100%	1%	8 / 200	✗	Win 32
Code-Graph [59]	Static	91%	0%	300 / 100	✗	Win 32
Call-Gram [38]	Static	98.4%	2.7%	3234 / 3256	✗	Win 32
API-CFG [36, 37]	Static	97.53%	1.97%	2140 / 2305	✗	Win 32
DTA [97]	Dynamic	100%	3%	56 / 42	✗	Win XP 64
VSA-1 [58]	Static	100%	0%	25 / 30	✗	Win 32
VSA-2 [43]	Dynamic	98%	2.9%	385 / 826	✗	Win XP 64
Opcode-HMM-Wong [96]	Static	~90%	~2%	40 / 200	✗	Win & Linux 32
Chi-Squared [93]	Static	~98%	~2%	40 / 200	✗	Win & Linux 32
Opcode-Graph [78]	Static	100%	1%	41 / 200	✗	Win 32
Histogram [75]	Static	100%	0%	40 / 60	✗	Win 32
Opcode-HMM-Austin [5]	Static	93.5%	0.5%	102 / 77	✗	Win & Linux 32
Opcode-SD [83]	Static	~98%	~0.5%	40 / 800	✗	Linux 32
Opcode-Seqs-Santos [80]	Static	96%	6%	1000 / 1000	✗	Win 32
Opcode-Seqs-Shabtai [82]	Static	~95%	~0.1%	20416 / 5677	✗	Win 32

Real-time here means the detection is fully automatic and finishes in a reasonable amount of time. The perfect results should be validated with more number of samples than tested in the paper. The values for *Opcode-Graph* are not directly mentioned in the paper. We compute these values by picking a threshold of 0.5 from the similarity score in the paper.

2.2 Intermediate Languages

This Section discusses the academic and the commercial research efforts in the development of intermediate languages for malware analysis and detection. We also discuss why we need a new intermediate language for malware analysis and detection, and compare MAIL (Malware Analysis Intermediate Language), the new intermediate language developed as part of the thesis and described in detail in Chapter 3, with these research efforts. First we present one of the commercial efforts and then move on to the academic efforts. The reasons for selecting these research efforts are: (1) Information about them is available publicly. (2) They are well described, i.e. at least part of the syntax and semantics is either described or defined mathematically. (3) They are currently being used in either academic or commercial malware analysis and detection tools.

REIL is an intermediate language that is being used in a commercial reverse engineering tool named *BinNavi* [34, 92]. Although REIL is not specifically designed for malware analysis, it is used in *BinNavi* for manual malware analysis and detection. In [81], Sepp et al. proposed an extension of REIL with relational information by translating the instruction's side effects via its flag setting actions into arithmetic instructions. The extension also helps reduce the size of a REIL program. The core language has a very reduced instruction set. It consists of only 17 different instructions and uses a flat memory model. The native instructions are translated to REIL instructions using a map. Based on the experiments carried out by the authors, on average one original native instruction is translated into approximately 20 REIL instructions. Unhandled native instructions are replaced with NOP instructions which may introduce inaccuracies in disassembling. There are no examples in the paper of translating an assembly program into REIL. Furthermore, REIL does not translate FPU, MMX and SSE instructions, nor any privileged instructions such as system calls, interrupts and other kernel-level instructions. The reason for not including these instructions is that the authors think that these instructions are not yet being used to exploit security vulnerabilities. REIL cannot translate instructions of the type that select registers with an index, as in the PowerPC. REIL cannot handle self-modifying code. The reason for this is that the REIL instructions cannot be overwritten or modified during interpretation of REIL code.

SAIL is an intermediate language presented in [20] that represents a CFG of the program under analysis, and is used in a prototype malware detection tool developed by the authors. Each instruction in SAIL is either an assignment statement or a call statement, and becomes a block [1] and a node in the CFG. The *operators* supported in SAIL are arithmetic, bit-vector, relational and the special memory addressing operator. A node in the CFG contains only a single SAIL instruction, which can make the number of nodes in the CFG extremely large and therefore can make analysis excessively slow for larger binary programs.

The VINE Intermediate Language (VINE-IL) proposed by Song et al. [85] is the intermediate language of the static analysis framework VINE used in the *BitBlaze* project. *BitBlaze* provides an extensible binary analysis platform for security applications. It is not specifically designed for malware detection but for general security applications. *BitBlaze* is used in the tool *Panorama* [97] for malware analysis and detection. The authors chose simplicity over efficiency, so VINE first translates a binary to VEX, an intermediate language used in Valgrind [68] (a dynamic binary instrumentation tool) and then to VINE-IL. The reason for not using VEX intermediate language directly, is the presence of implicit side effects in VEX instructions. In VINE-IL the final translated instructions have all the side effects explicitly exposed as VINE instructions. While exposing all the side effects in VINE-IL may be appropriate for general security applications such as program verification, this may not be efficient for specific security applications such as malware detection. Different platforms have different number and type of flags. Exposing all the side effects makes this approach general but also makes it difficult to maintain platform independence. In contrast to VINE-IL, side-effects are avoided in MAIL, making the language much simpler and providing the basis for efficient malware detection.

In [4], the authors use an intermediate language called CFGO-IL to simplify transformation of a program in the x86 assembly language to a CFG. After translating a binary program to CFGO-IL, the program is optimized to make its structure simpler. The optimizations also remove various malware obfuscations from the program. These optimizations include dead code elimination, removal of unreachable branches, constant folding and removal of fake conditional branches inserted by malware. Side effects of the assembly instructions are exposed explicitly in the instructions of the

CFGO-IL. The authors developed a prototype malware detection tool using CFGO-IL that takes advantage of the optimizations and the simplicity of the language. However, by exposing all the side effects of an instruction, the language faces the same problem of maintaining the platform independence as VINE-IL. Furthermore, the size of a CFGO-IL program tends to much larger than the original assembly program.

In [17], Cesare and Xiang introduce a new intermediate language for malware analysis named WIRE. The language is currently being used in the *Malwise* tool [19] developed by the authors. To the best of our knowledge, this is the only research effort that has the same goals as the MAIL language. The language is formally defined using an incomplete set of BNF notations. The authors defined the operational semantics of WIRE and provided manual examples to check the semantic equivalence of obfuscated code using these operational semantics. WIRE does not explicitly specify indirect jumps, making malware detection more complicated. There is only one instruction *ijmp* in WIRE that uses a register as the branch target. The register contents (address) can be known or unknown and hence can complicate the malware analysis, and may render an incorrect analysis. To simplify malware analysis in MAIL, this information is made explicit in the instruction.

Furthermore, the authors mention side effects of the assembly instructions as one of the difficulties of using the native assembly, but do not say anything about the side effects of the WIRE instructions. It is not clear how the language is used in the *Malwise* tool to automate the malware analysis and detection process. None of the referenced papers [15, 16, 17, 18, 19] covers the automation process using WIRE.

There are other such research efforts [12, 46, 98, 99] that also use an intermediate representation/language to simplify the static analysis of malware, and do not give much detail of the language itself, so we are not able to review or compare them here.

2.2.1 Why a New Language for Malware Analysis?

Table 2.2 gives a summary of all the intermediate languages discussed above. The machine model of all the intermediate languages is based on registers, because the majority of the platform architectures, such as Intel x86 and ARM, available today are register-based machines.

A mathematical (formal) model (definition) of a language can be used as a precise,

Table 2.2: Summary of the intermediate languages developed for malware analysis and detection discussed in Section 2.2 and there comparison with MAIL

Intermediate Language	**Machine Model**	**General Format**	**Side Effects**	**Tool Support**	**Well Defined**
REIL [34]	Register	Three Address Code	One Implicit	BinNavi	✗
SAIL [20]	Register	Open Form	None	Noname Tool	✗
VINE-IL [85]	Register	Open Form	All Explicit	Panorama	✗
CFGO-IL [4]	Register	Open Form	All Explicit	Noname Tool	✗
WIRE [17]	Register	Three Address Code	NA	Malwise	~✓
MAIL	Register	Open Form	None	MARD	✓

Well defined means that a mathematical model of the language is completely defined and is available publicly. Unlike *three address code* [1], which always contain three operands, *open form* is a combination of different formats and may contain one or more than one operands.

unambiguous and platform independent standard for the language. This definition can be used to implement the language for any platform. A well defined language helps us formally reason about the programs written in that language. Techniques such as model checking [21, 84] can be used on such a language to decide if two programs are similar or not, which is important for malware detection. MAIL has been developed as a well defined language with all definitions complete, unlike other such languages, and hence provides all the advantages as mentioned above.

Whenever a new language is introduced a question arises, *why not extend one of the existing languages?* Our answer to this question is as follows.

Extending an intermediate language without a complete formal model being defined may change the semantics of the language into something other than what the original author intended. In this case we may have to rewrite some or all of the tools for the extended intermediate language.

None of the previous research efforts on the existing intermediate languages ex-

plain in detail how assembly language instructions are translated to intermediate language statements. For example, how is an Intel x86 instruction *PREFETCH* (all other architectures also support some kind of *prefetch* instruction) transformed to intermediate language? To translate a set of instructions (e.g. there are 500+ different instructions in Intel x86-64 instruction set architecture [27]), we need to know what specific information should be included, or excluded from, the intermediate language to optimize malware analysis and detection. Without such information, it is non-trivial to extend a language for malware analysis and detection. We believe it requires more labour and time to get this information from the source code of the tools written for an existing language than to write tools for the new language.

The existing intermediate languages, as discussed above, have not shown the capability of automating malware analysis and detection. Because of the unavailability of a well defined formal model and detailed explanation, enhancing this capability of an existing language may require more work than designing a new language with this capability.

Based on the discussion above there is a need to develop a new intermediate language for malware analysis and detection. MAIL as an intermediate language takes a new step towards automating and optimizing malware analysis and detection.

Chapter 3

MAIL (Malware Analysis Intermediate Language)

Intermediate languages are used in compilers to translate the source code into a form that is easy to optimize and to provide portability. The term intermediate language also refers to the intermediate language used by the compilers of high level languages that do not produce any machine code, such as Java and C#. An example of adding two numbers in the intermediate language CIL (Common Intermediate Language) used in implementing C# is as follows:

```
a = a + b;
is translated to the following CIL code:
ldloc.0        ; Push the first local on the stack
ldloc.1        ; Push the second local on the stack
add            ; Pop the two locals, add them and push the result on the stack
stloc.0        ; Pop the result and store it in the first local
```

CIL is a stack-based language, i.e, the data is pushed on the stack instead of pulled from the registers. That is one of the reasons why, in the example above, one simple add statement is translated into four stack-based statements. The same add statement can be translated into the *three address code* [1] as:

```
a := a + b
```

The *three address code* format is an intermediate language used by most compilers in current use. The two popular open source compilers GCC [91] and LLVM [57] use *three address code* in their intermediate languages.

3.1 Why an Intermediate Language for Malware Analysis?

In Chapter 2, we have discussed and presented a critical review of other languages used for malware analysis and detection and why we need a new language. Here we are going to list some of the general reasons why we need to transform a program in an assembly language to an intermediate language.

1. There are typically hundreds of different instructions in an assembly language. For example the number of instructions in three ISAs (Instruction Set Architectures) are: 500+ for Intel x86-64 [27], 200+ for ARM [74] and 500+ for IBM PowerPC [26]. We need to reduce the number of these instructions considerably to speed up static analysis of an assembly program.

2. Not only are there many instructions, but they can contain much complexity. Examples include the Intel x86-64 instructions *PREFETCH*, *MOVD* and *MOVQ*. The instruction *PREFETCH* moves data from the memory to the cache. It is unclear whether this action is important if we are performing static analysis for malware detection. There are other instructions that can be ignored during malware analysis. Our intermediate language hides/ignores these instructions and makes the language more transparent to static analysis. The instructions *MOVD* and *MOVQ* copy a double word or a quad word, respectively, from the source operand to the destination operand. We do not take into account the size of the word being copied in our static analysis, and replace these kinds of instructions with a much simpler *ASSIGN* instruction. Using such techniques, an intermediate language allows us to use simpler instructions to make the static analysis much simpler.

3. We want a common intermediate language that can be used with different platforms, such as Intel x86-64 and ARM (the two most popular architectures in current computers), so that we do not have to perform a separate static analysis for each platform.

4. Assembly instructions can have multiple hidden side effects, such as effects on the flags, that can substantially increase the effort required for static analysis. In this case, there are three options for an intermediate language that make static analysis easier; either remove all side effects, or support only one side

effect, or explicitly define all side effects in the instruction. Because our focus is mainly on malware analysis, out of these three, in our opinion the first option is the best option.

5. An intermediate language can be easily translated into a string, a tree or a graph and hence can be optimized for various analyses that are required for malware analysis and detection, such as pattern matching and data mining.

6. To reduce the number of different instructions for static analysis, functionally equivalent assembly instructions can be grouped together in one intermediate language instruction, such as:

```
(xor eax, eax) | (add eax, 0) | (sub eax, eax) => eax = 0
(add ebx, 0x2000) & (add eax, ebx) | (lea eax, [ebx + 0x2000]) => eax = expr
```

   where expr = (ebx + 0x2000) and its value can be known or unknown depending on the value of ebx. This information should be explicitly defined in the language.

7. Unknown branch addresses in an assembly program make it difficult to build a correct CFG for the program. An intermediate language for malware analysis can take care of these branches. For example, for indirect jumps and calls (which are branches whose target is unknown or cannot easily be determined by static analysis) only a change in the source code can affect them, so it is safe to ignore these branches for malware analysis where the change is only carried out in the machine code. In the following paragraphs, we explain this in detail using an example from one of the PARSEC [9] benchmarks. Using the same example, we also highlight one of the major disadvantages of using dynamic analysis for malware detection, i.e. the inability to reach and analyse all the execution paths in a program.

The following example shows the function *Condition()* from one of the benchmarks in the PARSEC benchmark suite [9]. This function initializes a static condition variable of a thread. A local variable *rv* is used in a *switch* statement to jump to an appropriate exception generated by the *pthread_cond_init()* function. This function initializes the condition variable of a thread and returns zero if successful, otherwise it returns an error number. The value returned by the *pthread_cond_init()* function can only be determined at runtime, as is also the case for the value of *rv*.

```
The C++ source code with the translated (disassembled) assembly code:

Condition::Condition(Mutex &_M)
           throw(CondException)
{                                     471b50: push %rbp
   int rv;                            471b51: push %rbx
   M = $_M;                           471b52: sub $0x38,%rsp
   nWaiting = 0;                      471b52: sub $0x38,%rsp
   nWakeupTickets = 0;                471b56: mov %rsi,(%rdi)
   rv = pthread_cond_init(&c, NULL);  471b59: movl $0x0,0x8(%rdi)
                                      471b60: movl $0x0,0xc(%rdi)
                                      471b67: xor %esi,%esi
                                      471b69: add $0x10,%rdi
                                      471b6d: callq 404b60 <pthread_cond_init@plt>

   switch(rv) {   [rv UNKNOWN]        471b72: cmp $0x16,%eax
      case 0:                         471b75: jbe 471bb0 <Condition:Mutex>
         break;                       471b77: mov 0x21934a(%rip),%r8
      case EAGAIN:                    471b7e: mov $0x8,%edi
      case ENOMEM: {                  471b83: lea 0x10(%r8),%rbp
         CondResourceException e;     471b87: mov %rbp,(%rsp)
         throw e;                     471b8b: callq 404d00 <allocate_exception@plt>
         break;                       471b90: mov 0x219359(%rip),%rdx
      }                               471b97: mov 0x219342(%rip),%rsi
      case EBUSY:                     471b9e: mov %rax,%rdi
      case EINVAL: {                  471ba1: mov %rbp,(%rax)
         CondInitException e;         471ba4: callq 404da0 <cxa_throw@plt>
         throw e;                     471ba9: nopl 0x0(%rax)
         break;                       471bb0: lea 0x6995(%rip),%rcx <Exception>
      }                               471bb7: mov %eax,%ebx
      default: {                      471bb9: movslq (%rcx,%rbx,4),%rax
         CondUnknownException e;      471bbd: lea (%rax,%rcx,1),%rdx
         throw e;                     471bc1: jmpq *%rdx    [UNKNOWN BRANCH TARGET]
         break;                       471bc3: nopl 0x0(%rax,%rax,1)
      }                               471bc8: mov 0x219231(%rip),%rdi
   }                                  471bcf: lea 0x10(%rdi),%rbx
}                                     471bd3: mov $0x8,%edi
                                      471bd8: mov %rbx,0x10(%rsp)
```

Dynamic analysis can be used to determine the value of *rv*, but it is possible that such an analysis may not be able to trace all of the executable paths (such as one of the cases of the *switch* statements), e.g. when *rv* is always zero and is non-zero

only in rare cases. These rare cases may not get executed, or may be executed only after running the program for a very long time. In this latter case, analysis becomes impractical. A malware writer can exploit such a weakness and inject the malware code by changing the target address of any of the branches inside the *switch* statement to his/her own malicious code. In such a case the dynamic analysis may not be able to detect this malicious behavior.

To overcome this failing of dynamic analysis, we use static analysis to build a CFG that covers all the execution paths in a program, in this case that would be all the *switch* statements. The disassembled code above generates an unknown branch target address (tagged as [UNKNOWN BRANCH TARGET]). This address cannot easily be computed using static analysis. It is not possible for a malware writer to use this particular instruction in its current form for malicious code. He/she will have to change this instruction. For example, the register *rdx* can be replaced with the address of some malicious code, in which case the branch target address will become known. Therefore it is safe to ignore such branches.

3.2 Binary Analysis for Malware Detection

MAIL is based on binary analysis to optimize malware detection. This section provides some background on binary analysis for malware detection.

Almost all malware uses binaries (instructions that a computer can interpret and execute) to infiltrate a computer system. Binary analysis is the process of analysing the structure and behavior of a binary program either automatically, manually or both. There are several goals for such analysis. They include optimization, verification, profiling, performance tuning, reverse engineering and malware detection. In this thesis we perform binary analysis for malware detection. We further explain how binary analysis can help us understand a program and detect malware in the program by using a simple binary program (a function called *sort*) that is part of the class *Merge* in a sorting program. This function performs a merge sort on an array of integers. The source code of this function in C++ is shown in Figure 3.1 (b). The binary analysis of this function is listed below and explained in the following paragraphs.

The listing 3.1 is divided into two columns numbered I and II separated by a colon (:). There are a total of 104 assembly instructions in this function. This function is part of a binary program (in ELF x86-64 file) that is first disassembled and then the disassembled program is analysed and used to build CFGs for functions in the

program. In Listing 3.1 each instruction is assigned a block number and an address. Columns I and II are further divided into five subcolumns. Subcolumn 1 is the block number, subcolumn 2 is the address, subcolumn 3 is the machine code, subcolumns 4 and 5 are the assembly instructions in Intel syntax.

```
Listing 3.1 Binary Analysis of The Disassembled Function
Merge::sort(int key[], int size)
```

```
                    Column I                                              Column II

0  40108e             55  PUSH                  RBP       :  5  40113b        488b45c8   MOV        RAX, [RBP-0x38]
0  40108f         4889e5   MOV             RBP, RSP       :  5  40113f          4189f9   MOV               R9D, EDI
0  401092             53  PUSH                  RBX       :  5  401142          4189f0   MOV               R8D, ESI
0  401093       4883ec48   SUB            RSP, 0x48       :  5  401145          4889de   MOV               RSI, RBX
0  401097       48897dc8   MOV      [RBP-0x38], RDI       :  5  401148          4889c7   MOV               RDI, RAX
0  40109b       488975c0   MOV      [RBP-0x40], RSI       :  5  40114b      e8e2fdffff  CALL               0x400f32
0  40109f         8955bc   MOV      [RBP-0x44], EDX       :  5  401150          8b45e8   MOV        EAX, [RBP-0x18]
0  4010a2         8b45bc   MOV      EAX, [RBP-0x44]       :  5  401153            01c0   ADD               EAX, EAX
0  4010a5           4898  CDQE                            :  5  401155          0145ec   ADD        [RBP-0x14], EAX
0  4010a7       48c1e002   SHL             RAX, 0x2       :  6  401158          8b45e8   MOV        EAX, [RBP-0x18]
0  4010ab         4889c7   MOV             RDI, RAX       :  6  40115b          8b55bc   MOV        EDX, [RBP-0x44]
0  4010ae     e8e9f9ffff  CALL             0x400a9c       :  6  40115e            89d1   MOV               ECX, EDX
0  4010b3       488945d8   MOV      [RBP-0x28], RAX       :  6  401160            29c1   SUB               ECX, EAX
0  4010b7 c745e801000000   MOV DWORD [RBP-0x18], 0x1      :  6  401162            89c8   MOV               EAX, ECX
0  4010be     e9f2000000   JMP        0x4011b5 [11]       :  6  401164          3b45ec   CMP        EAX, [RBP-0x14]
1  4010c3 c745ec00000000   MOV DWORD [RBP-0x14], 0x0      :  6  401167          0f9fc0  SETG                     AL
1  4010ca     e989000000   JMP         0x401158 [6]       :  6  40116a            84c0  TEST                 AL, AL
2  4010cf         8b45e8   MOV      EAX, [RBP-0x18]       :  6  40116c    0f855dffffff   JNZ           0x4010cf [2]
2  4010d2         8b55ec   MOV      EDX, [RBP-0x14]       :  7  401172  c745ec00000000   MOV  DWORD [RBP-0x14], 0x0
2  4010d5         8d0402   LEA       EAX, [RDX+RAX]       :  7  401179      e923000000   JMP           0x4011a1 [9]
2  4010d8         0345e8   ADD      EAX, [RBP-0x18]       :  8  40117e          8b45ec   MOV        EAX, [RBP-0x14]
2  4010db         3b45bc   CMP      EAX, [RBP-0x44]       :  8  401181            4898  CDQE
2  4010de   0f8e11000000   JLE         0x4010f5 [4]       :  8  401183        48c1e002   SHL               RAX, 0x2
3  4010e4         8b45ec   MOV      EAX, [RBP-0x14]       :  8  401187        480345c0   ADD        RAX, [RBP-0x40]
3  4010e7         8b55bc   MOV      EDX, [RBP-0x44]       :  8  40118b          8b55ec   MOV        EDX, [RBP-0x14]
3  4010ea           89d1   MOV             ECX, EDX       :  8  40118e          4863d2 MOVSXD              RDX, EDX
3  4010ec           29c1   SUB             ECX, EAX       :  8  401191        48c1e202   SHL               RDX, 0x2
3  4010ee           89c8   MOV             EAX, ECX       :  8  401195        480355d8   ADD        RDX, [RBP-0x28]
3  4010f0         2b45e8   SUB      EAX, [RBP-0x18]       :  8  401199            8b12   MOV             EDX, [RDX]
3  4010f3           eb03   JMP         0x4010f8 [5]       :  8  40119b            8910   MOV             [RAX], EDX
4  4010f5         8b45e8   MOV      EAX, [RBP-0x18]       :  8  40119d        8345ec01   ADD  DWORD [RBP-0x14], 0x1
5  4010f8         8945e4   MOV      [RBP-0x1c], EAX       :  9  4011a1          8b45ec   MOV        EAX, [RBP-0x14]
5  4010fb         8b45ec   MOV      EAX, [RBP-0x14]       :  9  4011a4          3b45bc   CMP        EAX, [RBP-0x44]
5  4010fe           4898  CDQE                            :  9  4011a7          0f9cc0  SETL                     AL
5  401100       48c1e002   SHL             RAX, 0x2       :  9  4011aa            84c0  TEST                 AL, AL
5  401104         4889c1   MOV             RCX, RAX       :  9  4011ac    0f85ccffffff   JNZ           0x40117e [8]
5  401107       48034dd8   ADD      RCX, [RBP-0x28]       :  10 4011b2          d165e8   SHL  DWORD [RBP-0x18], 0x1
5  40110b         8b45ec   MOV      EAX, [RBP-0x14]       :  11 4011b5          8b45e8   MOV        EAX, [RBP-0x18]
5  40110e         4863d0 MOVSXD             RDX, EAX      :  11 4011b8          3b45bc   CMP        EAX, [RBP-0x44]
5  401111         8b45e8   MOV      EAX, [RBP-0x18]       :  11 4011bb          0f9cc0  SETL                     AL
5  401114           4898  CDQE                            :  11 4011be            84c0  TEST                 AL, AL
5  401116       488d0402   LEA       RAX, [RDX+RAX]       :  11 4011c0    0f85fdfeffff   JNZ           0x4010c3 [1]
5  40111a       48c1e002   SHL             RAX, 0x2       :  12 4011c6        488b45d8   MOV        RAX, [RBP-0x28]
5  40111e         4889c2   MOV             RDX, RAX       :  12 4011ca          4889c7   MOV               RDI, RAX
5  401121       480355c0   ADD      RDX, [RBP-0x40]       :  12 4011cd      e82af9ffff  CALL               0x400afc
5  401125         8b45ec   MOV      EAX, [RBP-0x14]       :  12 4011d2        4883c448   ADD              RSP, 0x48
5  401128           4898  CDQE                            :  12 4011d6              5b   POP                    RBX
5  40112a       48c1e002   SHL             RAX, 0x2       :  12 4011d7              c9 LEAVE
5  40112e         4889c3   MOV             RBX, RAX       :  12 4011d8    ff0502000000 INC_A             [RIP+0x02]
5  401131       48035dc0   ADD      RBX, [RBP-0x40]       :  12 4011de            eb04 JMP_A               0x4011e4
5  401135         8b7de4   MOV      EDI, [RBP-0x1c]       :  12 4011e0        00000000 CTR_A
5  401138         8b75e8   MOV      ESI, [RBP-0x18]       :  12 4011e4              c3   RET
```

The total number of blocks in this function is 12. The block sizes vary. For example, block number 4 has only 1 instruction whereas block number 5 has 30

instructions. A block is a basic block [1] which is a maximal sequence of instructions with the following properties: (1) It has only one entry point but can have more than one exit points. (2) An instruction with a branch to another block in the same function ends the block. (3) If an instruction is a target of another branch within the same function then that instruction starts a new block.

If an instruction branches to another block in the function listed above, the target instruction's block number is listed at the end enclosed in brackets. For example the last instruction in block 1 ends with 6, because this instruction is branching to the address 401158 which is the address of the first instruction of block 6. Based on the analysis information listed above we build a CFG of this function that is shown in Figure 3.1 (a). We are going to compare this CFG with the source code of this function in C++ shown in Figure 3.1 (b).

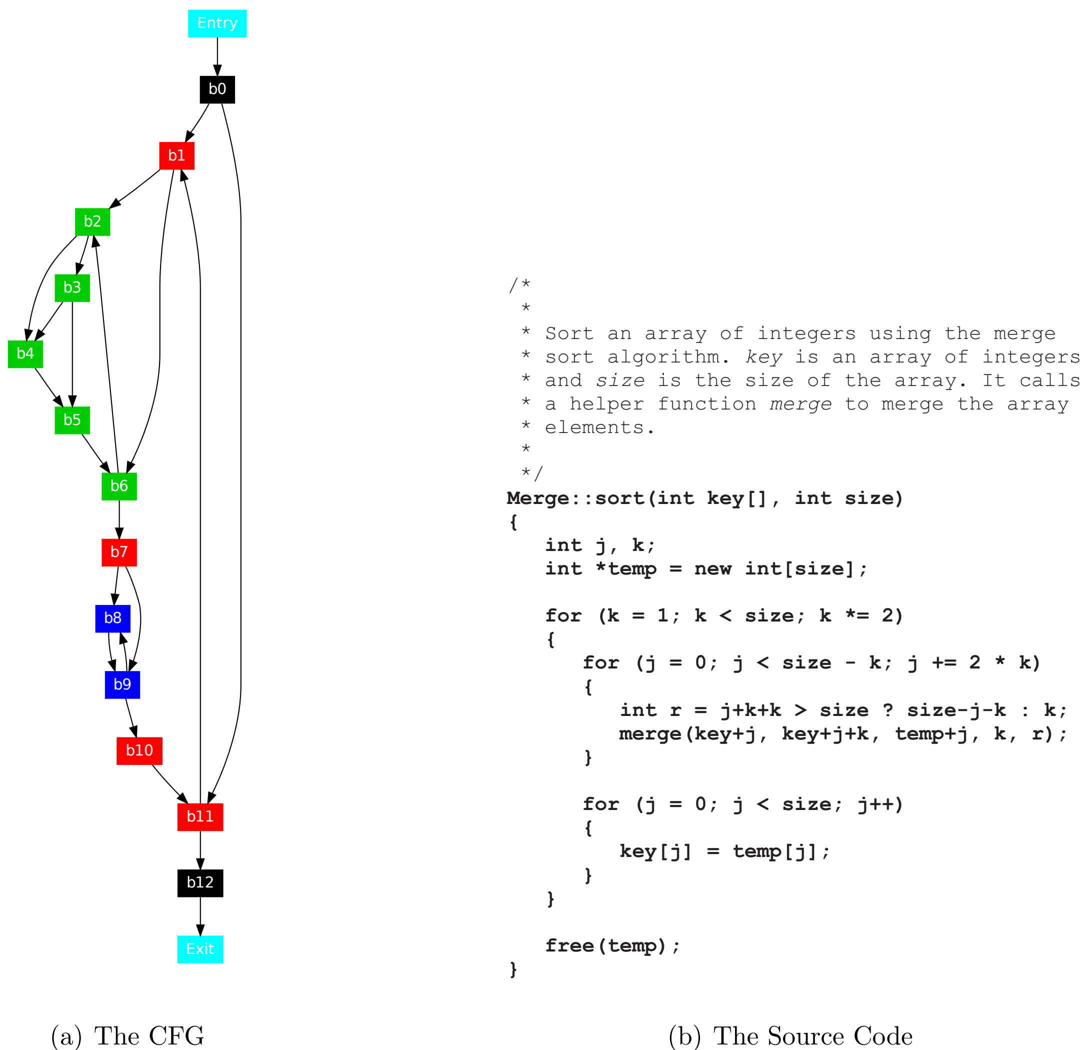
The source code was not made available to our binary analysis tool, and the CFG that is built by this tool is only based on the information available in the binary program.

This function has been instrumented, i.e: additional code has been added to this function. There are three instructions at the end (tagged *INC_A*, *JMP_A* and *CTR_A*) in Listing 3.1, and are not included in the binary analysis presented here. The addition of _A is to show that these instructions have been added by a binary instrumentation tool. Malware writers can use such tools to add malicious instructions. The first instruction *INC_A* increments a 32 bit counter *CTR_A* at address 4011e0. The second instruction *JMP_A* jumps over the counter storage space to address 4011e4 which contains the RET instruction. The third instruction *CTR_A* is not an instruction but a counter that counts the number of times this function is called. This kind of instrumentation is carried out during the binary analysis of a program for profiling and optimizing the program.

The CFG of the function shown in Figure 3.1 (a) starts at block 0 and ends at block 12. Block 11 jumps back to block 1, which indicates a loop (blocks 1 – 11). This loop has two inner loops, which consist of blocks 2 – 6 and blocks 8 and 9. The source code of the function *Merge::sort()* has one outer loop with two inner loops.

3.2.1 More Examples of Obfuscation

Chapter 1 provides some examples of obfuscations used in malware. This section provides some more examples of such obfuscations and some exploits that can be



(a) The CFG (b) The Source Code

Figure 3.1: The CFG and the Source Code in C++ of the Function in Listing 3.1

used for malicious purposes, using the code in Listing 3.1.

A malware writer can change (in just the machine code) the last instruction of block 11 in Listing 3.1:

```
from: 11  4011c0          0f85fdfeffff          JNZ 0x4010c3 (1)
to:   11  4011c0          ebfdfeffffff          JMP 0x4010c3 (1)
```

This simple example illustrates both an obfuscation and an exploit in the binary code, by changing the signature and making the outer loop an infinite loop. When the function *Merge::sort()* is called, the program never returns. The malware writer in this case has added an unconditional back jump, which in general is a legal jump.

Similarly other back jumps (last instructions of blocks 6 and 9) can also be changed by a malware writer to make more infinite loops.

What if this unconditional jump instruction is a legal instruction, i.e: it has not been added by a malware writer and is part of the program? For example event-based programs contain one or more infinite loops.

A signature based malware detection tool will not be able to detect such kinds of malware. Without the behavioral information for a binary program, obtained either statically or dynamically, manual detection with a debugger is required to detect such malware. This manual labor is very time consuming and can become very expensive financially.

Other malicious changes, such as the following control flow change combined with register renaming, in block 2 in Listing 3.1, cannot be detected by a signature based malware detector:

```
from:
   2  4010cf        8b45e8   MOV   EAX, [RBP-0x18]
   2  4010d2        8b55ec   MOV   EDX, [RBP-0x14]
   2  4010d5        8d0402   LEA    EAX, [RDX+RAX]
   2  4010d8        0345e8   ADD   EAX, [RBP-0x18]
   2  4010db        3b45bc   CMP   EAX, [RBP-0x44]
   2  4010de  0f8e11000000   JLE          0x4010f5
to:
   2  4010cf        8b5de8   MOV   EBX, [RBP-0x18]  ; EAX --> EBX
   2  4010d2        8b55ec   MOV   EDX, [RBP-0x14]
   2  4010d5        8d1c1a   LEA    EBX, [RDX+RBX]  ; EAX --> EBX
   2  4010d8        035de8   ADD   EBX, [RBP-0x18]  ; EAX --> EBX
   2  4010db        3b5dbc   CMP   EBX, [RBP-0x44]  ; EAX --> EBX
   2  4010de  0f8e10100000   JLE          0x4011e5  ; Jump to some malicious code
```

Another technique used by malware writers to deceive signature based detectors is to use instructions other than *JMP* and *CALL* to change the control flow of a program. We show this by replacing the last instruction with two instructions in block 7 in Listing 3.1 as follows:

```
from: 7  401179      e923000000   JMP              0x4011a1
to:   7  401179      68e5114000  PUSH        QWORD 0x4011e5
      7  40117e              c3   RET
```

The change in Listing 3.1 is not complete. For the code to work correctly, the addresses following these instructions and all the affected jump target addresses need

to be updated. A malware writer may or may not update them depending on the complexity of the malware. A tool could be used by the malware writer that can automate the updating of these addresses.

Further binary analysis on the above instructions reveals that the last value pushed on the stack before the RET instruction is 4011e5, so the RET instruction will load the value 4011e5 into the RIP register, the instruction pointer. The instruction at address 4011e5 (malicious code) will be the next one executed.

In order to detect such malware automatically and to distinguish between a malicious and a benign change, we may in general need to build specific control flow patterns (using the binary analysis presented in Section 3.2) and compare them with the previous control flow patterns of malware of this kind.

Sometimes the binary provides information about the start and end of all the functions in a program. If this information is not available, it can be difficult to determine where functions begin and end. For example, the addition of the two instructions shown above in Listing 3.1 in block 7 divides the function into two functions, and makes it difficult to find the original function. For malware detection we may only need to find where the control is flowing (i.e: just the behavior and not the function boundaries), and then compare this behavior with the previous samples of malware available to detect such malware.

We have shown above, using an elaborate example, how trivial changes in a binary program can make malware analysis and detection intricate, difficult and expensive. With suitable tools and appropriate binary analysis it is possible to analyse and detect such malware automatically. This is one of the goals of this thesis. In the next section we describe the design of the intermediate language MAIL that automates and optimizes this step.

3.3 Design of MAIL

We believe a good language must start small and simple, and must give opportunities to the language developers to grow (extend) the language with the users. Therefore MAIL is designed as a small, simple, and extensible language. In this and next subsections, we describe how MAIL is designed in detail.

The basic purpose of MAIL is to represent structural and behavioral information of an assembly program for malware analysis and detection. MAIL also makes the program more readable and understandable by a human malware analyst. An assembly program may consist of the following type of instructions (we use Intel x86-64 [27] assembly instructions as sample instructions):

Control instructions: include instructions that can change the control flow of the program, such as JMP, CALL, RET, CMP, CMPS, CMPPS, PCMPEQW, REP and LOOP instructions.

Arithmetic instructions: perform arithmetic operations, such as ADD, SUB, MUL, DIV, FSIN, FCOS, PADDW, PSUBW, ADDPS, ADDPD, PMULLD, PAVGW, DPPD, SHR and SHL.

Logical instructions: perform logical operations, such as AND, OR, and NOT.

Data transfer instructions: involve data moving instructions, such as MOV, CMOV, XCHG, PUSH, POP, LODS, STOS, MOVS, MOVAPS, MOVAPD, IN, OUT, INS, OUTS, LAHF, SAHF, PREFETCH, FLDPI, FLDCW, FXSAVE, LEA, and LDS.

System instructions: provide support for operating system functions and include instructions such as LOCK, LGDT, SGDT, LTR, STR and XSAVE, etc.

Miscellaneous instructions: All other instructions that do not fit into any of the above groups are included in this group of instructions, such as NOP, CPUID, SCAS, CLC, STC, CLI, HLT, WAIT, MFENCE, PACKSSWB, MAXPS, and UD (undefined instruction).

Designing a language that is small and simple, and accurately represents all these instructions for structural and behavioral information is non-trivial. Our goal is to create as few statements as possible in the intermediate language and map as many instructions as possible to these statements. For example we do not translate (i.e. we ignore) the following x86 instructions:

```
CLFLUSH:   Flush caches
```

```
CLTS:      Clear TLB (Translation lookaside buffer)
SMSW:      Restore machine status word
VERR:      Verify if a segment can be read
WBINVD:    Writing back and flushing of external caches
XRSTOR:    Restore processor extended states from memory
XSAVE:     Save processor extended states from memory
```

The complete list of x86 instructions that are not translated into the MAIL statements appears in [3]. We also provide examples of translating a x86 and an ARM assembly program into a MAIL program.

3.3.1 MAIL Statements

The MAIL statements are divided into the following 8 basic statements (the complete MAIL grammar is listed in Appendix A):

```
statements    ::= ( statement* ) ;
statement     ::= assignment_s+ | control_s+
                  | condition_s+ | function_s+
                  | jump_s+ | lib_call_s+
                  | 'halt' | 'lock' ;
```

Every statement in the MAIL language has a *type*, which is also called a *pattern*, that can be used for pattern matching during malware analysis and detection. These *patterns* are introduced and explained in Section 3.3.3. MAIL has its own registers but also reuses the registers present in the architecture that is being translated to the MAIL language. There are other special registers such as:

- **Flag registers:** ZF (zero flag), CF (carry flag), PF (parity flag), SF (sign flag) and OF (overflow flag). These flag registers are of size one byte and are used in conditional statements.
  e.g. if (ZF == 1) jmp 0x405632;
- **eflags:** stores the flag registers.
- **sp:** to keep track of the stack pointer.
- **gr and fr:** these provide an unbounded number of general purpose registers for use in integer and floating point instructions, respectively. When they are

used, they have a numeric suffix. Examples are gr1, gr2, gr3, fr1, fr2, and fr3 etc.

The majority of the assembly instructions are data movement instructions, as shown above. We introduce, in the following, two MAIL assignment statements covering the data transfer, arithmetic, logical and some of the system instructions. We use EBNF [33] notation to define these statements:

```
assignment_s  ::= register_s
                  | address_s ;
register_s    ::= register '=' (math_operator)? expr
                  | register '=' (expr)? math_operator expr
                  | register '=' lib_call_s ;
address_s     ::= address '=' (math_operator)? expr
                  | address '=' (expr)? math_operator expr
                  | address '=' lib_call_s ;

expr          ::= register | address | digit+ ;
register      ::= 'eflags'
                  | 'gr_' digit+ | 'fr_' digit+ | 'sp'
                  | register_name (':' register_name)? ;
register_name ::= letter+ ['0' - '9']?
                  | 'ZF' | 'CF' | 'PF' | 'SF' | 'OF' ;
address       ::= '[' digit+ ']' | reg_address
                  | 'UNKNOWN' ;
```

Control instructions are very important because they can change the behavior of a program, and they can be changed or added by polymorphic and metamorphic malware to avoid detection. The following MAIL control statement represents the control instructions:

```
control_s     ::= ('if' condition_s
                             (jump_s | assignment_s))
                  ('else' (jump_s | assignment_s))? ;
jump_s        ::= 'jmp' address ;
lib_call_s    ::= letter+ '(' address (, args)* ')' ;
function_s    ::= 'start_function_' digit+ statement
                  'end_function_' digit+ ;
condition_s   ::= (expr rel_operator expr)+ ;
```

3.3.2 MAIL Library

The current MAIL library contains 22 functions. The following are some of the examples of MAIL library functions:

- *compare(opl, opr):* compares two values *opl* and *opr* and then set the flag register.
- *max(opl, opr)* and *min(opl, opr):* returns the maximum and minimum of the parameters *opl* and *opr* respectively.
- *swap(opl, opr):* swap the bits in *opl* and write back in *opr*.

Details about all these library functions are given in [3]. These library functions can help in translating most of the complex assembly instructions present in current processor architectures. The purpose of these functions is not to capture the exact functionality of the assembly instruction(s) but to help in analysing the structure and behavior of the assembly program, and in capturing some of the patterns in the program that can help detect malware.

3.3.3 MAIL Patterns for Annotation

MAIL can also be used to annotate a CFG of a program using different patterns available in the language. The purpose of these annotations is to assign patterns to MAIL statements that can be used later for pattern matching during malware detection. There is a total of 21 patterns in the MAIL language as follows:

ASSIGN: An assignment statement, *e.g.* EAX=EAX+ECX;

ASSIGN_CONSTANT: An assignment statement including a constant, *e.g.* EAX=EAX+0x01;

CONTROL: A control statement where the target of the jump is unknown, *e.g.* if (ZF == 1) JMP [EAX+ECX+0x10];

CONTROL_CONSTANT: A control statement where the target of the jump is known. *e.g.* if (ZF == 1) JMP 0x400567;

CALL: A call statement where the target of the call is unknown, *e.g.* CALL EBX;

CALL_CONSTANT: A call statement where the target of the call is known, *e.g.* CALL 0x603248;

FLAG: A statement including a flag, *e.g.* CF = 1;

FLAG_STACK: A statement including flag register with stack, *e.g.* EFLAGS = [SP=SP–0x1];

HALT: A halt statement, *e.g.* HALT;

JUMP: A jump statement where the target of the jump is unknown, *e.g.* JMP [EAX+ECX+0x10];

JUMP_CONSTANT: A jump statement where the target of the jump is known, *e.g.* JMP 0x680376

JUMP_STACK: A return jump, *e.g.* JMP [SP=SP–0x8]

LIBCALL: A library call, *e.g.* compare(EAX, ECX);

LIBCALL_CONSTANT: A library call including a constant, *e.g.* compare(EAX, 0x10);

LOCK: A lock statement, *e.g.* lock;

STACK: A stack statement, *e.g.* EAX = [SP=SP–0x1];

STACK_CONSTANT: A stack statement including a constant, *e.g.* [SP=SP+0x1] = 0x432516;

TEST: A test statement, *e.g.* EAX and ECX;

TEST_CONSTANT: A test statement including a constant, *e.g.* EAX and 0x10;

UNKNOWN: Any unknown assembly instruction that cannot be translated.

NOTDEFINED: The default pattern, *e.g.* all the new statements when created are assigned this default value.

3.4 Conclusion

In this Chapter we have presented the design and development of the new language MAIL for malware anlaysis. The two main contributions of MAIL are: **(1)** Platform independence and automation for malware analysis and detection tools. **(2)** Optimizing the creation of a behavioral signature of a program. In the following chapters, we show how MAIL provides automation and optimization for malware analysis and detection.

Chapter 4

MARD (Metamorphic Malware Analysis and Real-Time Detection)

In this chapter, we define the model and discuss the design principles underlying our proposed framework for Metamorphic Malware Analysis and Real-Time Detection (MARD).

4.1 Model

Before discussing the design principles, we formally define MARD as follows:

MARD $= \langle C,\ B \rangle$,
where
C = set of components C_n;
B = set of bindings B_n;
n = number of components.

We define a component (that can contain other sub-components) of MARD as follows:

$C = \langle S,\ I,\ P,\ SC,\ SB \rangle$,
where
S = system platform of the component and the sub-components;
SC = set of components SC_k;
SB = set of bindings SB_k;
k = number of sub-components in the component;

I = set of interfaces or services provided by the component;

P = set of functional properties or characteristics of the component.

The SC and SB can be empty sets.

MARD contains two types of bindings, horizontal and vertical, that connect components.

Let us assume the following two components:

$C_i = \langle S_i, I_i, P_i, SC_i, SB_i \rangle$ and $C_j = \langle S_j, I_j, P_j, SC_j, SB_j \rangle$

We define horizontal binding H and vertical binding V between C_i and C_j as follows:

$H = I_i, I_j, P_i, P_j, SC_i, SC_j, SB_i, SB_j \models CM$

$V = S_i, S_j, I_i, I_j, P_i, P_j, SC_i, SC_j, SB_i, SB_j \models CM$

where

CM = component model and $\models$ means *complies* (staisfy the rules)

These bindings determine whether a set of mutually connected components can be treated as a component itself. That is, whether an assembly composed of a set of components fully complies with the rules imposed by the component model. The two major components of MARD, the *Front End* and the *Back End* are connected through horizontal binding, and all the other components are connected trough vertical bindings, as shown in Figure 4.1.

4.2 Design

Figure 4.1 gives an overview of MARD. First a training dataset is built, also called *Malware Templates* in Figure 4.1, using the malware training samples. After a program (sample) is translated to MAIL and to a behavioral signature (generated using one of the two proposed techniques described in Chapters 5 and 6) the *Similarity Detector* (Figure 4.1) detects the presence of malware in the program, using the *Malware Templates*. All the steps as shown in Figure 4.1 are completley automated. There is no manual intervention during the entire run. The tool automatically generates the report after all the samples are processed.

In the current version of MARD, the component *Unpacker* is not implemented, and we assume that all the samples are unpacked before they are disassembled. We have taken this assumption in accord with the view that if a program cannot be

unpacked by currently available unpackers then the program must be malware, and also any good available unpacker can be interfaced with MARD.

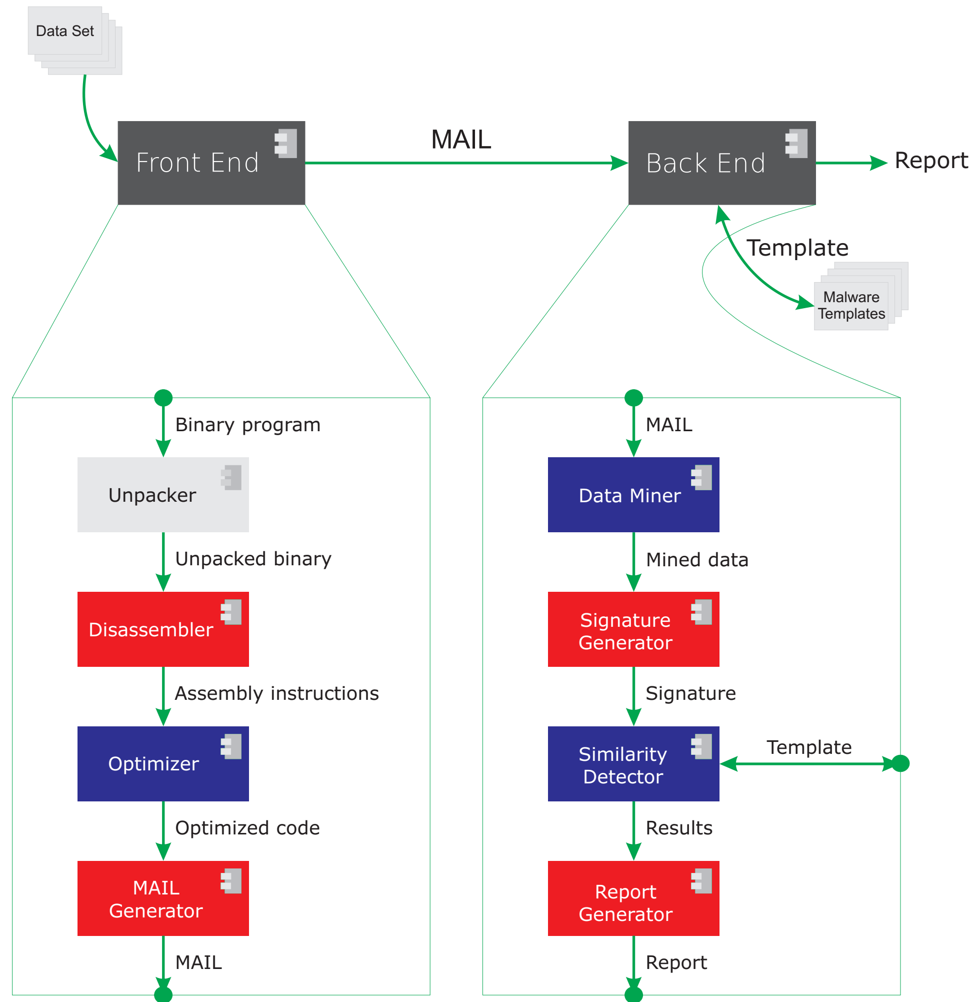


Figure 4.1: High Level Overview of MARD

4.3 Characteristics

Some of the major characteristics and advantages of MARD design described above are as follows:

Platform independence: To help make modern compilers [1, 67] platform-independent, they are divided into two major components: a *front end* and a *back end*. The design of the MARD framework follows the same principle. In compilers, the same C++ *front end* can be used with different *back ends* to generate code for different platforms such as x86, ARM and PowerPC etc. In the case of MARD the same *back end* can be used with different *front ends* to detect malware for different platforms (Windows and Linux etc). For example, we can implement a *front end* for the PE (Windows executable) files and another *front end* for the ELF (Linux executable) files. Both these *front ends* generate their output in our intermediate language (i.e. MAIL) that is used by the *back end*. Programs compiled for different architectures such as Intel x86 and ARM (the two most popular architectures) can be translated to MAIL. So we only need to implement one back end that is able to perform analysis and detect malware using MAIL. The use of MAIL in our design keeps the *front end* completely separate from the *back end* and therefore provides an opportunity for platform independence.

Optimization: The main purpose of the optimizations in the *front end* is to reduce the number and complexity of assembly instructions for malware analysis performed by the *back end*. We achieve this by: **(1)** Removing the unnecessary instructions that are not required for malware analysis such as NOP instructions etc. **(2)** Generating an optimized intermediate representation using MAIL, which provides a high level representation of the disassembled binary program. MAIL includes specific information such as control flow information, function/API calls and patterns etc, for easier and optimized analysis and detection of malware. We also use parallelization and graph reduction techniques to optimize the runtime of MARD.

Automation: The use of the MAIL language, that is generated by the *front end*, with data mining and similarity detecting in the *back end*, automates the complete process of malware analysis and detection. The *back end* generates a behavioral signature from the input. The system keeps a database of known malware as signatures. The signatures from this database are used by the similarity detector to detect known and unknown malware.

4.4 Components of MARD

The design of MARD is based on the principle of modularity [72] and is a component-based framework. First we divide the design into two completely separate components communicating using MAIL as discussed above. Then each of these components is further divided into separate sub-components with well defined interfaces. This section describes these components. The *Unpacker* component is not implemented in this version of MARD, so we do not describe it here.

The **Disassembler** first checks the format (PE or ELF) of the binary program and then disassembles the program according to that format. If the program is neither PE or ELF it still disassembles the program, but as a raw binary. There are two standard techniques for disassembling a binary program.

1. The *Linear Sweep* technique starts from the first byte of the code and disassembles one instruction at a time until the end. This technique assumes that the instructions are stored in adjacent memory location and hence does not distinguish between embedded data and actual instructions. When data is mixed with the code either by the compiler or by malware writers, this technique may produce wrong results. The advantage of this technique is that it provides complete coverage of the code.

2. The *Recursive Traversal* technique relies on the control flow of the program and decodes the bytes by following the control flow of the program. This technique only disassembles an instruction if it is referenced by another instruction. The advantage of this technique is that it distinguishes code from data. But in case of a branch whose destination address cannot be known statically, this technique may fail to find and disassemble valid instructions.

Both techniques have some deficiencies. To overcome these deficiencies a good disassembler would combine both techniques. One such open source disassembler, for non-commercial use, is *distorm* [30]. MARD uses *distorm* to disassemble a binary program.

The **Optimizer** performs normalization of the assembly code. The normalizations performed are the removal of NOP, junk and some of the prefixes such as *REP* in x86 assembly code etc. The Optimizer also prepares the assembly code to be translated

to MAIL by removing other instructions that are not required for malware analysis as explained in Chapter 3.

The **MAIL Generator** translates an assembly program to a MAIL program. We explain the approach in [3] with examples of translating a x86 and an ARM assembly program into a MAIL program. Some of the major tasks performed by this component are: (1) Translating each assembly instruction to the corresponding MAIL statement(s). Some of the assembly instructions, such as *PUSHA* (x86) and *STMDB* (ARM), are translated to more than one MAIL statement. (2) Assigning a pattern to each MAIL statement.

The **Data Miner** searches for the control and structural information in a MAIL program to help build a behavioral signature of the program. Details of this component are given in Chapters 5 and 6.

The **Signature Generator** generates a signature for a MAIL program. Currently it can generate two types of signature: ACFG (Annotated Control Flow Graph) and SWOD-CFWeight (Sliding Window of Difference and Control Flow Weight). More details of this component and the signatures are given in Chapters 5 and 6.

The **Similarity Detector** decides whether a program is benign or malware. The basic task of this component is to match the signature of the program against the signatures of the training samples based on thresholds that are computed empirically. We also parallelize this component to considerably reduce the runtime of MARD. Signature matching is explained in more detail in Chapters 5 and 6.

The **Report Generator** generates a report containing the results in a readable format. One of the reports generated by this component is listed in Appendix B.

4.5 Conclusion

In this Chapter we have presented a formal definition and described the design of the new proposed framework MARD. MARD uses MAIL and enables detection automation, platform independence, and optimizations for real-time performance. The current implementation of MARD provides two sub-components as part of the *Signature Generator* component. These two sub-components implement the two new techniques proposed in this thesis in Chapters 5 and 6 for building a behavioral signature for metamorphic malware. Antimalware developers can provide, write, and test their own components to generate a new malware signature based on MAIL for malware detection.

Chapter 5

ACFG (Annotated Control Flow Graph)

One of the goals of this thesis is to extract behavioral and structural information from a program to detect the presence of malware in the program. Control Flow Analysis (CFA) [1, 67] is one of the techniques used in compilers for optimizing a program. CFA captures control flow semantics of a program that can be used for malware analysis and detection. CFA is expressed as a Control Flow Graph (CFG), as defined below. Current techniques [4, 12, 16, 36, 37, 40, 46, 54, 71, 86] that use CFG for malware detection are either computationally intensive or have poor detection rates, cannot handle smaller size malware, and are not suitable for real-time detection.

We propose, in this chapter, a new technique named Annotated Control Flow Graph (ACFG), that provides a faster matching of ACFGs compared to other such techniques without compromising accuracy, can handle malware with smaller CFGs compared to other such techniques, and contains more information and hence provides better accuracy than a CFG.

5.1 Definitions

Before describing the technique proposed, we first define an ACFG as follows:

DEFINITION 1: A **Basic block** is a maximal sequence of instructions with a single entry and a single exit point. There is no instruction after the first instruction that is the target instruction of a jump instruction, and only the last instruction

can jump to a different block. Instructions that can start a basic block include the following. The first instruction, target of a branch or a function call, and a fall through instruction, i.e, an instruction following a branch, a function call or a return instruction. Instructions that can end a basic block include the following. A conditional or unconditional branch, a function call and a return instruction.

DEFINITION 2: **Control flow edge** is an edge that represents a control flow between basic blocks. A control flow edge from block a to block b is denoted e = (a, b).

DEFINITION 3: A **CFG** is a directed graph G = (V, E), where V is the set of basic blocks and E is the set of control flow edges. The CFG of a program represents all the paths that can be taken during the program execution.

DEFINITION 4: An **Annotated Control Flow Graph (ACFG)** is a CFG such that each statement of the CFG is assigned a MAIL Pattern.

5.2 ACFG For Metamorphic Malware Detection

In our proposed approch for metamorphic malware detection using ACFG, a binary program is first disassembled and translated to a MAIL program. The MAIL program is then annotated with patterns as described above. We then build a CFG of the annotated MAIL program yielding the corresponding ACFG. The constructed ACFG becomes part of the signature of the program and is matched against a database of known malware samples to see if the program contains a malware or not. This approach is very useful in detecting known malware but may not be able to detect unknown malware.

For detecting unknown malware, after a program sample is translated to MAIL, an ACFG for each function in the program is built. Instead of using one large ACFG as signature, we divide a program into smaller ACFGs, with one ACFG per function. A program signature is then represented by the set of corresponding (smaller) ACFGs. A program that contains part of the control flow of a training malware sample, is classified as a malware, i.e. if a percentage (compared to some predefined threshold) of the number of ACFGs involved in a malware signature match with the signature of a program then the program is classified as a malware.

5.2.1 Subgraph Matching

Before explaining the subgraph matching technique used in this paper for malware detection, we first define *graph isomorphism* [45] as follows:

Let $G = (V_G, E_G)$ and $H = (V_H, E_H)$ be any two graphs, where V_G, V_H and E_G, E_H are the sets of vertices and edges of the graphs, respectively.

DEFINITION 5: A vertex bijection (one-to-one mapping) denoted as $f_V = V_G \rightarrow V_H$ and an edge bijection denoted as $f_E = E_G \rightarrow E_H$ are **consistent** if, for every edge $e \in E_G$, f_V maps the endpoints of e to the endpoints of edge $f_E(e)$.

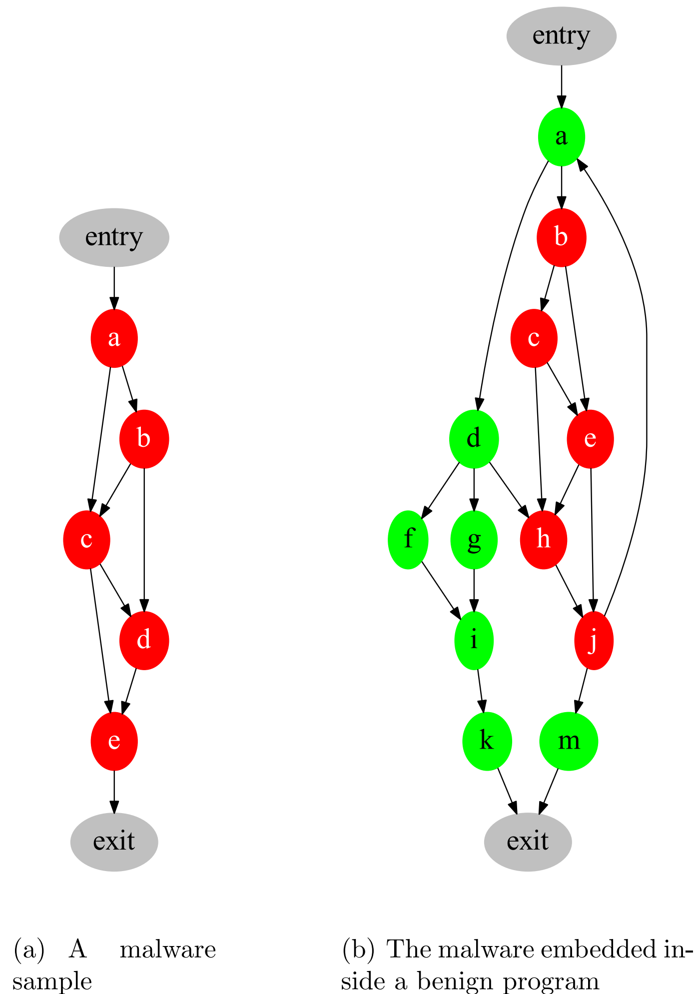
DEFINITION 6: G and H are **isomorphic graphs** if there exists a vertex bijection f_V and an edge bijection f_E that are consistent. This relationship is denoted as $G \cong H$.

In our malware detection approach, graph matching is defined in terms of **subgraph isomorphism**. Given two graphs, *subgraph isomorphism* determines if one of the graphs contains a subgraph that is isomorphic to the other graph. Generally, *subgraph isomorphism* is an NP-Complete problem [24]. The ACFG of a program is usually a sparse graph, therefore it is possible to compute the isomorphism of two ACFGs in a reasonable amount of time. An example of subgraph matching is shown in Figure 5.1.

Based on the definition of *graph isomorphism* presented above we formulate our ACFG matching approach as follows:

Let $P = (V_P, E_P)$ denote an ACFG of the *program* and $M = (V_M, E_M)$ denote an ACFG of the *malware*, where V_P, V_M and E_P, E_M are the sets of vertices and edges of the graphs, respectively. Let $P_{sg} = (V_{sg}, E_{sg})$ where $V_{sg} \subseteq V_P$ and $E_{sg} \subseteq E_P$ (i.e. P_{sg} is a subgraph of P). If $P_{sg} \cong M$ then P and M are considered as matching graphs.

After the binary analysis performed, we obtain a set of ACFGs (each corresponding to a separate function) of a program. To detect if a program contains malware we compare the ACFGs of the program with the ACFGs of known malware samples from



(a) A malware sample

(b) The malware embedded inside a benign program

Figure 5.1: An example of subgraph matching. The graph in Figure (a) is matched as a subgraph of the graph in Figure (b).

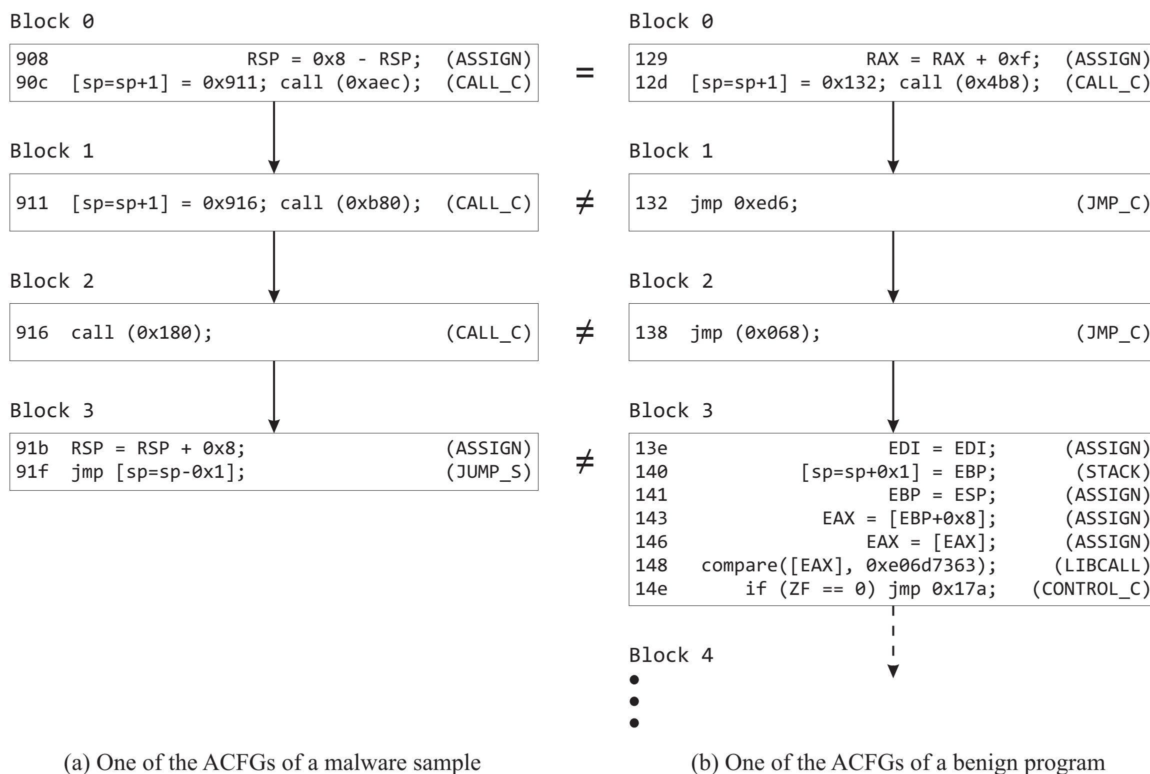
our training database. If a percentage of the ACFGs of the program, greater than a predefined threshold, matches one or more of the ACFGs of a malware sample (from the database) then the program will be classified as a malware.

5.2.2 Pattern Matching

Very small graphs when matched against a large graph can produce a false positive. Therefore, to alleviate the impact of small graphs on detection accuracy, we integrate a *Pattern Matching* sub-component within the *Subgraph Matching* component. Every statement in MAIL is assigned a *pattern* as explained in Chapter 3. If an ACFG

of a malware sample matches with an ACFG of a program (i.e. the two ACFGs are *isomorphic*), then we further use the *patterns*, assigned to MAIL statements, to match each statement in the matching nodes of the two ACFGs. A successful match requires all the statements in the matching nodes to have the same (exact) patterns, although there could be differences in the corresponding statement blocks.

An example of *Pattern Matching* of two *isomorphic* ACFGs is shown in Figure 5.2. One of the ACFGs of a malware sample, shown in Figure 5.2 (a), is *isomorphic* to a subgraph of one of the ACFGs of a benign program, shown in Figure 5.2 (b). The benign program is not detected as a malware, because not all the statements have the same pattern.



(a) One of the ACFGs of a malware sample

(b) One of the ACFGs of a benign program

Figure 5.2: Example of *pattern matching* of two *isomorphic* ACFGs. The ACFG in (a) is *isomorphic* to the subgraph (blocks 0 - 3) of the ACFG in (b).

5.3 Runtime Optimization with Parallelization

There are two components in the MARD framework that consume most of its runtime, one is the *Translator* (part of the *MAIL Generator* in Figure 4.1) and the other is the

Subgraph Matching (part of the *Similarity Detector* in Figure 4.1). The *Translator* reads each assembly instruction one by one and translates it to the MAIL language. The translation time increases with the size of the binary. There is little we can do to optimize this translation time. The *Subgraph Matching* component matches the graph (ACFG) against all the malware sample graphs. As the number of nodes in the graph increases the *Subgraph Matching* runtime increases. The runtime also increases with the increase in the number of malware samples.

There are two opportunities to parallelize the *Subgraph Matching* component.

1. Each ACFG matching in a sample is independent of the other matchings, these matchings can be performed in parallel.

2. Each sample can be processed independently of the other samples, the processing of the samples can be performed in parallel.

Multicore processors are becoming increasingly common. All the current desktops, laptops and even the energy efficient small mobile devices contain a multicore processor. Intel has recently announced its *Single-Chip Cloud Computer* [64], a processor with 48 cores on a chip.

Keeping in view the ubiquitousness of multicore CPUs in the host machines (also called the end points) and the two opportunities discussed above, to optimize the runtime, we decided to use Windows threads to parallelize the *Subgraph Matching* component.

We carried out an experiment using different numbers of threads ranging from 2 to 250. The number of malware samples used was 250 and the number of benign applications used was 30, each one with a different size for its ACFG. The main reason for this experiment was to develop an equation to estimate the number of threads to be used in the *Subgraph Matching* component, based on empirical results. The experiment was run on machines with 2 and 4 cores. Details of the machines used and the results of this experiment are shown in Table 5.1. The percentage of CPU utilization on average was 3 times more with threads than without threads. This is also confirmed by the improvement in the runtime.

As we increased the number of threads, reductions in the runtime were observed up to a certain number of threads (8 in 2 Cores and 64 in 4 Cores), after which the reductions became less evident. As we increased the number of benign samples from 30 to 1387 (the last row in Table 5.1), the runtime only reduced by a factor of almost

Table 5.1: Runtime improvement after parallelizing the *Subgraph Matching* component (using different number of threads)

2 Cores		4 Cores	
Number of Threads	**Runtime Reduced By**	**Number of Threads**	**Runtime Reduced By**
2	3.20 times	4	3.98 times
4	5.92 times	8	4.11 times
8	7.20 times	32	5.95 times
16	5.72 times	64	7.76 times
32	5.64 times	128	7.53 times
250	5.41 times	250	6.37 times
8	14.87 times	64	14.53 times

Machines used:
Intel Core i5 CPU M 430 (2 Cores) @ 2.27 GHz with 4GB of RAM and Windows 8 Professional installed
Intel Core 2 Quad CPU Q6700 (4 Cores) @ 2.67 GHz with 4GB of RAM and Windows 7 Professional installed
The machines used for this experiment have 2/4 Cores. Intel Cores use hyperthreading, so each core can run 2 threads at the same time. The maximum number of physical threads these machines can run is 4/8 respectively.

2. Beside other factors such as the number of CPUs/Cores and the memory available to the application, the main reasons for the smaller performance gains are more time being spent on thread management and increased communication between threads.

Based on this experiment we developed Equation 5.1 to estimate the maximum number of threads to be used by the *Subgraph Matching* component. We also give an option to the user to choose that maximum number of threads used by the tool for the *Subgraph Matching* component.

$$Threads = (NC)^3 \tag{5.1}$$

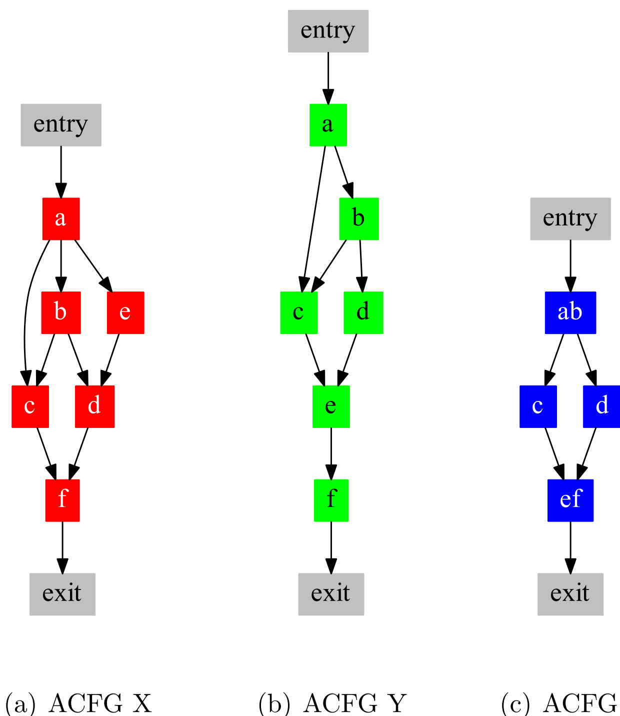
where NC = Number of CPUs/Cores

5.4 Runtime Optimization with ACFG Reduction

One of the advantages of using MAIL is that it provides patterns for malware detection. Our detection method use both subgraph matching and pattern matching techniques for metamorphic malware detection. Even if we reduce the number of blocks in an ACFG (it is possible for an ACFG of some binaries to reduce to very few number of blocks) we still get a good detection rate because of the combination of the two techniques, subgraph and pattern matching.

To reduce the number of blocks (nodes) in an ACFG for runtime optimization (reduce the time for subgraph matching) we carried out ACFG reduction, also called ACFG shrinking. We reduce the number of blocks in an ACFG by merging them together. Two blocks are combined only if their merger does not change the control flow of the program.

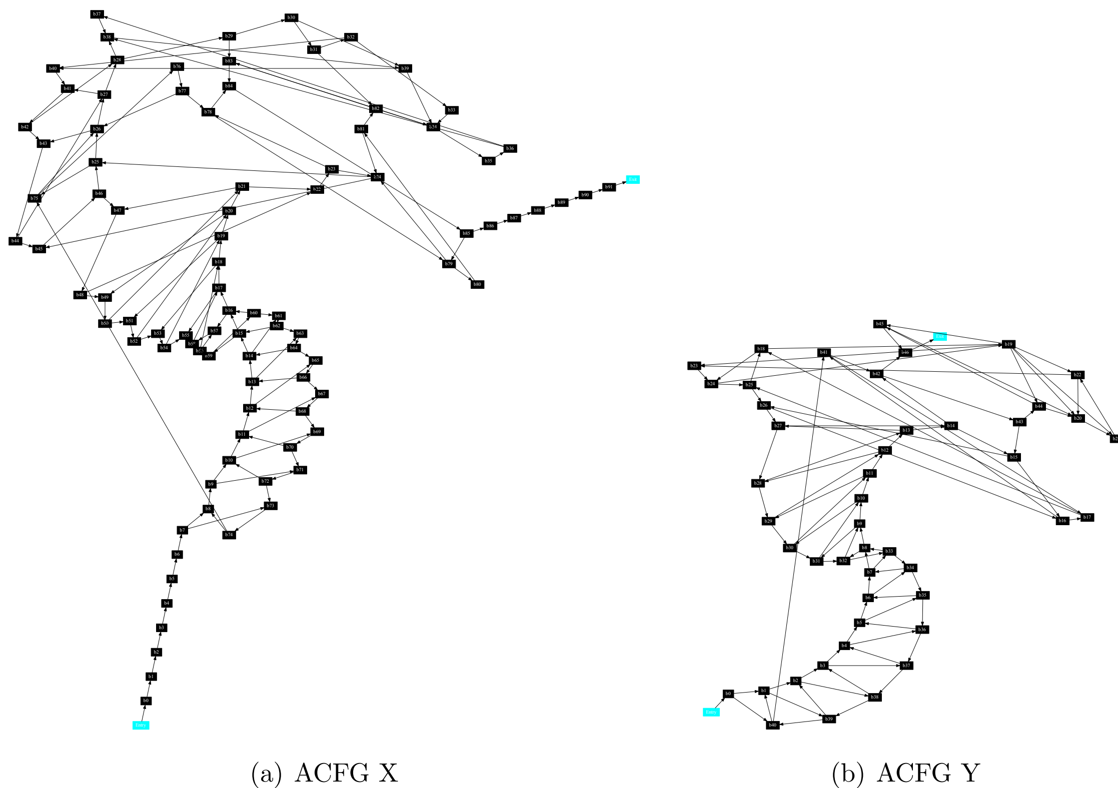
Given two blocks A and B in an ACFG, if all the paths that reach node B pass through block A, and all the children of A are reachable through B, then A and B are merged. An example of ACFG shrinking is shown in Figure 5.3.



(a) ACFG X (b) ACFG Y (c) ACFG Z

Figure 5.3: Example of ACFG shrinking. ACFG X is not shrinkable. ACFG Y with 6 blocks is shrinked to ACFG Z with 4 blocks.

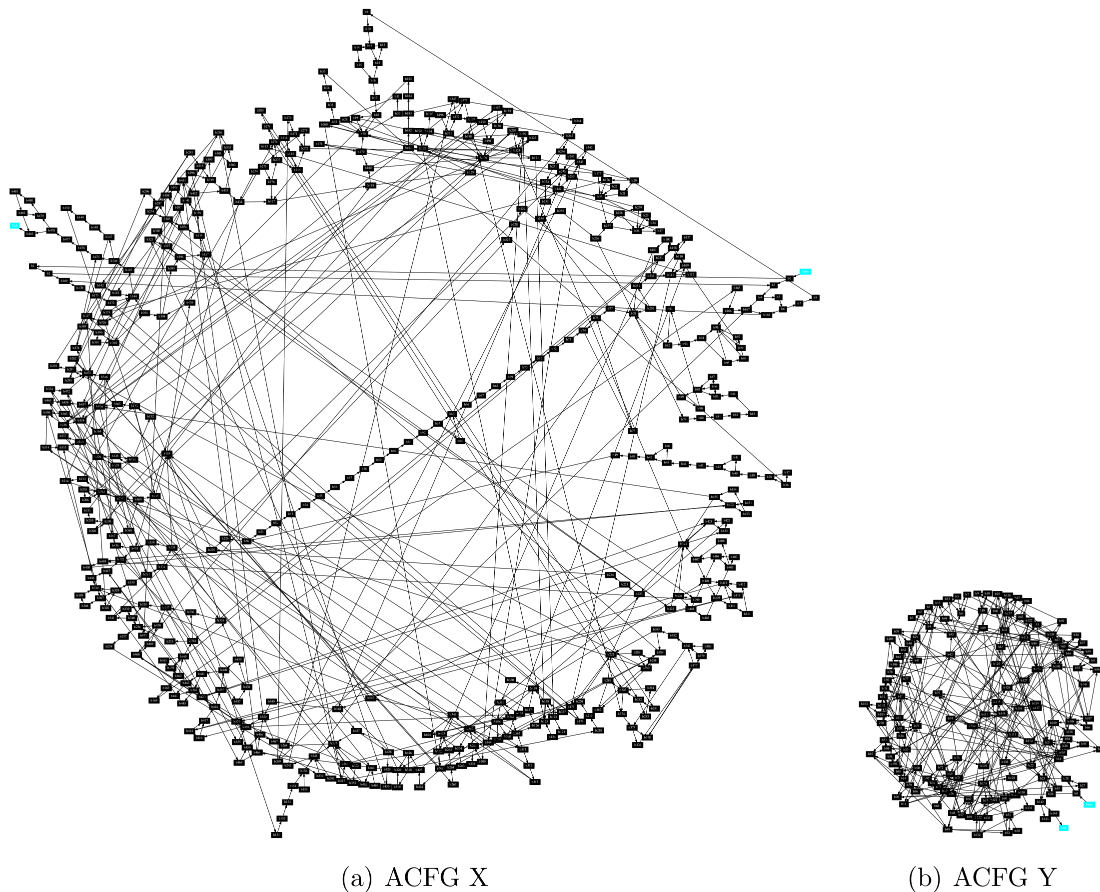
Figures 5.4, 5.5 and 5.6 show examples of ACFGs, from the dataset used in this paper, before and after shrinking. Figures 5.4 and 5.5 are ACFGs of functions of two different malware samples and Figure 5.6 is an ACFG of a function of a benign sample. The shrinking does not change the shape of a graph, as we can see in the Figures, the shapes of the graphs before and after shrinking are the same. More of these examples are available at [2].



(a) ACFG X (b) ACFG Y

Figure 5.4: Example of an ACFG, of one of the functions of one of the samples of the MWOR class of malware, before and after shrinking. The ACFG has been reduced from 92 nodes to 47 nodes.

We were able to substantially reduce the number of nodes per ACFG (in total a 90.6% reduction), as shown in Table 7.2. This reduced the runtime of MARD on average by a factor of 5 (for smaller datasets) and a factor of 100 (for larger dataset), while still achieving the same detection and false positive rates.

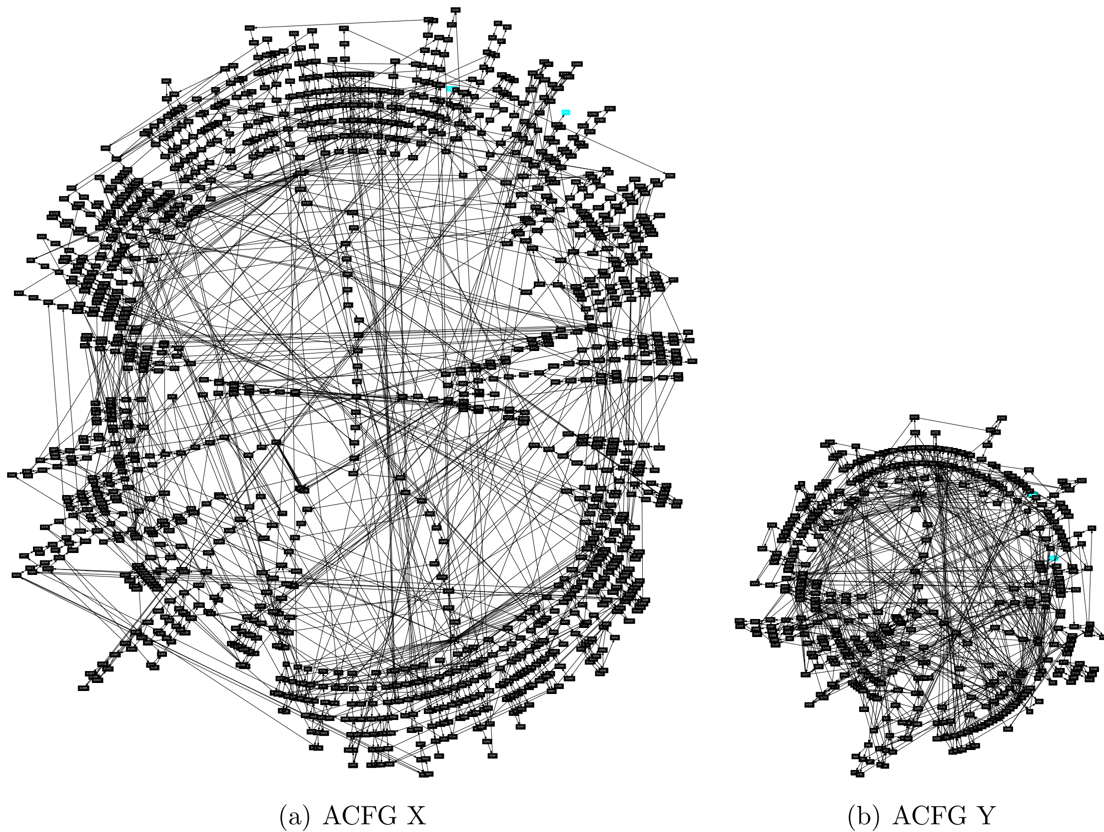


(a) ACFG X (b) ACFG Y

Figure 5.5: Example of an ACFG, of one of the functions of one of the samples of the MWOR class of malware, before and after shrinking. The ACFG has been reduced from 484 nodes to 145 nodes.

5.5 Summary

In this chapter we have presented a new scheme named Annotated Control Flow Graph (ACFG) to efficiently detect metamorphic malware. An ACFG is built by annotating the CFG of a binary program. It is used for graph and pattern matching to analyse and detect metamorphic malware. We also improve the runtime of malware detection through parallelization and ACFG reduction, while maintaining the same accuracy (as without ACFG reduction) for malware detection. The ACFG proposed in this chapter: (1) Captures the control flow semantics of a program. (2) Provides a faster matching of ACFGs and can handle malware with smaller CFGs, compared to other such techniques, without compromising the accuracy. (3) Contains more information and hence provides more accuracy than a CFG.



(a) ACFG X (b) ACFG Y

Figure 5.6: Example of an ACFG, of one of the functions of the Windows disk free space utility program *df.exe*, before and after shrinking. The ACFG has been reduced from 894 nodes to 283 nodes.

Chapter 6

SWOD-CFWeight (Sliding Window of Difference and Control Flow Weight)

We present in this chapter a new approach for malware detection that uses opcodes and new metrics based on a sliding window of difference.

6.1 Motivations and Overview

Techniques based on behavior analysis [12, 38, 40, 43, 46, 54, 86, 94, 97] are used to detect metamorphic malware, but are compute intensive and are not suitable for real-time detection. Some other techniques [5, 75, 78, 80, 82, 83, 93, 96] that use opcodes to detect malware, such as [5, 75, 78, 83, 93, 96], have been shown to detect metamorphic malware. One of the advantages of using opcodes for detecting malware is that it can be performed in real-time. However, the current techniques using opcodes for malware detection have several disadvantages including the following:

1. The patterns of opcodes can be changed by using a different compiler or the same compiler with a different level of optimizations.
2. The patterns of opcodes can also change if the code is compiled for a different platform.
3. Obfuscations introduced by polymorphic and metamorphic malware can change the opcode distributions.

4. The execution time depends on the number of features selected for mining in a program. Selecting too many features results in a high detection rate but also increases the execution time. Selecting too few features has the opposite effects.

We propose a new opcode-based malware detection technique which addresses the above limitations by transforming the assembly code to an intermediate language that makes the analysis independent of different compilers, ISAs and OSs. In order to mitigate the effect of obfuscations introduced by polymorphic and metamorphic malware we extract and analyze the control flow semantics of the program. Furthermore we use statistical analysis of opcode distributions to develop a set of heuristics that helps in selecting an appropriate number of features and reduces the runtime cost.

The MAIL language that we have introduced in Chapter 3 adequately addresses some of the problems described above. Table 6.1 depicts an example that highlights how the frequency of opcodes changes significantly compared to the frequency of a MAIL Pattern for a Windows program (*sort.exe*) compiled with different levels of optimizations. Patterns present in MAIL, which correspond to tokens, are a high level representation of opcodes and can be used in a similar manner. This high level representation of opcodes can help select an appropriate number of patterns that results in a high detection rate and considerably helps reduce the runtime. We use control patterns present in MAIL to include control flow information of a program for metamorphic malware analysis and detection.

In this chapter we introduce a novel scheme built around two new metrics derived from the notions of sliding window of difference (SWOD) and control flow weight (CFWeight) that help mitigate the challenges mentioned earlier. Likewise, we refer to the new technique as *SWOD-CFWeight*. *SWOD* is a window that represents differences in opcode distributions; its size can change, and it slides through an opcode distributions graph. *CFWeight* captures the control flow of a program to an extent that helps detect metamorphic malware in real-time. We show how they can be used on MAIL Patterns for effective metamorphic malware detection.

Table 6.1: An example, comparing the change in frequency of Opcodes with the change in frequency of MAIL Pattern *ASSIGN*, of a Windows program *sort.exe* compiled with different level of optimizations.

Opcode / **MAIL Pattern**	Optimization Level 0 Total Instructions 4045	Optimization Level 1 Total Instructions 1880		Optimization Level 2 Total Instructions 2276	
	Number of Instructions	**Number of Instructions**	**%age Change**	**Number of Instructions**	**%age Change**
MOV	1339 (33.1%)	532 (28.29%)	14.53	607 (26.66%)	19.50
ADD	115 (2.84%)	35 (1.86%)	34.51	49 (2.15%)	24.30
LEA	59 (1.46%)	43 (2.29%)	56.85	54 (2.38%)	63.01
SUB	57 (1.41%)	22 (1.17%)	17.02	27 (1.19%)	15.60
AND	23 (0.57%)	13 (0.69%)	21.05	11 (0.49%)	14.04
INC	4 (0.21%)	21 (0.52%)	147.62	18 (0.79%)	276.20
OR	14 (0.35%)	4 (0.21%)	40	4 (0.17%)	51.43
NEG	3 (0.16%)	5 (0.12%)	25	6 (0.27%)	51.43
XOR	53 (1.31%)	62 (3.30%)	151.91	60 (2.63%)	100.76
ASSIGN	1692 (41.83%)	761 (40.48%)	**3.22**	866 (38.10%)	**8.91**

While translating an assembly program to MAIL, all the 9 opcodes shown are tagged with pattern *ASSIGN*. We can see the frequencies of the 9 opcodes change from 21.05% to as much as 151.91% from optimization level 0 to optimization level 1, and from 14.04% to as much as 276.20% from optimization level 0 to optimization level 2, whereas the frequency of the MAIL Pattern *ASSIGN* changes only by 3.22% and 8.91%.

6.2 Statistical Analysis of MAIL Pattern Distributions for Metamorphic Malware

A recent study [10] analysed the statistical distributions of opcodes in 77 malware and 67 benign programs. The study found that the malware opcode fequencies differ significantly from non-malicious code and this difference can be used for malware detection. In general MOV and PUSH were the most used opcodes in the samples tested. This is the most significant research on opcodes and provides ideas and motivations to use opcodes [80, 82] for malware detection. Another recent study [5] presented some interesting opcode statistics of the assembly code produced by different compilers. The study found that the opcode distributions are different between compilers, and can be used to identify the compiler. These results also confirm that the frequencies of opcodes change through use of a different compiler.

Motivated by these studies [5, 10], we carried out an empirical study using metamorphic malware. Our study differs from the studies described in [5, 10], since it focuses specifically on metamorphic malware and MAIL patterns. The main purpose of our study is to study MAIL pattern distributions by computing term frequencies, and establish how MAIL pattern frequencies differ between malware and benign samples. We then use the findings as the basis for defining new metrics from MAIL patterns for detecting metamorphic malware. We describe in this section the dataset used in our work and study mail pattern distributions based on a subset of the dataset.

6.2.1 Dataset

The dataset used for the experiments consists of 5305 sample programs. Out of the 5305 programs, 1020 are metamorphic malware samples collected from three different resources [63, 75, 78], and the other 4285 are Windows and Cygwin benign programs. The dataset distribution based on the size of each sample file is shown in Table 6.2.

The dataset contains programs compiled with different compilers (Visual C++, Visual C#, Visual Basic and GCC) for the Windows (32 and 64 bits, PE format) and Linux (32 and 64 bits, ELF format) operating systems. The sizes of the malware samples range over 1 KB – 299 KB and the sizes of the benign samples have a range of 9 bytes – 10 MB.

The 1020 malware samples belong to the following three metamorphic *family* of viruses: Next Generation Virus Generation Kit (NGVCK) [69], Second Generation

Table 6.2: Dataset distribution based on the size of each program sample

Malware samples (1020)		Benign program samples (4285)	
Range of Size (bytes)	**Number of Samples**	**Range of Size (bytes)**	**Number of Samples**
1343 – 1356	50	9 – 19997	817
3584 – 4096	35	20336 – 29984	406
8192 – 16384	215	30144 – 49800	492
29356 – 35324	200	50144 – 119936	935
36864 – 40772	102	120296 – 980840	1514
40960 – 46548	101	1001936 – 1553920	50
52292 – 57828	200	1606112 – 3770368	48
67072 – 69276	101	4086544 – 5771408	12
70656 – 74752	4	6757888 – 8947200	4
271872 – 299520	12	9074688 – 10124353	7

Virus Generator (G2) [41] and Metamorphic Worm (MWOR) generated by metamorphic generator [63]. NGVCK and MWOR family of viruses are further divided into two and seven *classes* respectively. This *class* distribution is shown in Table 6.3.

This variety of sample files and malware *classes* in the samples provides a good testing platform for the proposed malware detection scheme.

6.2.2 MAIL Pattern Distributions

We used a subset of the dataset presented above that consists of 25% of the samples selected randomly to compute and analyze mail pattern distributions. First, we translated each sample from the dataset to the corresponding MAIL program and then collected the statistics about the MAIL Patterns distributions for each sample. The distributions for each program sample, benign and malware, are shown in Figure 6.1. Only seven MAIL pattern distributions that have a significant difference in values are shown. For clarity and readability we made the graphs sparse, and not all the values of the MAIL pattern distributions are shown.

The graph plots in Figure 6.1 show the number of samples on the X-axis and the Patterns' percentage on the Y-axis. On average the occurrence of MAIL pattern

Table 6.3: *Class* distribution of the 1020 metamorphic malware samples

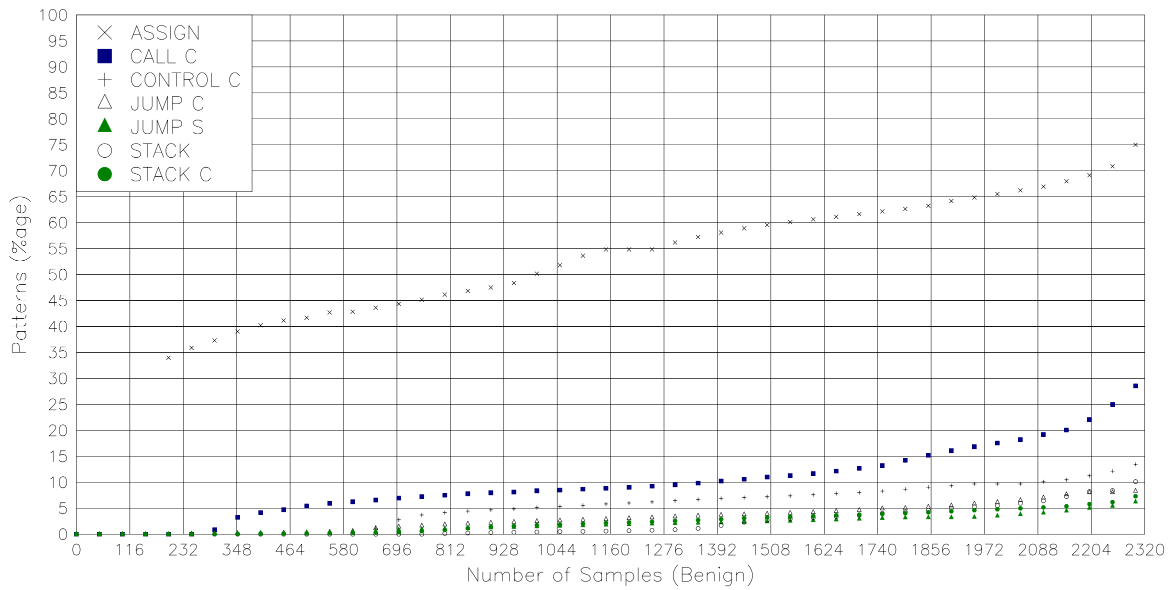
Class	**Number of Samples**	**Comments**
NGVCK_1	70	Generated by NGVCK with simple set of obfuscations, such as dead code insertion and instruction reordering etc.
NGVCK_2	200	Generated by NGVCK with complex set of obfuscations, such as indirect jump (e.g; push followed by a ret instruction) to one of the data sections etc.
G2	50	Generated by G2
MWOR_1	100	Generated by MWOR with a padding ratio of 0.5
MWOR_2	100	Generated by MWOR with a padding ratio of 1.0
MWOR_3	100	Generated by MWOR with a padding ratio of 1.5
MWOR_4	100	Generated by MWOR with a padding ratio of 2.0
MWOR_5	100	Generated by MWOR with a padding ratio of 2.5
MWOR_6	100	Generated by MWOR with a padding ratio of 3.0
MWOR_7	100	Generated by MWOR with a padding ratio of 4.0

MWOR uses two morphing techniques: dead code insertion and equivalent instruction substitution. The padding ratio is the ratio of the number of dead code instructions to the core instructions of the malware. A padding ratio of 0.5 means that the malware has half as many dead code instructions as core instructions [63].

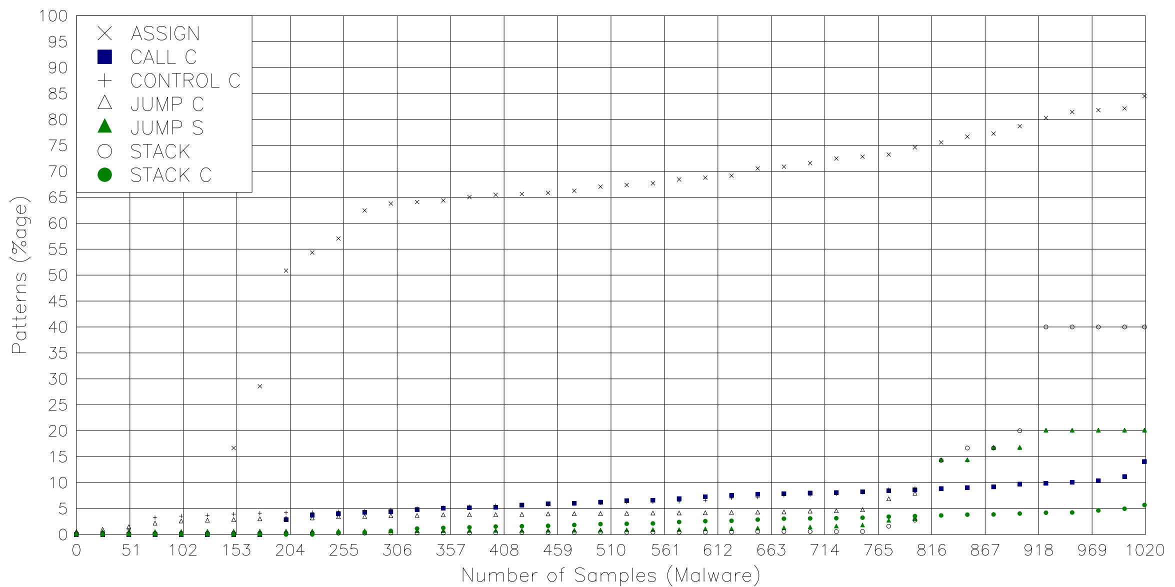
ASSIGN is the highest in over 85% of the samples. While translating a binary program to a MAIL program, all assembly instructions that make changes to a register or a memory place excluding stack, e.g., MOV instructions, are tagged with MAIL pattern *ASSIGN*. In [10], MOV instruction is also the most used opcode in the samples tested.

To show the difference between the malware and benign MAIL pattern distributions we superimpose the two Figures 6.1(a) and 6.1(b) using all the values of the MAIL Pattern distributions. For clarity and readability only three Patterns, *ASSIGN*, *CONTROL_C* and *STACK*, are superimposed. Figure 6.2 shows the superimposition of these MAIL patterns distributions.

The graph in Figure 6.2 is divided using perpendicular lines. When the two plots (malware and benign) of a pattern horizontally divide the space between two per-



(a) MAIL Pattern distributions for benign samples



(b) MAIL Pattern distributions for malware samples

Figure 6.1: MAIL Patterns distributions based on the percentage of the MAIL Patterns in each sample in the dataset

pendicular lines, this division is called a *pocket of the window*. There are significant pockets of windows in MAIL pattern *ASSIGN* that show significant differences between malware and benign samples. There are a few pockets of windows in MAIL patterns *CONTROL_C* and *STACK* that show significant difference between malware

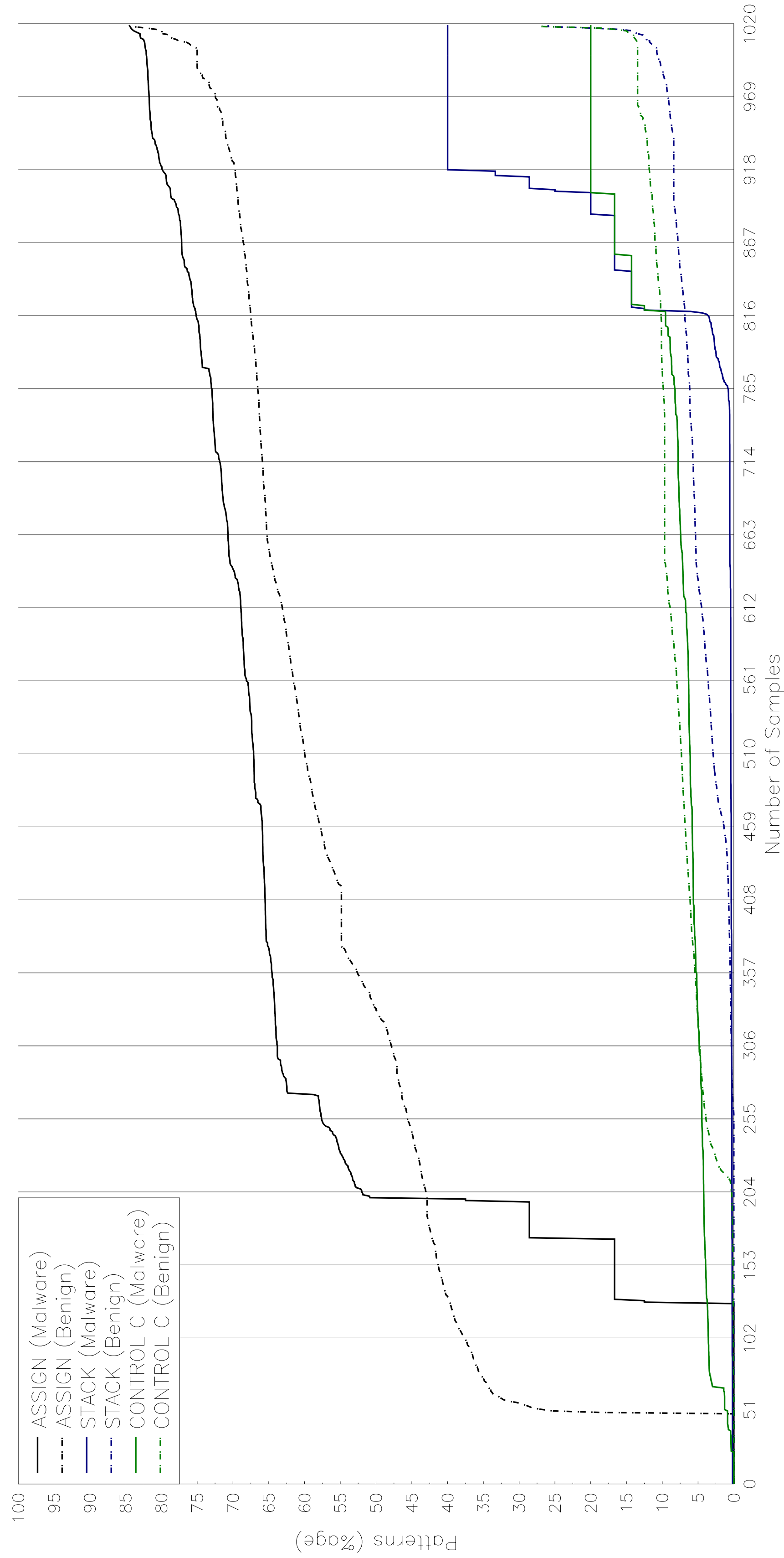


Figure 6.2: Superimposing three of the MAIL Patterns distributions from Figures 6.1(a) and 6.1(b).

and benign samples. We use this observation and the differences as a motivation to develop a Sliding Window of Difference ($SWOD$) based on MAIL patterns, which we define formally in Section 6.3.1. Instead of using the probabilities of the occurrences and differences of MAIL patterns in a dataset and then applying stochastic processes, we employ empirical methods using heuristics to develop the $SWOD$ for a dataset. We believe this is much closer to a practical solution to the problems of malware detection described above.

Out of the seven patterns shown in Figure 6.1 four of them namely, *CALL_C*, *CONTROL_C*, *JUMP_C* and *JUMP_S* are the MAIL control patterns. We can use this difference in MAIL control patterns between a malware sample and a benign sample for metamorphic malware detection. These statistics persuaded us to develop *CFWeight* (Control Flow Weight) as a metric that captures the amount of change in the control flow of a program.

6.3 Metamorphic Malware Detection Model

In this section we introduce in detail our proposed metamorphic malware detection technique. We define a set of heuristics based on MAIL patterns that underlie our detector. The main goal for developing these heuristics is to reduce the runtime for metamorphic malware detection while keeping the same or improving the other performance metrics.

6.3.1 Sliding Windows of Difference

Assume that we have a dataset D consisting of m malware samples and b benign samples. Let M_i be the *ith* malware sample and P_jM_i be the percentage of the *jth* MAIL pattern in M_i. Similarly, Let B_i be the *ith* benign sample and P_jB_i be the percentage of the *jth* MAIL Pattern in B_i. We compute the MAIL pattern distributions in a MAIL program, as follows:

$$P_jX_i = \frac{p_j}{\sum_{i=1}^{N} p_i} \times 100 \tag{6.1}$$

where X_i can be either a malware or a benign sample, p_j and p_i are the number of times the *jth* and *ith* patterns occur in a MAIL program, and N is the total number of patterns in the MAIL language; currently there are 21 patterns in MAIL ($N = 21$).

Let l denote the minimum of b and m: *l=min(m,b)*. Given a mail pattern P_j, let PD_{mj} denote the l first mail pattern distributions of P_j in the dataset of malware samples **sorted in decreasing order**: $PD_{mj} = \{P_jM_1,\ P_jM_2,\ P_jM_3, \ldots \ldots P_jM_l\}$.

Similarly, PD_{bj} denote the l first mail pattern distributions of P_j in the dataset of benign samples: $PD_{bj} = \{P_jB_1,\ P_jB_2,\ P_jB_3, \ldots \ldots P_jB_l\}$.

Assume that b and m are selected such that $N \leq l$. Let x denote an integer such that $1 \leq x \leq N$.

Using the above notation, we define the components of our sliding window of difference scheme as follows:

DEFINITION 7: **Vertical Window of Difference (VWOD)** for pattern P_j is the absolute difference between the percentages of the occurrences of P_j in a malware sample M_k and a benign sample B_k, where $1 \leq k \leq l$, and is defined as $VWOD_{jk} = |P_jM_k - P_jB_k|$. This difference is referred to as the size of $VWOD_{jk}$. A minimum value for the size of $VWOD_{jk}$, denoted ***minsize***, is predefined as a parameter of our proposed malware detector.

DEFINITION 8: **Horizontal Window of Difference (HWOD)** for pattern P_j over interval $[(x-1)n+1, xn]$ is defined as the set $HWOD_{jx} = \{\, VWOD_{j[(x-1)\times n+1]},\ VWOD_{j[(x-1)\times n+2]},\ VWOD_{j[(x-1)\times n+3]}, \ldots \ldots,\ VWOD_{j[x\times n]}\}$, where $\boldsymbol{n} = \lceil \frac{l}{N} \rceil$. Size of $HWOD_{jx} = x \times n - [(x-1) \times n + 1] + 1 = n$.

DEFINITION 9: We define the *ratio* for $HWOD_{jx}$, denoted *ratio($HWOD_{jx}$)*, as the percentage out of n of *VWODs* in $HWOD_{jx}$ that are greater or equal to the *minsize*. A minimum value for the *ratio* denoted ***minratio*** is predefined as a parameter of our proposed malware detector. For example, a *minratio* of 70 for a *HWOD* means 70% of all the *VWODs* in the *HWOD* are greater or equal to *minsize*.

DEFINITION 10: A window $HWOD_{jx}$ satisfies *minratio* denoted $HWOD_{jx}$ **sat** *minratio* if and only if *ratio($HWOD_{jx}$)* $\geq$ *minratio*. We refer to this particular kind of window as a **Sliding Window of Difference (SWOD)** for pattern P_j over

interval $[(x-1)n+1, xn]$ denoted $SWOD_{jx}$[1].

DEFINITION 11: The *SWOD* for pattern P_j denoted $SWOD_j$ is the union set of all the $SWOD_{jx}$:
$SWOD_j = \bigcup_{1 \leq x \leq N} SWOD_{jx}$

The determination of the SWOD depends on the values of *minsize* and *minratio*. They are computed empirically by first selecting a suitable number of malware samples and benign samples that adequately represent the dataset. Using these samples we obtain the values of *minsize* and *minratio* that yield the best performance for the malware detector. A graphical depiction of a *SWOD* is shown in Figure 6.3.

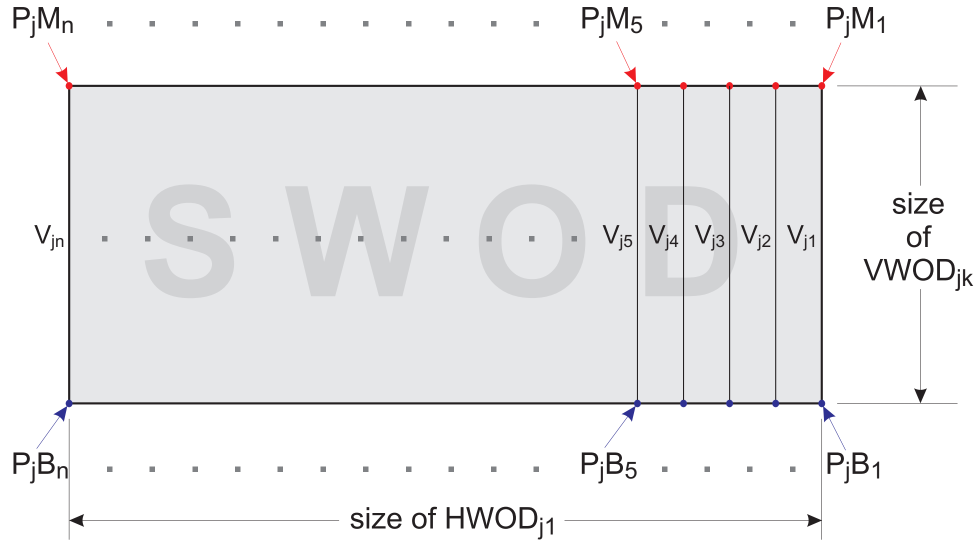


Figure 6.3: Sliding Window of Difference ($SWOD_{j1}$) as defined in *Definition 10*. $HWOD_{j1} = \{V_{j1}, V_{j2}, V_{j3}, \ldots V_{jn}\}$, where $V_{j1}, V_{j2}, V_{j3}, \ldots V_{jn}$, are the *VWODs*.

Figure 6.4 shows an example of *SWOD* for the MAIL pattern *ASSIGN* computed using 25% of samples selected randomly from the dataset presented earlier, which corresponds to $l = 1020$ samples.

The *SWOD* for the example assumes a *minsize* = *5* and *minratio* = *60*, and considers only a subset of the MAIL patterns consisting of $N = 20$ patterns, which gives $n = 51$ $(\frac{1020}{20})$.

[1]In other words, $HWOD_{jx}$ is a $SWOD_{jx}$ if and only if it satisfies the *minratio*.

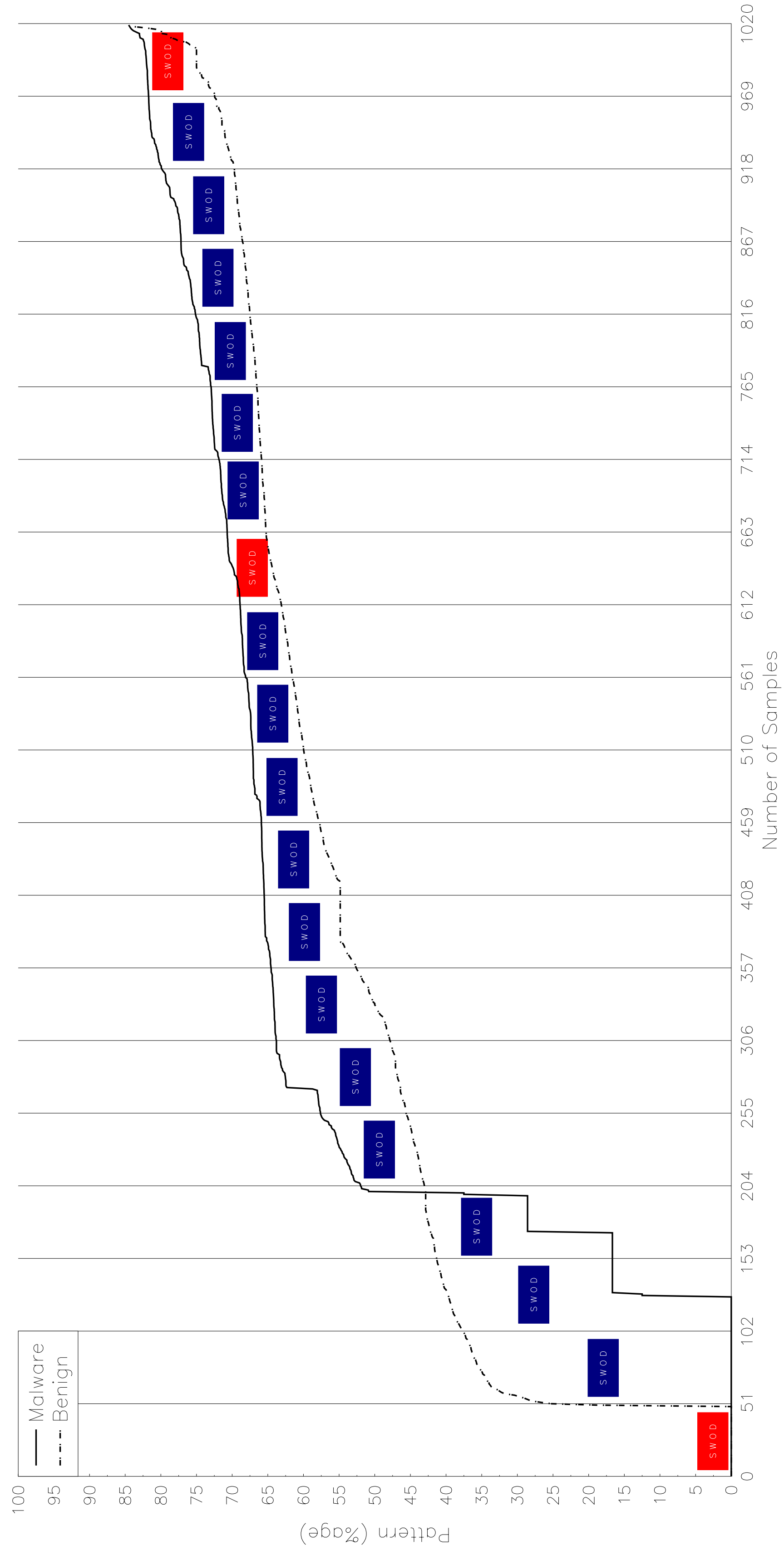


Figure 6.4: Sliding Window of Differences (*SWODs*) for the MAIL Pattern *ASSIGN*.

According to the definitions presented above at least 30 ($51 \times 0.60 \approx 30$) of the *VWODs* should have a value ≥ 5, for the example of *SWODs* shown in Figure 6.4. Since we assume $N = 20$ for the example, there are in total 20 virtual *SWODs* (or *HWODs*), and while sliding through the graph of the MAIL Pattern *ASSIGN*, 17 of them satisfy the *minratio* condition, and can then be considered as the real *SWODs*.

In our work, the cardinality or size of $SWOD_j$ represents the weight of pattern P_j over the dataset. For instance, in the above example, the weight assigned to the MAIL Pattern *ASSIGN* is 17.

Algorithm 1 presents in detail the steps for computing the weight of a MAIL pattern in a dataset.

Algorithm 1 Computing weight for a MAIL pattern in a dataset

```
1  procedure ComputeWeight(PM, PB, N, m, b, minsize, minratio)
2      for p ← 1 to N do
3          for i ← 1 to m do
4              ListMalware[p] ← PM[N][i]
5          end for
6          for i ← 1 to b do
7              ListBenign[p] ← PB[N][i]
8          end for
9          ListMalware[p] ← Sort(ListMalware[p])
10         ListBenign[p] ← Sort(ListBenign[p])
11     end for
12     n ← ⌈min(m, b) / N⌉
13     for p ← 1 to N do
14         SWODs ← ComputeSWODs(ListMalware[p], ListBenign[p], minsize, minratio, n)
15         PatternWeights[p] ← SWODs.Size
16     end for
17     return PatternWeights
18 end procedure
```

PM and PB are the MAIL pattern distributions. $minsize$, $minratio$ and n are defined in *Definitions 7 – 9*. N is the number of patterns in MAIL. m is the number of malware samples and b is the number of benign samples in the dataset. The function *Sort* sorts the list in descending order. $SWODs = \{SWOD_1, SWOD_2, SWOD_3, ...\}$ are computed as per *Definition 11*.

Algorithm 1 computes the difference of the data value in two lists, *ListMalware* and *ListBenign* to find the *SWODs*. These lists are the distributions for each MAIL pattern as defined earlier. We give priority to the samples that have greater occurrences of a MAIL pattern, therefore we first sort the lists in descending order, as shown in lines 9 and 10. For finding the real *SWODs* (as explained above) we go through candidate *SWODs* in the lists, as shown in lines through 13 – 16 in Algo-

rithm 1, and stop with the shorter list. The number of real *SWODs* found in a MAIL Pattern is assigned as weight of the MAIL pattern.

6.3.2 Control Flow Weight and MAIL Program Signature

We assign to each statement in a MAIL program a partial weight computed from a control flow graph (CFG), and referred to as *CFWeight* (for Control Flow Weight). Before computing *CFWeight* we build an interprocedural CFG (a CFG for each function) of a MAIL program. The heuristics for computing ***CFWeight*** are summarized as follows.

1. Each block's last statement, and each JUMP and CALL statement is assigned a weight of 1.
2. Each CONTROL statement is assigned a weight of 2.
3. Each control flow change (JUMP, CONTROL or CALL) is assigned a weight equal to the length of the jump, which correlates with the number of blocks jumped.
4. The weight of a backwards jump (possibly used for a loop) is double the length of the jump.
5. A jump whose target is outside the function is assigned a weight equal to the distance (measured as the number of blocks) of the jump statement from the last block of the function + 1.

Every MAIL statement is assigned a pattern during translation from assembly language to MAIL. The *CFWeight* of a MAIL statement is computed by adding all the weights assigned to the elementary statements involved in it, based on the above categorization.

The final weight of a MAIL statement is the sum of its *CFWeight* and the weight of the pattern assigned to the statement. The final weights of the statements of a MAIL program are stored in a weight vector that represents the program signature. This signature is then sorted in ascending order for easy and efficient comparison.

Algorithm 2 presents in detail the steps for building the signature of a MAIL program. In Algorithm 2, lines 4 – 20 express the heuristics for computing *CFWeight*. Lines 21 – 27 of Algorithm 2 show the computation of a MAIL program signature using the final weight of each MAIL statement in the program computed earlier.

6.3.3 Signature Matching and Malware Detection

To detect if a new program is malware or not, we build its signature as explained above. This signature is compared with the signatures of all the training malware samples. In case of a match with any of the signatures of the training malware samples we tag the new program as a malware.

Let $S_i = [s_{ik}]_{1 \leq k \leq p}$ and $S_j = [s_{jk}]_{1 \leq k \leq q}$ denote the signature vectors, sorted in decreasing order, of two different programs, such that $p \leq q$.

To match the two signatures, we compare each weight's value in one of the signatures with the corresponding weight's value in the other signature, by computing the following:

$d_{ijk} = \left| \frac{s_{ik} - s_{jk}}{max(s_{ik}, s_{jk})} \right| \times 100$, where $1 \leq k \leq p$.

Two weights s_{ik} and s_{jk} match if and only if $d_{ijk} \leq \epsilon_1$, where $\epsilon_1 = 3 \times minsize$.

Let y and r denote the total number of weight pairs (s_{ik}, s_{jk}) and the number of non-zero values in S_j, respectively. Signatures S_i and S_j match if and only if $\frac{y}{r} \times 100 \geq \epsilon_2$, where $\epsilon_2 = \frac{minratio}{2.25}$.

The two values, ϵ_1 and ϵ_2 are computed empirically from the dataset.

Figure 6.5 shows an example of malware detection using MAIL program signatures. There are in total 19 statements in the sample MAIL program. To generate the signature each statement is assigned a weight as explained above. After sorting the signature, it is stored in an index-based array for efficient comparison. This index-based array stores the number of weights with same value at the index corresponding to the value. For the example shown in Figure 5 there are three weights with the value 3, so we store a 3 at index 3. There is only one weight with the value 5, so we store a 1 at index 5, and so on. This index-based signature array of the MAIL program is compared with the index-based signature arrays of the training malware samples. We can see that there is a successful match of the signature of the MAIL program with the signature of the malware sample number M12 and hence the program is tagged as a malware. The comparison stops after the first successful match with the signature of a malware sample. The lines without a $\times$ (12 of them, $\geq \sim\epsilon_2$) show a match (percentage of difference is $\leq \epsilon_1$) between the corresponding weights, and the lines marked with a $\times$ (6 of them) show no match (percentage of difference is $> \epsilon_1$) between the corresponding weights.

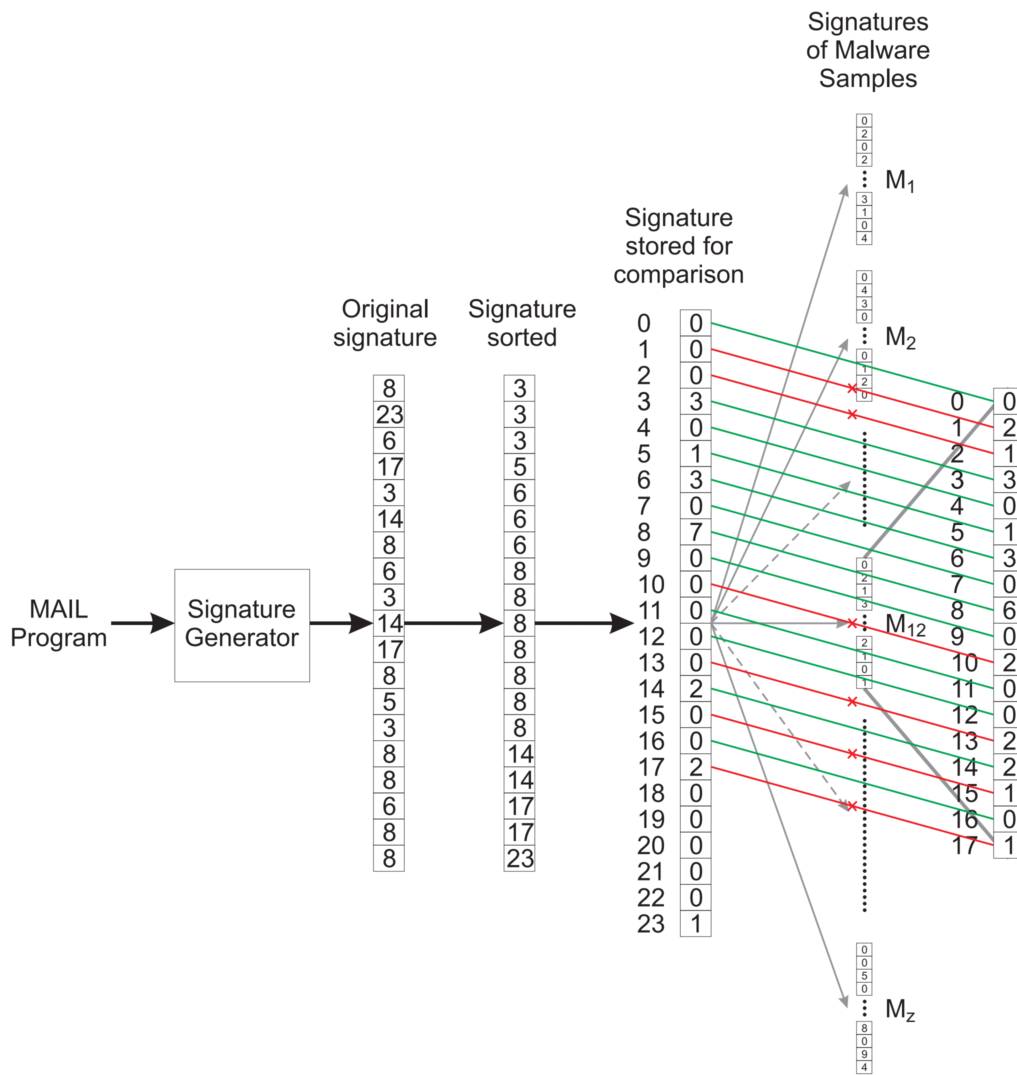


Figure 6.5: Malware detection using MAIL program signatures.

6.3.4 Complexity Analysis

Time complexity of Algorithm 1: There are two outer loops in function *ComputeWeight()*. The run time of the second loop depends on the function *ComputeSWODs()*. Basically this function finds the difference of the data value in two lists and stops with the shorter list. Therefore the time of the function *ComputeSWODs()* depends on the number of either malware or benign samples (whichever is smaller). We compute the time complexity of Algorithm 1 as follows: *time first loop + time second loop = N(nm + nb + nm log nm + nb log nb) + N min(nm, nb)*, and is simplified to $\mathbf{O(\boldsymbol{N}\ \boldsymbol{n} \log \boldsymbol{n})}$, where N is the number of patterns

in MAIL, and n is either the number of malware samples or the number of benign samples whichever is larger.

Time complexity of Algorithm 2: The time for the initialization of *Sig* at line 2 and the two loops in the function *BuildSignature()* depends on the number of statements (NS) in the MAIL program. Including the time for function *Sort()* at line 23 the time complexity of Algorithm 2 is *3NS + NS log NS*, and is simplified to $\mathbf{O(n \log n)}$, where n is the number of statements in the MAIL program.

The average and worst case time complexities of both algorithms depend on the *Sort()* function used in the implementation. The time computed (both average and worst case) above is for a *merge sort* implementation [25].

6.4 Summary

Techniques based on opcode patterns have the potential to be used for real-time malware detection, but have the following issues: (1) The frequencies of opcodes can change by using different compilers, compiler optimizations and operating systems. (2) Obfuscations introduced by polymorphic and metamorphic malware can change the opcode distributions. (3) Selecting too many features (patterns) results in a high detection rate but also increases the runtime and vice versa.

In this chapter we have presented a novel technique named *SWOD-CFWeight* (Sliding Window of Difference and Control Flow Weight) that helps mitigate these effects and provides a solution to these problems. The *SWOD* size can be changed; this property gives anti-malware tool developers the ability to select appropriate parameters to further optimize malware detection. The *CFWeight* feature captures control flow information to an extent that helps detect metamorphic malware in real-time.

Algorithm 2 Computing $CFWeight$ (Control Flow Weight) and Signature of a MAIL program

Precondition: $PatternWeights$ assigned using Algorithm 1

```
1  procedure BuildSignature(MailProgram, PatternWeights, NS)
2      Sig ← 0
3      for s ← 1 to NS do
4          CFWeight ← PatternWeights[MailProgram[s].Pattern]
5          if MailProgram[s].Pattern = ControlPattern then
6              if MailProgram[s].IsJumpOutsideFunction then
7                  CFWeight ← CFWeight + MailProgram[s].DistanceLastStatement + 1
8              else if MailProgram[s].IsBackJump then
9                  CFWeight ← CFWeight + 2 × MailProgram[s].LengthOfJump
10             else
11                 CFWeight ← CFWeight + MailProgram[s].LengthOfJump
12             end if
13             if MailProgram[s].Pattern = IfPattern then
14                 CFWeight ← CFWeight + 2
15             else if MailProgram[s].Pattern = JumpCallPattern then
16                 CFWeight ← CFWeight + 1
17             end if
18         else if MailProgram[s].IsLastStmtOfBlock then
19             CFWeight ← CFWeight + 1
20         end if
21         Sig[s] ← CFWeight
22     end for
23     Sig ← Sort(Sig)
24     Signature ← AssignMem(Sig[Sig.Size] + 1)
25     for s ← 1 to Sig.Size do
26         Signature[Sig[s]] ← Signature[Sig[s]] + 1;
27     end for
28     return Signature
29 end procedure
```

NS is the number of statements in the MAIL program. The function *Sort* sorts the array *Sig* in ascending order. The loop in lines 25 - 27 stores the Signature for easy and fast comparison. e.g: The Signature: 111114447777777777 is stored as: 05003009 i.e: there are 5 1's, 3 4's and 9 7's in the Signature.

Chapter 7

Evaluation, Analysis and Comparison

In this chapter we evaluate the correctness and the efficiency of our proposed metamorphic malware detection techniques, and present results, discussions and analysis of this evaluation. We compare the proposed techniques with other similar malware detection techniques.

7.1 Performance Metrics

We use **n-fold cross validation** to estimate the performance of our techniques. In n-fold cross validation the original sample is divided into n equal size subsamples. One of the samples is used for testing and the remaining $n - 1$ samples are used for training. The cross validation process is then repeated n times with each of the n subsamples used exactly once for validation. The overall performance results are obtained by averaging the results obtained in the n different runs.

Before evaluating the performance of the proposed techniques, we first define the following performance metrics:

True positive (**TP**) is the number of malware programs that are classified as malware. True negative (**TN**) is the number of benign programs that are classified as benign. False positive (**FP**) is the number of benign programs that are classified as malware. False negative (**FN**) is the number of malware programs that are classified as benign.

Precision is the fraction of detected malware samples that are correctly identified.

Accuracy is the fraction of samples, including malware and benign, that are correctly identified as either malware or benign. These two metrics are defined as follows:

$$Precision = \frac{TP}{TP + FP} \qquad Accuracy = \frac{TP + TN}{M + N}$$

where M and N are the total number of malware and benign programs respectively. Now we define the mean maximum precision (**MMP**) and mean maximum accuracy (**MMA**) for n-fold cross-validation as follows:

$$MMP = \frac{1}{n} \sum_{i=1}^{n} Precision_i \tag{7.1}$$

$$MMA = \frac{1}{n} \sum_{i=1}^{n} Accuracy_i \tag{7.2}$$

We also define two other metrics, TP rate and FP rate. The TP rate (**TPR**), also called detection rate (**DR**), indicates the number of samples correctly recognized as malware out of the total malware dataset. The FP rate (**FPR**) metric indicates the number of samples incorrectly recognized as malware out of the total benign dataset. These two metrics are defined as follows:

$$TPR = \frac{TP}{M} \tag{7.3}$$

$$FPR = \frac{FP}{N} \tag{7.4}$$

7.2 Performance of *ACFG*

We carried out an empirical study to analyse the correctness and efficiency of the proposed *ACFG* technique. We present, in this section, the empirical study, obtained results and analysis.

7.2.1 Dataset Based on ACFGs

Graph matching is a computationally intensive task, therefore we selected a subset of the dataset described in Section 6.2.1 for the experiments carried out to analyse the performance of ACFG. The subset contained 3350 sample Windows and Cygwin programs. Out of the 3350 programs, 1020 are metamorphic malware samples. The

dataset distribution based on the number of ACFGs for each sample, and size of the CFG after normalization is shown in Tables 7.1 and 7.2. The normalizations carried out are removal of NOP, junk (some prefixes etc) and other instructions that are not required for malware analysis as listed in [3].

Table 7.1: Dataset distribution based on the number of Annotated Control Flow Graphs (ACFGs) for each program sample

Malware samples (1020)		Benign program samples (2330)	
Number of ACFGs	**Number of Samples**	**Number of ACFGs**	**Number of Samples**
2	250	0 – 50	1277
5 – 32	204	51 – 100	317
33 – 57	222	101 – 199	310
58 – 84	133	201 – 399	282
85 – 133	105	400 – 598	111
140 – 249	94	606 – 987	30
133 – 1272	12	1001 – 1148	3

The dataset contains a variety of programs with ACFGs, ranging from simple to complex for testing. As shown in Table 7.1, the number of ACFGs per malware sample range from 2 to 1272 and the number of ACFGs per benign program range from 0 to 1148. Some of the Windows DLLs (dynamic link libraries) that were used in the experiment do not contain code but only data (cannot be executed) and that is why they have 0 node graphs (ACFGs). The sizes of these ACFGs are shown in Table 7.2. The sizes of the ACFGs of the malware samples range from 1 node to 301 nodes, and the sizes of the ACFGs of the benign programs range from 1 node to 521 nodes.

7.2.2 Empirical Study

In this section, we discuss the experiments, and present the results for precision, and accuracy, to evaluate the performance of ACFG. The experiments were run on the following machine: an Intel Core i5 CPU M 430 (2 Cores) @ 2.27 GHz with 4GB RAM, running Windows 8 Professional.

Table 7.2: Dataset distribution based on the size (number of nodes) for each Annotated Control Flow Graph (ACFG) after normalization and shrinking

Malware samples (1020)		Benign program samples (2330)	
Number of Nodes	**Number of ACFGs**	**Number of Nodes**	**Number of ACFGs**
1 – 10	60284	1 – 10	242240
11 – 20	1652	11 – 20	3466
21 – 39	2177	21 – 39	1086
41 – 69	288	40 – 69	368
70 – 96	207	70 – 99	61
104 – 183	254	100 – 194	60
221 – 301	2	245 – 521	15

Total number of nodes before ACFG shrinking = 4908422
Total number of nodes after ACFG shrinking = 462974 (90.6% reduced)
Runtime on average reduced by a factor of 5 (for smaller datsets) and a factor of 100 (for larger datasets)

To get accurate and the best results from the available dataset, we carried out two experiments: one with a smaller dataset using 10-fold cross validation and the other with a larger dataset using 5-fold cross validation. In the following two sections we present and describe these two experiments.

Experiment with Smaller Dataset Using 10-fold Cross Validation

10-fold cross validation is a compute intensive experiment, so out of 3350 Windows programs we selected 1351 Windows programs. Out of these 1351 programs, 250 are metamorphic malware samples belonging to classes NGVCK_1 and NGVCK_2, and the other 1101 are benign programs. The size of the training set used was 25 samples. We conducted further evaluation by increasing the size of the training set from 25 samples to 125 malware samples (50% of the malware samples). The obtained results are listed in Table 7.3. The DR improved from 94% when the size of the training set is 25 to 99.6% when we used a training dataset of 125 samples.

Table 7.3: Malware detection results for smaller dataset.

Training set size	**DR**	**FPR**	**MMP**	**MMA**	**Real-Time**
25	94%	3.1%	0.86	0.96	✓
125	99.6%	4%	0.85	0.97	✓

Real-time here means the detection is fully automatic and finishes in a reasonable amount of time. On average it took MARD 15.2037 seconds with ACFG shrinking and 40.4288 seconds without ACFG shrinking to complete the malware analysis and detection for 1351 samples including 25 training malware samples. This time excludes time for the training. MARD achieved the same values for all the other performance metrics (DR, FPR, MMP and MMA) with and without ACFG shrinking.

Experiment with Larger Dataset Using 5-fold Cross Validation

The size of the training set used was 204 samples. We conducted further evaluation by increasing the size of the training set from 204 samples to 510 malware samples (50% of the malware samples). The obtained results are listed in Table 7.4. The DR improved from 97% when the size of the training set is 204 to 98.9% when we used a training dataset of 510 samples.

Table 7.4: Malware detection results for larger dataset.

Training set size	**DR**	**FPR**	**MMP**	**MMA**	**Real-Time**
204	97%	4.3%	0.91	0.96	✓
510	98.9%	4.5%	0.91	0.97	✓

Real-time here means the detection is fully automatic and finishes in a reasonable amount of time. On average it took MARD 946.5824 seconds with ACFG shrinking and over 125400 seconds (over 34 hours) without ACFG shrinking to complete the malware analysis and detection for 3350 samples including 204 training malware samples. This time excludes time for the training. Because of the time constraints we did not perform 5-fold cross validation without ACFG shrinking. The time (over 34 hours) reported is just for one run of the experiment without ACFG shrinking.

7.2.3 Comparison with Others

Table 7.5 gives a comparison of *ACFG* with the research efforts of detecting malware (including metamorphic malware) discussed in Chapter 2. None of the prototype systems implemented can be used as a real-time detector. Furthermore a few systems that claim perfect detection rate were validated using small datasets.

Out of all the research efforts, *API-CFG*, *Call-Gram* and *VSA-2* show impressive

results. *API-CFG* does not yet support detection of metamorphic malware, *VSA-2* is using a controlled environment for detection, and *Call-Gram* is not fully automated and their performance overheads are not mentioned in the paper. The dataset used by *VSA-2* is comparatively smaller than the other two.

Table 7.5 reports the best DR results achieved by these detectors. Out of the 9 systems, ACFG clearly shows superior results and, unlike others is fully automatic, supports metamorphic malware detection for 64 bit Windows (PE binaries) and Linux (ELF binaries) platforms and has the potential to be used as a real-time detector.

7.3 Performance of *SWOD-CFWeight*

We carried out an empirical study to analyse the correctness and efficiency of the proposed *SWOD-CFWeight* technique. We present, in this section, the empirical study, obtained results and analysis. We also present a comparison of *SWOD-CFWeight* with *ACFG* using the same dataset and experimental settings as used in Section 7.2.

7.3.1 Empirical Study

First we compute the values of *minsize* and *minratio*, as defined in *Definition 7* and *Definition 9* (Chapter 6) respectively. We computed and used the following values: *minsize = 5* and *minratio = 70* for the dataset used in our experiments.

Eight experiments were conducted using different sizes of the dataset (described in Section 6.2.1). We also change the training set size to provide more variations for testing. The results of different set of experiments were validated using different (value of n) n-fold cross validations. For example for a fair comparison we used the same n-fold cross validation as used in Section 7.2 while comparing *SWOD-CFWeight* with *ACFG*.

7.3.2 Performance Results of *SWOD-CFWeight* and Comparison with *ACFG*

All the eight experiments were run on the following machine: Intel Core i5 CPU M 430 (2 Cores) @ 2.27 GHz with 4GB RAM, operating system Windows 8 professional. The results of all the experiments are listed in Table 7.6.

Table 7.5: Summary and comparison with ACFG of the metamorphic malware analysis and detection systems discussed in Chapter 2

System	Analysis Type	DR	FPR	Data Set Size Benign/Malware	Real-Time	Platform
ACFG	Static	98.9%	4.5%	2330 / 1020	✓	Win & Linux 64
API-CFG [36, 37]	Static	97.53%	1.97%	2140 / 2305	✗	Win 32
Call-Gram [38]	Static	98.4%	2.7%	3234 / 3256	✗	Win 32
Code-Graph [59]	Static	91%	0%	300 / 100	✗	Win 32
DTA [97]	Dynamic	100%	3%	56 / 42	✗	Win XP 64
Model-Checking [86]	Static	100%	1%	8 / 200	✗	Win 32
MSA [94]	Static	91%	52%	150 / 1209	✗	Win 32
VSA-1 [58]	Static	100%	0%	25 / 30	✗	Win 32
VSA-2 [43]	Dynamic	98%	2.9%	385 / 826	✗	Win XP 64

Real-time here means the detection is fully automatic and finishes in a reasonable amount of time. The perfect results should be validated with more samples than tested in the paper. The values for *Opcode-Graph* are not directly mentioned in the paper. We compute these values by picking a threshold of 0.5 for the similarity score in the paper.

Table 7.6: Malware detection results for *SWOD-CFWeight* and comparison with *ACFG*

Technique Used	**Training Set Size**	**Dataset Size Benign/Malware**	**DR**	**FPR**	**MMA**	**Cross Validation**	**Testing Time (seconds)**
ACFG	25	1101 / 250	94%	3.1%	0.96	10-fold	15.2
SWOD-CFWeight	25	1101 / 250	99.08%	0.93%	0.99	10-fold	2.27
ACFG	204	2330 / 1020	97%	4.3%	0.96	5-fold	946.58
SWOD-CFWeight	204	2330 / 1020	94.69%	10.59%	0.91	5-fold	6.13
SWOD-CFWeight	612	2330 / 1020	97.26%	12.44%	0.91	1-fold	8.38
SWOD-CFWeight	204	4168 / 1020	94.69%	9.12%	0.92	5-fold	12.08
SWOD-CFWeight	612	4168 / 1020	97.36%	10.14%	0.92	1-fold	15.89
SWOD-CFWeight	612	4285 / 1020	97.26%	11.29%	0.92	1-fold	15.92

The dataset used in the last row contains additional benign files whose sizes range from 1 MB to 10 MB. All the other datasets contain files (both benign and malware) whose sizes range up to 1 MB. Testing time is the time to check if a file is benign or not and does not include the training time. The time reported is the testing time of all the files in the dataset.

The first four rows in Table 7.6 compare the results of *SWOD-CFWeight* with *ACFG*. For the smaller dataset, *SWOD-CFWeight* shows a much better DR and FPR than *ACFG*, but for the larger dataset, *ACFG* shows a better DR. The main difference between *SWOD-CFWeight* and *ACFG* is the testing time. *ACFG* uses graph matching for malware detection, and in spite of reducing the graph size considerably and hence the time by 2.7 times for the smaller dataset and 100 times for the larger dataset, still the testing time is much larger compared to *SWOD-CFWeight* especially for the larger dataset. As the size of a program (sample) increases the size of the resulting graph (*ACFG*) also increases, and hence the time for graph matching. The testing time increases from the smaller dataset to the larger dataset by 1.7 times for *SWOD-CFWeight* and by 61.3 times for *ACFG*. Keeping in view these results *SWOD-CFWeight* should be used instead of *ACFG* where the time for malware detection is more important as in practical (real-time) anti-malware applications.

The next four rows give more insight into *SWOD-CFWeight*. As expected, the testing time increases linearly with the size of the dataset. The DR decreases by over 4% and the FPR increases by over 10% as the size of the dataset increases. We believe that the reason for this is the size of the *SWOD* used in the experiments. As mentioned before, we randomly selected 25% of the samples from the dataset to compute the size parameters of *SWOD*. This shows that the size of the *SWOD* effects the performance of the malware detector and needs to be computed for every new dataset.

In our future work we will investigate this more and see how we can improve the selection of the samples to compute an optimal size of the *SWOD* for a dataset. For example, dividing benign and malware samples into *classes*, and then selecting an appropriate number of samples from each *class*, can further optimize computation of the size parameters of the *SWOD*. To achieve optimal size values for the SWOD, the frequency of MAIL patterns in each sample must be considered when classifying these samples.

The last row shows how the sizes of the files affects the performance of the detector. The dataset used in the last row contains additional benign files whose sizes range from 1 MB to 10 MB. Comparing the results in the last two rows, the testing time does increase but just by 30 ms, the DR is almost the same and the FPR increases only by 1.15. This shows that the sizes of the files have a very small effect on the results and can be neglected. Therefore we can say that the performance (DR and FPR, and to a certain extent testing time) of the proposed scheme is almost independent of the size

of the files. To verify this claim, it will be validated in future work with additional experiments.

7.3.3 Comparison with Others

Table 7.7 gives a comparison of *SWOD-CFWeight* with the existing opcode-based malware detection approaches discussed in Chapter 2. None of the prototype systems' current implementations can be used as a real-time detector. Most of the techniques, such as *Chi-Squared*, *Opcode-SD*, *Opcode-Graph* and *Opcode-Histogram* show good results, and some of them may have the potential to be used in a real-time detector by improving their implementation. *Opcode-Seqs-Santos* and *Opcode-Seqs-Shabtai* also show impressive results but do not yet support detection of metamorphic malware.

SWOD is a window that represents differences in MAIL Patterns distributions (instead of opcodes) and hence makes the analysis independent of different compilers, ISAs and OSs, compared to existing techniques. *SWOD* size can change, this property gives a user (anti-malware tool developers) the ability to select appropriate parameters for a dataset to further optimize malware detection.

All the systems use the frequency of occurrence of opcodes to capture the execution flow of a program, but fail to capture the control flow of a program that changes the execution flow of the program. *CFWeight* proposed in this paper include this information to an extent that helps detect metamorphic malware.

Table 7.7 reports the best DR result achieved by these detection techniques. Out of the 9 techniques, *SWOD-CFWeight* clearly shows superior results and, unlike others supports metamorphic malware detection for 64 bit Windows (PE binaries) and Linux (ELF binaries) platforms and has the potential to be used in a real-time detector.

Table 7.7: Comparison of *SWOD-CFWeight* with the malware detection techniques discussed in Chapter 2

Technique	**Analysis Type**	**DR**	**FPR**	**Dataset Size Benign/Malware**	**Real Time**	**Platform**
SWOD-CFWeight	Static	99.08%	0.93%	1101 / 250	✓	Win & Linux 64
Opcode-HMM-Wong [96]	Static	~90%	~2%	40 / 200	✗	Win & Linux 32
Chi-Squared [93]	Static	~98%	~2%	40 / 200	✗	Win & Linux 32
Opcode-HMM-Austin [5]	Static	93.5%	0.5%	102 / 77	✗	Win & Linux 32
Opcode-SD [83]	Static	~98%	~0.5%	40 / 800	✗	Linux 32
Opcode-Graph [78]	Static	100%	1%	41 / 200	✗	Win & Linux 32
Opcode-Histogram [75]	Static	100%	0%	40 / 60	✗	Win & Linux 32
Opcode-Seqs-Santos [80]	Static	96%	6%	1000 / 1000	✗	Win 32
Opcode-Seqs-Shabtai [82]	Static	~95%	~0.1%	20416 / 5677	✗	Win 32

Some of the above techniques, need more number of benign samples (more than 40/41) than tested in the papers for further validation. The DR and FPR values for *Opcode-Graph* are not directly mentioned in the paper. We computed these values by picking a threshold of 0.5 from the similarity score in the paper.

Chapter 8

Conclusion and Future Work

8.1 Discussion

There are two basic techniques for malware analysis and detection on the end host. The technique proposed in this thesis is based primarily on static analysis. We discuss the advantages and weaknesses of static analysis versus dynamic analysis and then summariaze the contributions made in this thesis.

8.1.1 Static Analysis

Static analysis is performed on a binary program without executing the program. This technique is mostly used by antimalware software for automatic malware analysis and detection. In general, the complexity of the methods used ranges from simple to deep analysis. Simple static analysis is usually based on string or instruction sequence scanning and matching [44, 90]. More sophisticated methods (deep analysis) rely on control flow analysis [1, 67], value set analysis [6], opcode-based analysis [10] or more complex methods such as model checking [21], etc.

Depending on the method used, static analysis has the potential to be used for real-time malware detection. It captures all the paths taken by an executable program and is easier to automate. There is no possibility of infecting the end point system, because there is no need to run the program to detect the malware. These advantages make static analysis suitable for use in an industrial antimalware software product for malware detection in end point systems.

Beside these advantages, there are some disadvantages of static analysis:

1. It is difficult for a static analysis tool to support different platforms for malware detection.

2. Static analysis for malware detection can be time consuming when conducted manually.

3. When a program cannot be unpacked, then the instructions in a binary file on disk will be different than the instructions at runtime. This makes it difficult for a static analysis tool to detect malware that introduce changes during runtime, such as with metamorphic malware.

8.1.2 Dynamic Analysis

Dynamic analysis is performed on a binary program when the program is executing. This technique is mostly used by antimalware software for manual malware analysis to get a specific signature of a malware to be used later for malware detection. This technique is also used when a program cannot be unpacked during static analysis.

The emphasis on this thesis is more on correct malware detection than malware analysis, so our approach (as mentioned before in Chapter 4) to the problem of unpacking is, that if a program cannot be unpacked with the available unpackers then it is detected as malware. Dynamic analysis can then be performed in a controlled environment for unpacking and classifying the malware. Observing different behaviors of a running program in a dynamic environment may produce a high malware detection rate but is not suitable for real-time detection.

The program automatically unpacks itself when it runs, and with appropriate tools, dynamic analysis is easier to conduct on any platofrm.

Beside these advantages, there are some disadvantages of dynamic analysis:

1. The inability to capture all the executable paths of a program for malware detection, as shown in Chapter 3. One possible solution of this problem, is to force a conditional branch to take multiple paths, which is a non-trivial problem to solve. In general, exploring all the multiple execution paths in a program using this method is time consuming and may render this technique impractical. Recently, Moser et al. [66] explored multiple execution paths for malware analysis in a dynamic-only environment (an emulator), based on specific inputs, and therefore, in general, failed to consider all the execution paths in a program.

2. There is a possibility of infecting the end point system, because there is a need to run the program to detect the malware.

3. The ability of a malware program to detect if it is running in a controlled environment (emulator, virtual machine, etc) and stop to evade analysis makes this technique unreliable for malware detection. Recently a sophisticated banking malware program, *Shylock Trojan* [100], detected execution on virtual machines to evade analysis.

4. Running a program in a controlled environment for automatic analysis may take more time than performing an automatic static analysis on the same program.

These disadvantages make dynamic analysis unsuitable for use in an industrial antimalware software product for malware detection at the end point systems.

8.2 Summary of Contributions

As is clear from the above discussion, out of the two techniques described, static analysis is more suitable for real-time malware detection. For a complete malware analysis and detection system, a combination of these two techniques are used, and is called a hybrid system.

In this thesis, we use static analysis, and mitigate its disadvantages by providing platform independence, automation and optimizations for real-time metamorphic malware detection. In the future we will combine lightweight dynamic analysis with static analysis for in-browser malware analysis and detection.

In this thesis, we have presented a new real-time metamorphic malware detection framework named **MARD**. The framework is based on *MAIL* as a new intermediate language, and implements two novel malware detection techniques, *ACFG* and *SWOD-CFWeight*. We have shown through experimental evaluation, its effectiveness for metamorphic malware analysis and real-time detection. We have also compared *MARD* with other such detection systems. *MARD* with *MAIL* and the two proposed techniques provide: (1) detection automation (2) platform independence and (3) optimizations for real-time performance. In the future, an adequate unpacker will be interfaced with *MARD*.

MAIL provides an abstract representation of an assembly program and hence the ability for a tool to automate malware analysis and detection. By translating binaries compiled for different platforms to *MAIL*, a tool can achieve platform independence. Each *MAIL* statement is annotated with patterns that can be used by a tool to optimize malware analysis and detection.

It is important to note that a program translated to *MAIL* when executed may not produce the same output as the original program. *MAIL* is designed to perform static binary analysis and is not suitable for performing dynamic binary analysis.

The patterns developed, if used with a behavioral signature of a binary program, have the capability to produce useful classifications for malware analysis and detection, as shown by the results in Chapter 7. However if the patterns are used alone, it may not produce the desired results.

The side effects of an assembly instruction are not directly included in the *MAIL* statement. With the presence of various flag registers in the *MAIL* language it is possible for a malware analysis tool to include the side effect(s) of an assembly instruction by generating more statements and updating the affected flag registers.

ACFG can enhance the detection of metamorphic malware and can handle malware with smaller CFGs. We have also optimized the runtime of a malware detector through parallelization and *ACFG* reduction, that makes the comparison (matching with other *ACFGs*) faster, keeping the same accuracy (without *ACFG* reduction) for malware detection, than other techniques that use CFG for malware detection. The annotations in an *ACFG* provide more information, and hence can provide more accuracy than a CFG.

Currently we are carrying out further research into using similar techniques for web malware analysis and detection. Our future work will also consist of strengthening our existing algorithms by investigating and incorporating more powerful pattern recognition techniques.

SWOD mitigates and addresses issues related to the change of the frequency of opcodes in a program, such as the use of different compilers, compiler optimizations and OSs. **CFWeight** includes control flow semantics of a program to an extent that helps reduce the runtime considerably compared to other techniques (that also use control flow semantics) and results in a high detection rate for metamorphic malware detection.

8.3 Future Work

The techniques and mechanisms of infection and malware are moving from PCs to mobile devices, so in the future we will be extending the techniques and tools that are developed in this thesis for web and mobile applications security.

Lightweight dynamic analysis combined with static analysis may be suitable to analyse applications that run inside a web browser for malware detection, such as a combination of Javascript, HTML and CSS (cascading style sheets) that rely on a common web browser to render the application. Such a hybrid system [29] can use an already processed structure (e.g. an abstract syntax tree [1] of the Javascript) by the web browser for malware analysis and detection.

In the future, we will explore different methods to select samples from the dataset to compute an optimal size of *SWOD*. We will also investigate the performance of *SWOD-CFWeight* using much larger files (over 10 MB) and datasets. Keeping in view the ubiquitousness of multicores in the host machines (also called the end points) and to further optimize the runtime, we plan to parallelize the implementation of *SWOD-CFWeight* in *MARD*. We have only tested metamorphic malware as part of our experiments, but we believe that our proposed technique can also be used for the detection of other malware and would like to carry out such experiments in the future.

The MARD framework proposed in this thesis contains parameters and components that can be adjusted, fine tuned and extended to adapt to different environments and systems. The tool developed as part of this thesis to implement the framework is a prototype system. It has the potential to be used in an industrial product after refinement of the two proposed techniques with extensive experiments. *SWOD-CFWeight* is very suitable for use in a real-time detector.

We believe our work in this thesis provides a promising basis for future researchers interested in the area of real-time metamorphic malware analysis and detection.

Appendix A

MAIL Grammar

```
TITLE:       MAIL (Malware Analysis Intermediate Language) Grammar in EBNF
AUTHOR:      Shahid Alam (salam@cs.uvic.ca)
DATED:       March 24, 2013
REVISION:    1.0

DESCRIPTION:

The grammar can be defined by a 3-tuple G = (T, N, P) where
T = set of terminals
N = set of non-terminals
P = set of production rules

This document describes the grammar for MAIL. The grammar uses the EBNF,
syntax where '|' means a choice, ? means optional, * means zero or more
times and + means one or more times. Line Comments start with "--".
Terminator symbol is ";". Terminals are enclosed in single quotes.

--------------------------------------------------------------------------------
--                              PRODUCTION RULES                              --
--------------------------------------------------------------------------------

statements          ::= ( statement* ) ;
statement           ::= assignment_s+
                        | control_s+
                        | condition_s+
                        | function_s+
                        | jump_s+
                        | lib_call_s+
```

```
                        | 'halt'
                        | 'lock' ;


assignment_s        ::= register_s
                        | address_s ;

register_s          ::= register '=' (math_operator)? expr
                        | register '=' (expr)? math_operator expr
                        | register '=' lib_call_s ;

address_s           ::= address '=' (math_operator)? expr
                        | address '=' (expr)? math_operator expr
                        | address '=' lib_call_s ;

control_s           ::= ( 'if' condition_s (jump_s | assignment_s) )
                        ( 'else' (jump_s | assignment_s) )? ;

jump_s              ::= 'jmp' address ;

lib_call_s          ::= letter+ '(' address (, args)* ')' ;

function_s          ::= 'start_function_' digit+ statement 'end_function_' digit+ ;

condition_s         ::= (expr rel_operator expr)+ ;

--------------------------------------------------------------------------------
--                                HELPER RULES                                --
--------------------------------------------------------------------------------

expr                ::= register
                        | address
                        | digit+ ;

register            ::= 'eflags'
                        | 'gr_' digit+
                        | 'fr_' digit+
                        | 'sp'
                        | register_name (':' register_name)? ;

register_name       ::= letter+ ['0' - '9']? ;
```

```
address             ::= '[' digit+ ']'
                        | reg_address
                        | 'UNKNOWN' ;

reg_address         ::= '[' register ( arith_operator (register | digit+) )* ']'
                        | '[' sp '=' sp ('+' | '-') digit+ ']'
                        | '[' register (':' register)? ']' ;

letter              ::= ['a' - 'z'] ['A' - 'Z'] ;

digit               ::= '0x' ['0' - '9'] | ['A' - 'F'] ;

math_operator       ::= arith_operator | log_operator ;

arith_operator      ::= '+' | '-' | '*' | '/' | '%' | '.' ;

log_operator        ::= 'and' | 'or' | 'xor' | ! | '<<' | '>>' ;

args                ::= address (',' address)* ;

rel_operator        ::= '<' | '>' | '<=' | '>=' | '==' | '!=' ;

comment             ::= '--' blank | tab | character | comment* newline ;

character           ::= '!' | '"' | '#' | '$' | '%' | '&' | ''' | '(' | ')'
                        | '[' | '\' | ']' | '^' | '_' | '`' | '{' | '|' | '}'
                        | '*' | '+' | '-' | '/' | ',' | '.' | '~'
                        | ':' | ';' | '<' | '=' | '>' | '?' | '@'
                        | ['0' - '9'] | letter ;

--------------------------------------------------------------------------------
--                                  TOKENS                                    --
--------------------------------------------------------------------------------

WS                  ::= blank | tab | newline ;
COMMENT             ::= '--' blank | tab | character | comment* newline ;
NUM                 ::= digit+ ;
COMMA               ::= ',' ;
COLON               ::= ':' ;
SCOLON              ::= ';' ;
LOP                 ::= 'and' | 'or' | 'xor' | ! | '<<' | '>>' ;
AOP                 ::= '+' | '-' | '*' | '/' | '%' | '.' ;
```

```
ROP                 ::= '<' | '>' | '<=' | '>=' | '==' | '!=' ;
SFUN                :: 'start_function_' digit+ ;
EFUN                :: 'end_function_'  digit+ ;
EQUAL               :: '=' ;
MUL                 ::= '*' ;
DIV                 ::= '/' ;
PLUS                ::= '+' ;
MINUS               ::= '-' ;
LBRKT1              ::= '(' ;
RBRKT1              ::= ')' ;
LBRKT2              ::= '[' ;
RBRKT2              ::= ']' ;
IF                  ::= 'if' ;
ELSE                ::= 'else' ;
UNKNOWN             ::= 'UNKNOWN' ;
```

Appendix B

One of the Reports Generated by MARD

The complete report is more than 20000 lines long, so is not listed here in its entirety. The *translation time* reported here includes the time to read, disassemble, translate to MAIL and building CFG of the program. The *total testing time* reported here is the testing (a 1 means benign and a 0 means a malware) time of all the testing programs in the dataset, and excludes the translating time.

```
--------------------------------------------
|              Printing Report             |
--------------------------------------------

MAX THREADS: 0

|
|   Parsing ..\Virus-Samples\M__MWOR\DC0.5\MWOR_0
|

Parser::BuildCFG: PE signature not found. Now checking for the ELF signature
Parser::Parse: ELF file
Translation time: 0.015 second(s)
Number of blocks: ..\Virus-Samples\M__MWOR\DC0.5\MWOR_0: 660
Number of functions: ..\Virus-Samples\M__MWOR\DC0.5\MWOR_0: 23
|
|   Parsing ..\Virus-Samples\M__MWOR\DC0.5\MWOR_1
|
Parser::BuildCFG: PE signature not found. Now checking for the ELF signature
Parser::Parse: ELF file
Translation time: 0.014 second(s)
Number of blocks: ..\Virus-Samples\M__MWOR\DC0.5\MWOR_1: 625
Number of functions: ..\Virus-Samples\M__MWOR\DC0.5\MWOR_1: 34
```

```
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
Parser::Parse: PE file
Translation time: 0.001 second(s)
Number of blocks: ..\Virus-Samples\M__NGVCK\NGVCK_1.EXE: 93
Number of functions: ..\Virus-Samples\M__NGVCK\NGVCK_1.EXE: 13
|
|   Parsing ..\Virus-Samples\M__NGVCK\NGVCK_10.EXE
|
Parser::Parse: PE file
Translation time: 0.001 second(s)
Number of blocks: ..\Virus-Samples\M__NGVCK\NGVCK_10.EXE: 90
Number of functions: ..\Virus-Samples\M__NGVCK\NGVCK_10.EXE: 13
|
|   Parsing ..\Virus-Samples\M__NGVCK\NGVCK_11.EXE
|
Parser::Parse: PE file
Translation time: 0.002 second(s)
Number of blocks: ..\Virus-Samples\M__NGVCK\NGVCK_11.EXE: 95
Number of functions: ..\Virus-Samples\M__NGVCK\NGVCK_11.EXE: 12
|
|   Parsing ..\Virus-Samples\M__NGVCK\NGVCK_12.EXE
|
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
|
|   Parsing ..\sample\agcore.dll
|
Parser::Parse: PE file
Translation time: 9.378 second(s)
Number of blocks: ..\sample\agcore.dll: 505666
Number of functions: ..\sample\agcore.dll: 32283
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
|
|   Parsing ..\sample\GoogleUpdaterService.exe
|
Parser::Parse: PE file
Translation time: 0.132 second(s)
Number of blocks: ..\sample\GoogleUpdaterService.exe: 13195
Number of functions: ..\sample\GoogleUpdaterService.exe: 1121
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
|
|   Parsing ..\sample\iecleanup.exe
|
Parser::Parse: PE file
Translation time: 0.111 second(s)
Number of blocks: ..\sample\iecleanup.exe: 6759
```

```
Number of functions: ..\sample\iecleanup.exe: 483
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
Parser::Parse: PE file
Translation time: 1.377 second(s)
Number of blocks: ..\sample\gij-3.exe: 85613
Number of functions: ..\sample\gij-3.exe: 8048
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
|
|   Parsing ..\sample\iTunes.exe
|
Parser::Parse: PE file
Translation time: 0.02 second(s)
Number of blocks: ..\sample\iTunes.exe: 2196
Number of functions: ..\sample\iTunes.exe: 203

|
|   Parsing ..\sample\WORDVIEW.EXE
|
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
Parser::Parse: PE file
Translation time: 10.14 second(s)
Number of blocks: ..\sample\WORDVIEW.EXE: 484739
Number of functions: ..\sample\WORDVIEW.EXE: 18929
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
--------------------------------------------
|   Printing Report For Signature Matching  |
--------------------------------------------
                                     Filename                                    Number  Benign
                                                 ..\sample\_bisect.dll       0      1
                                                  ..\sample\_bsddb.dll       1      1
                                                 ..\sample\_Client.dll       2      1
                                              ..\sample\_codecs_cn.dll       3      1
                                              ..\sample\_codecs_hk.dll       4      1
                                         ..\sample\_codecs_iso2022.dll       5      1
                                              ..\sample\_codecs_jp.dll       6      1
                                              ..\sample\_codecs_kr.dll       7      1
                                              ..\sample\_codecs_tw.dll       8      1
                                            ..\sample\_collections.dll       9      1
                                                   ..\sample\_Core.dll      10      1
                                                    ..\sample\_csv.dll      11      1
                                                 ..\sample\_ctypes.dll      12      1
                                            ..\sample\_ctypes_test.dll      13      1
                                                 ..\sample\_curses.dll      14      1
                                           ..\sample\_curses_panel.dll      15      1
                                                  ..\sample\_Delta.dll      16      1
```

..\sample_elementtree.dll	17	1
..\sample_Fs.dll	18	1
..\sample_functools.dll	19	1
..\sample_hashlib.dll	20	1
..\sample_heapq.dll	21	1
..\sample_hotshot.dll	22	1
..\sample_io.dll	23	1
..\sample_ispmres.dll	24	0
..\sample_isusres.dll	25	1
..\sample_json.dll	26	1
..\sample_locale.dll	27	1
..\sample_lsprof.dll	28	1
..\sample_multibytecodec.dll	29	1
..\sample_multiprocessing.dll	30	1
..\sample_Ra.dll	31	1
..\sample_random.dll	32	1
- - - - - - - -		
- - - - - - - -		
- - - - - - - -		
..\sample\Microsoft.Expression.Utility.resources.dll	1734	1
..\sample\Microsoft.Expression.WindowsXamlPlatform.dll	1735	1
..\sample\Microsoft.Expression.WindowsXamlPlatform.resources.dll	1736	1
..\sample\Microsoft.Expression.WpfPlatform.dll	1737	1
..\sample\Microsoft.Expression.WpfPlatform.resources.dll	1738	1
..\sample\Microsoft.Ink.dll	1739	1
..\sample\Microsoft.JScript.dll	1740	1
..\sample\Microsoft.Management.Infrastructure.dll	1741	1
..\sample\Microsoft.Management.OData.dll	1742	1
..\sample\Microsoft.ManagementConsole.dll	1743	1
..\sample\microsoft.msxml.dll	1744	1
..\sample\Microsoft.NetEnterpriseServers.ExceptionMessageBox.dll	1745	1
..\sample\Microsoft.PowerShell.Activities.dll	1746	1
..\sample\Microsoft.PowerShell.Commands.Management.dll	1747	1
..\sample\Microsoft.PowerShell.Commands.Utility.dll	1748	1
..\sample\Microsoft.PowerShell.ConsoleHost.dll	1749	1
..\sample\Microsoft.PowerShell.Core.Activities.dll	1750	1
..\sample\Microsoft.PowerShell.Diagnostics.Activities.dll	1751	1
..\sample\Microsoft.PowerShell.Management.Activities.dll	1752	1
..\sample\Microsoft.PowerShell.ScheduledJob.dll	1753	1
..\sample\Microsoft.PowerShell.Security.Activities.dll	1754	1
..\sample\Microsoft.PowerShell.Security.dll	1755	0
..\sample\Microsoft.PowerShell.Utility.Activities.dll	1756	1
..\sample\Microsoft.PowerShell.Workflow.ServiceCore.dll	1757	1
..\sample\Microsoft.SqlServer.ConnectionInfo.dll	1758	1
..\sample\Microsoft.SqlServer.ConnectionInfoExtended.dll	1759	1
..\sample\Microsoft.SqlServer.Dac.dll	1760	1
..\sample\Microsoft.SqlServer.Dac.resources.dll	1761	0
..\sample\Microsoft.SqlServer.DataStorage.dll	1762	1
..\sample\Microsoft.SqlServer.DlgGrid.dll	1763	1
..\sample\Microsoft.SqlServer.Dmf.Adapters.dll	1764	1
..\sample\Microsoft.SqlServer.Dmf.dll	1765	1
..\sample\Microsoft.SqlServer.DmfSqlClrWrapper.dll	1766	1
..\sample\Microsoft.SqlServer.GridControl.dll	1767	1

..\sample\Microsoft.SqlServer.Management.Collector.dll	1768	1
..\sample\Microsoft.SqlServer.Management.CollectorEnum.dll	1769	1
..\sample\Microsoft.SqlServer.Management.Controls.dll	1770	1
..\sample\Microsoft.SqlServer.Management.HelpViewer.dll	1771	1
..\sample\Microsoft.SqlServer.Management.MultiServerConnection.dll	1772	1
..\sample\Microsoft.SqlServer.Management.Sdk.Sfc.dll	1773	1
..\sample\Microsoft.SqlServer.Management.SDK.SqlStudio.dll	1774	1
..\sample\Microsoft.SqlServer.Management.SqlWizardFramework.dll	1775	1
..\sample\Microsoft.SqlServer.Management.UserSettings.dll	1776	1
..\sample\Microsoft.SqlServer.Management.Utility.dll	1777	1
..\sample\Microsoft.SqlServer.Management.UtilityEnum.dll	1778	1
..\sample\Microsoft.SqlServer.Management.XEvent.dll	1779	1
..\sample\Microsoft.SqlServer.Management.XEventEnum.dll	1780	1
..\sample\Microsoft.SqlServer.PolicyEnum.dll	1781	1
..\sample\Microsoft.SqlServer.RegSvrEnum.dll	1782	1
..\sample\Microsoft.SqlServer.ServiceBrokerEnum.dll	1783	1
..\sample\Microsoft.SqlServer.SmoExtended.dll	1784	1
..\sample\Microsoft.SqlServer.SqlWmiManagement.dll	1785	1
..\sample\Microsoft.SqlServer.Sqm.dll	1786	1

- - - - - - - -
- - - - - - - -
- - - - - - - -

..\sample\wcstoreproxy.dll	4010	1
..\sample\wcsync.dll	4011	1
..\sample\WDE.dll	4012	1
..\sample\WDExpress.exe	4013	1
..\sample\WDExpressDesc.dll	4014	1
..\sample\WDExpressMnu.dll	4015	1
..\sample\WDExpressmui.dll	4016	1
..\sample\weave.exe	4017	1
..\sample\webapprt-stub.exe	4018	1
..\sample\webapp-uninstaller.exe	4019	0
..\sample\webdirprj.dll	4020	1
..\sample\webdirprjui.dll	4021	1
..\sample\WebKit2WebProcess.exe	4022	1
..\sample\WebKitQuartzCoreAdditions.dll	4023	1
..\sample\WFC.exe	4024	1
..\sample\wget.exe	4025	1
..\sample\where.exe	4026	1
..\sample\whets32MP.exe	4027	1
..\sample\whets32SSE.exe	4028	1
..\sample\whets64MP.exe	4029	1
..\sample\whets8thread32.exe	4030	1
..\sample\whets8Thread64.exe	4031	1
..\sample\which.exe	4032	1
..\sample\who.exe	4033	1
..\sample\whoami.exe	4034	1
..\sample\WiLogUtl.exe	4035	1
..\sample\wimax.dll	4036	1
..\sample\wimaxasncp.dll	4037	1
..\sample\wimaxmacphy.dll	4038	1
..\sample\wimserv.exe	4039	1
..\sample\Win32.dll	4040	1

..\sample\Win32BinaryFile.dll	4041	1
..\sample\Win32Site.dll	4042	1
..\sample\winamp.exe	4043	1
..\sample\winampa.exe	4044	1
..\sample\windmc.exe	4045	1
..\sample\WindowsBase.dll	4046	1
..\sample\WindowsFormsIntegration.Design.dll	4047	1
..\sample\WindowsFormsIntegration.dll	4048	0
..\sample\WindowsFormsIntegration.Package.dll	4049	1
..\sample\WindowsFormsIntegration.PackageUI.dll	4050	1
..\sample\WindowsLive.Client.dll	4051	1
..\sample\WindowsLive.Writer.Api.dll	4052	1
..\sample\WindowsLive.Writer.ApplicationFramework.dll	4053	1
..\sample\WindowsLive.Writer.BlogClient.dll	4054	1
..\sample\WindowsLive.Writer.BrowserControl.dll	4055	1
..\sample\WindowsLive.Writer.Controls.dll	4056	1
..\sample\WindowsLive.Writer.Extensibility.dll	4057	1
..\sample\WindowsLive.Writer.FileDestinations.dll	4058	1
..\sample\WindowsLive.Writer.HtmlEditor.dll	4059	1
..\sample\WindowsLive.Writer.HtmlParser.dll	4060	1
..\sample\WindowsLive.Writer.Instrumentation.dll	4061	1
..\sample\WindowsLive.Writer.Interop.dll	4062	1
..\sample\WindowsLive.Writer.Interop.Mshtml.dll	4063	1
..\sample\WindowsLive.Writer.Interop.SHDocVw.dll	4064	1
- - - - - - - -		
- - - - - - - -		
- - - - - - - -		
..\Virus-Samples\M__MWOR\DC1.0\MWOR_30	4409	0
..\Virus-Samples\M__MWOR\DC1.0\MWOR_31	4410	0
..\Virus-Samples\M__MWOR\DC1.0\MWOR_32	4411	0
..\Virus-Samples\M__MWOR\DC1.0\MWOR_33	4412	0
..\Virus-Samples\M__MWOR\DC1.0\MWOR_34	4413	0
..\Virus-Samples\M__MWOR\DC1.0\MWOR_35	4414	0
..\Virus-Samples\M__MWOR\DC1.0\MWOR_36	4415	0
..\Virus-Samples\M__MWOR\DC1.0\MWOR_37	4416	0
..\Virus-Samples\M__MWOR\DC1.0\MWOR_38	4417	1
..\Virus-Samples\M__MWOR\DC1.0\MWOR_39	4418	0
..\Virus-Samples\M__MWOR\DC1.0\MWOR_4	4419	0
..\Virus-Samples\M__MWOR\DC1.0\MWOR_40	4420	0
..\Virus-Samples\M__MWOR\DC1.0\MWOR_41	4421	0
..\Virus-Samples\M__MWOR\DC1.0\MWOR_42	4422	0
..\Virus-Samples\M__MWOR\DC1.0\MWOR_43	4423	0
..\Virus-Samples\M__MWOR\DC1.0\MWOR_44	4424	0
..\Virus-Samples\M__MWOR\DC1.0\MWOR_45	4425	0
..\Virus-Samples\M__MWOR\DC1.0\MWOR_46	4426	0
..\Virus-Samples\M__MWOR\DC1.0\MWOR_47	4427	0
..\Virus-Samples\M__MWOR\DC1.0\MWOR_48	4428	0
..\Virus-Samples\M__MWOR\DC1.0\MWOR_49	4429	0
..\Virus-Samples\M__MWOR\DC1.0\MWOR_5	4430	0
..\Virus-Samples\M__MWOR\DC1.0\MWOR_50	4431	0
..\Virus-Samples\M__MWOR\DC1.0\MWOR_51	4432	0
..\Virus-Samples\M__MWOR\DC1.0\MWOR_52	4433	0
..\Virus-Samples\M__MWOR\DC1.0\MWOR_53	4434	0

```
                                   ..\Virus-Samples\M__MWOR\DC1.0\MWOR_54    4435    0
                                   ..\Virus-Samples\M__MWOR\DC1.0\MWOR_55    4436    0
                                   ..\Virus-Samples\M__MWOR\DC1.0\MWOR_56    4437    0
                                   ..\Virus-Samples\M__MWOR\DC1.0\MWOR_57    4438    0
                                   ..\Virus-Samples\M__MWOR\DC1.0\MWOR_58    4439    0
                                   ..\Virus-Samples\M__MWOR\DC1.0\MWOR_59    4440    0
                                    ..\Virus-Samples\M__MWOR\DC1.0\MWOR_6    4441    0
                                   ..\Virus-Samples\M__MWOR\DC1.0\MWOR_60    4442    0
- - - - - - - -
- - - - - - - -
- - - - - - - -
                                   ..\Virus-Samples\M__NGVCK\ngvck190.EXE    5244    0
                                   ..\Virus-Samples\M__NGVCK\ngvck191.EXE    5245    0
                                   ..\Virus-Samples\M__NGVCK\ngvck192.EXE    5246    0
                                   ..\Virus-Samples\M__NGVCK\ngvck193.EXE    5247    0
                                   ..\Virus-Samples\M__NGVCK\ngvck194.EXE    5248    0
                                   ..\Virus-Samples\M__NGVCK\ngvck195.EXE    5249    0
                                   ..\Virus-Samples\M__NGVCK\ngvck196.EXE    5250    0
                                   ..\Virus-Samples\M__NGVCK\ngvck197.EXE    5251    0
                                   ..\Virus-Samples\M__NGVCK\ngvck198.EXE    5252    0
                                   ..\Virus-Samples\M__NGVCK\ngvck199.EXE    5253    0
                                   ..\Virus-Samples\M__NGVCK\ngvck200.EXE    5254    0
                                          ..\Virus-Samples\M__G2\G2_1.EXE    5255    0
                                         ..\Virus-Samples\M__G2\G2_10.EXE    5256    0
                                         ..\Virus-Samples\M__G2\G2_11.EXE    5257    0
                                         ..\Virus-Samples\M__G2\G2_12.EXE    5258    0
                                         ..\Virus-Samples\M__G2\G2_13.EXE    5259    0
                                         ..\Virus-Samples\M__G2\G2_14.EXE    5260    0
                                         ..\Virus-Samples\M__G2\G2_15.EXE    5261    0
- - - - - - - -
- - - - - - - -
- - - - - - - -
                                         ..\Virus-Samples\M__G2\G2_40.EXE    5289    0
                                         ..\Virus-Samples\M__G2\G2_41.EXE    5290    0
                                         ..\Virus-Samples\M__G2\G2_42.EXE    5291    0
                                         ..\Virus-Samples\M__G2\G2_43.EXE    5292    0
                                         ..\Virus-Samples\M__G2\G2_44.EXE    5293    0
                                         ..\Virus-Samples\M__G2\G2_45.EXE    5294    0
                                         ..\Virus-Samples\M__G2\G2_46.EXE    5295    0
                                         ..\Virus-Samples\M__G2\G2_47.EXE    5296    0
                                         ..\Virus-Samples\M__G2\G2_48.EXE    5297    0
                                         ..\Virus-Samples\M__G2\G2_49.EXE    5298    0
                                          ..\Virus-Samples\M__G2\G2_5.EXE    5299    0
                                         ..\Virus-Samples\M__G2\G2_50.EXE    5300    0
                                          ..\Virus-Samples\M__G2\G2_6.EXE    5301    0
                                          ..\Virus-Samples\M__G2\G2_7.EXE    5302    0
                                          ..\Virus-Samples\M__G2\G2_8.EXE    5303    0
                                          ..\Virus-Samples\M__G2\G2_9.EXE    5304    0

Total testing time: 15.915 second(s)
```

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[2] Shahid Alam. *Examples of CFGs Before and After Shrinking*. `http://www.cs.uvic.ca/~salam/PhD/cfgs.html`, 2013.

[3] Shahid Alam. *MAIL: Malware Analysis Intermediate Language*. `http://www.cs.uvic.ca/~salam/PhD/TR-MAIL.pdf`, 2013.

[4] S. S. Anju, P. Harmya, Noopa Jagadeesh, and R. Darsana. Malware Detection Using Assembly Code and Control Flow Graph Optimization. In *A2CWiC, 2010*, pages 65:1 – 65:4, New York, NY, USA, 2010. ACM.

[5] Thomas H. Austin, Eric Filiol, Sebastien Josse, and Mark Stamp. Exploring Hidden Markov Models for Virus Analysis: A Semantic Approach. In *System Sciences (HICSS), 2013 46th Hawaii International Conference on*, pages 5039–5048, Jan 2013.

[6] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. *WYSINWYX: What You See Is Not What You eXecute*. PhD thesis, University of Wisconsin, 2005.

[7] Burak Bayoglu and Ibrahim Sogukpinar. Graph Based Signature Classes for Detecting Polymorphic Worms via Content Analysis. *Comput. Netw.*, 56(2): 832–844, February 2012. ISSN 1389-1286.

[8] Donabelle Baysa, RichardM. Low, and Mark Stamp. Structural entropy and metamorphic malware. *Journal of Computer Virology and Hacking Techniques*, 9(4):179–192, 2013.

[9] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT'08*, pages 72 – 81, New York, NY, USA, 2008. ACM.

[10] Daniel Bilar. Opcodes As Predictor for Malware. *International Journal of Electronic Security and Digital Forensics*, 1(2):156–168, January 2007.

[11] Jean-Marie Borello and Ludovic M. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology*, 4(3):211–220, 2008. ISSN 1772-9890.

[12] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting Self-Mutating Malware Using Control-Flow Graph Matching. In *DIMVA, 2006*, pages 129 – 143, Berlin, Heidelberg, 2006. Springer-Verlag.

[13] PeterA. Buhr and Roy Krischer. Bound Exceptions in Object-Oriented Programming. In *Advanced Topics in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*, pages 1–21. Springer Berlin Heidelberg, 2006.

[14] Gerardo Canfora, AntonioNiccolo Iannaccone, and CorradoAaron Visaggio. Static analysis for the detection of metamorphic computer viruses using repeated-instructions counting heuristics. *Journal of Computer Virology and Hacking Techniques*, 10(1):11–27, 2014.

[15] S. Cesare and Yang Xiang. A fast flowgraph based classification system for packed and polymorphic malware on the endhost. In *AINA, 2010*, pages 721 – 728, April 2010.

[16] S. Cesare and Yang Xiang. Malware Variant Detection Using Similarity Search over Sets of Control Flow Graphs. In *TrustCom, 2011*, pages 181 – 189, November 2011.

[17] S. Cesare and Yang Xiang. Wire – A Formal Intermediate Language for Binary Analysis. In *TrustCom, 2012*, pages 515 – 524, June 2012.

[18] Silvio Cesare and Yang Xiang. Classification of Malware Using Structured Control Flow. In *AusPDC, 2010*, pages 61 – 70, Darlinghurst, Australia, Australia, 2010. Australian Computer Society, Inc.

[19] Silvio Cesare, Yang Xiang, and Wanlei Zhou. Malwise – An Effective and Efficient Classification System for Packed and Polymorphic Malware. *IEEE Transactions on Computers*, 62(6):1193–1206, 2013. ISSN 0018-9340.

[20] M. Christodorescu, S. Jha, S.A. Seshia, D. Song, and R.E. Bryant. Semantics-Aware Malware Detection. In *Security and Privacy, 2005 IEEE Symposium on*, pages 32 – 46, May 2005.

[21] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[22] F. Cohen. Computer Viruses: Theory and Experiments. *Computer Security*, 6 (1):22 – 35, Feburary 1987.

[23] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 184–196, New York, NY, USA, 1998. ACM. ISBN 0-89791-979-3.

[24] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[25] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[26] IBM Corporation. *POWER ISA Version 2.03*, September 2006.

[27] Intel Corporation. *Intel ® 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z*, January 2013.

[28] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, volume 2, pages 119–129. IEEE Computer Society, 2000.

[29] Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. ZOZZLE: Fast and Precise In-browser JavaScript Malware Detection. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association.

[30] Gil Dabah. *Powerful Disassembler Library For x86/AMD64*. `http://code.google.com/p/distorm`, Last accessed: August 16, 2014.

[31] C. De Dinechin. C++ Exception Handling. *Concurrency, IEEE*, 8(4):72 – 79, 2000.

[32] Sayali Deshpande, Younghee Park, and Mark Stamp. Eigenvalue analysis for metamorphic detection. *Journal of Computer Virology and Hacking Techniques*, 10(1):53–65, 2014.

[33] International Standard Organization document reference ISO/IEC. *Information Technology - Syntactic Metalanguage - Extended Backus-Naur Form*, 1996.

[34] Thomas Dullien and Sebastian Porst. REIL : A Platform-Independent Intermediate Representation of Disassembled Code for Static Code Analysis. *In Proceeding of CanSecWest*, 2009.

[35] Chris Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, San Francisco, CA, USA, 2008.

[36] Mojtaba Eskandari and Sattar Hashemi. A Graph Mining Approach for Detecting Unknown Malwares. *Journal of Visual Languages and Computing*, 23 (3):154 – 162, June 2012.

[37] Mojtaba Eskandari and Sattar Hashemi. ECFGM: Enriched Control Flow Graph Miner for Unknown Vicious Infected Code Detection. *Journal in Computer Virology*, 8(3):99 – 108, August 2012.

[38] Parvez Faruki, Vijay Laxmi, M. S. Gaur, and P. Vinod. Mining Control Flow Graph as API Call-Grams to Detect Portable Executable Malware. In *Security of Information and Networks*, SIN '12, New York, NY, USA, 2012. ACM SIGSAC.

[39] Eric Filiol and Sbastien Josse. A Statistical Model for Undecidable Viral Detection. *Journal in Computer Virology*, 3:65 – 74, 2007.

[40] Halvar Flake. Structural Comparison of Executable Objects. In Ulrich Flegel and Michael Meier 0001, editors, *DIMVA*, volume 46 of *LNI*, pages 161–173. GI, 2004.

[41] G2. *Second Generation Virus Generator*. `http://vxheaven.org/vx.php?id=tg00`, Last accessed: August 16, 2014.

[42] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[43] Mahboobe Ghiasi, Ashkan Sami, and Zahra Salehi. Dynamic Malware Detection Using Registers Values Set Analysis. In *Information Security and Cryptology*, pages 54 – 59, 2012.

[44] Kent Griffin, Scott Schneider, Xin Hu, and Tzi-cker Chiueh. Automatic Generation of String Signatures for Malware Detection. In Engin Kirda, Somesh Jha, and Davide Balzarotti, editors, *Recent Advances in Intrusion Detection*, volume 5758 of *Lecture Notes in Computer Science*, pages 101–120. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-04341-3.

[45] Jonathan L. Gross and Jay Yellen. *Graph Theory and Its Applications, Second Edition (Discrete Mathematics and Its Applications)*. Chapman & Hall/CRC, 2005.

[46] Haoran Guo, Jianmin Pang, Yichi Zhang, Feng Yue, and Rongcai Zhao. Hero: A novel malware detection framework based on binary translation. In *ICIS, 2010*, volume 1, pages 411 – 415, oct. 2010.

[47] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[48] Lucas Chi Kwong Hui. Color Set Size Problem with Application to String Matching. In *Proceedings of the Third Annual Symposium on Combinatorial Pattern Matching*, CPM '92, pages 230–243, London, UK, UK, 1992. Springer-Verlag. ISBN 3-540-56024-6.

[49] Nwokedi Idika and Aditya P. Mathur. A Survey of Malware Detection Techniques. *Purdue University*, 2007.

[50] ITU. The World in 2013: ICT Facts and Figures. © *ITU*, 2013.

[51] Thomas Jakobsen. A fast method for cryptanalysis of substitution ciphers. *Cryptologia*, 19(3):265–274, 1995.

[52] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28:11–21, 1972.

[53] Johannes Kinder and Helmut Veith. Jakstab: A Static Analysis Platform for Binaries. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 423 – 427. Springer Berlin Heidelberg, 2008.

[54] Engin Kirda, Christopher Kruegel, Greg Banks, Giovanni Vigna, and Richard A. Kemmerer. Behavior-based Spyware Detection. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[55] J.B. Kruskal. Multidimensional Scaling by Optimizing Goodness of fit to a Nonmetric Hypothesis. *Psychometrika*, 29:1 – 27, 1964.

[56] Nikolay Kuzurin, Alexander Shokurov, Nikolay Varnovsky, and Vladimir Zakharov. On the Concept of Software Obfuscation in Computer Security. In *Proceedings of the 10th International Conference on Information Security*, ISC'07, pages 281–298, Berlin, Heidelberg, 2007. Springer-Verlag.

[57] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO, 2004*, Washington, DC, USA, 2004. IEEE Computer Society.

[58] F. Leder, B. Steinbock, and P. Martini. Classification and Detection of Metamorphic Malware Using Value Set Analysis. In *MALWARE, 2009*, pages 39 – 46, oct. 2009.

[59] Jusuk Lee, Kyoochang Jeong, and Heejo Lee. Detecting Metamorphic Malwares Using Code Graphs. In *SAC, 2010*, pages 1970 – 1977, New York, NY, USA, 2010. ACM.

[60] Da Lin and Mark Stamp. Hunting for undetectable metamorphic viruses. *Journal in Computer Virology*, 7(3):201–214, 2011. ISSN 1772-9890.

[61] Cullen Linn and Saumya Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *ACM CCS*, pages 290 – 299, New York, NY, USA, 2003. ACM.

[62] David M. Chess and Steve R. White. An Undetectable Computer Virus. *Virus Bulletin Conference*, September 2000.

[63] Sudarshan Madenur Sridhara and Mark Stamp. Metamorphic worm that carries its own morphing engine. *Journal of Computer Virology and Hacking Techniques*, 9(2):49–58, 2013.

[64] Timothy G. Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, and Saurabh Dighe. The 48-core SCC Processor: the Programmer's View. In *SC'10*, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[65] Gary McGraw and Greg Morrisett. Attacking Malicious Code: A Report to the Infosec Research Council. *IEEE Softw.*, 17(5):33 – 41, September 2000. ISSN 0740-7459.

[66] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 231–245, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2848-1.

[67] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[68] Nicholas Nethercote and Julian Seward. Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.*, 42(6):89 – 100, June 2007.

[69] NGVCK. *Next Generation Virus Construction Kit*. `http://vxheaven.org/vx.php?id=tn02`, Last accessed: August 16, 2014.

[70] Philip OKane, Sakir Sezer, and Kieran McLaughlin. Obfuscation: The Hidden Malware. *IEEE Security and Privacy*, 9(5):41 – 47, September 2011.

[71] Vinod P., Vijay Laxmi, Manoj Singh Gaur, GVSS Phani Kumar, and Yadvendra S. Chundawat. Static CFG Analyzer for Metamorphic Malware Code. In

Proceedings of the 2Nd International Conference on Security of Information and Networks, SIN '09, pages 225–228, New York, NY, USA, 2009. ACM SIGSAC.

[72] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053 – 1058, December 1972.

[73] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011. ISBN 978-0123838728.

[74] ARM Holdings plc. *ARM ® Architecture Reference Manual ARMv7-A and ARMv7-R edition*, January 2012.

[75] B.B. Rad, M. Masrom, and S. Ibrahim. Opcodes Histogram for Classifying Metamorphic Portable Executables Malware. In *ICEEE*, pages 209 – 213, September 2012.

[76] Thomas Raschke. The New Security Challenge: Endpoints. © *International Data Corporation*, 2005.

[77] Stephen Robertson. Understanding inverse document frequency: On theoretical arguments for idf. *Journal of Documentation*, 60, 2004.

[78] Neha Runwal, Richard M. Low, and Mark Stamp. Opcode Graph Similarity and Metamorphic Detection. *Journal in Computer Virology*, 8(1-2):37 – 52, May 2012.

[79] A.H.J. Sale. *The Implementation of Case Statements in PASCAL*. Technical Report. Department of Information Science, University of Tasmania, 1979.

[80] Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo G. Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, 231(0):64 – 82, 2013. Data Mining for Information Security.

[81] Alexander Sepp, Bogdan Mihaila, and Axel Simon. Precise Static Analysis of Binaries by Extracting Relational Information. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, WCRE '11, pages 357 – 366, Washington, DC, USA, 2011. IEEE Computer Society.

[82] Asaf Shabtai, Robert Moskovitch, Clint Feher, Shlomi Dolev, and Yuval Elovici. Detecting unknown malicious code by applying classification techniques on opcode patterns. *Security Informatics*, 1(1):1–22, 2012.

[83] Gayathri Shanmugam, Richard M Low, and Mark Stamp. Simple substitution distance and metamorphic detection. *Journal of Computer Virology and Hacking Techniques*, 9(3):159–170, 2013.

[84] P.K. Singh and A. Lakhotia. Static Verification of Worm and Virus Behavior in Binary Executables Using Model Checking. In *Information Assurance Workshop, 2003. IEEE Systems, Man and Cybernetics Society*, pages 298 – 300, june 2003.

[85] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *ICISS, 2008*, pages 1 – 25, Berlin, Heidelberg, 2008. Springer-Verlag.

[86] Fu Song and Tayssir Touili. Efficient Malware Detection Using Model-Checking. In Dimitra Giannakopoulou and Dominique Mry, editors, *FM: Formal Methods*, volume 7436 of *Lecture Notes in Computer Science*, pages 418–433. Springer Berlin Heidelberg, 2012.

[87] Fu Song and Tayssir Touili. Pushdown model checking for malware detection. In Cormac Flanagan and Barbara Knig, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7214 of *Lecture Notes in Computer Science*, pages 110–125. Springer Berlin Heidelberg, 2012.

[88] Corporation Symantec. Norton Cybercrime Report. ©*Symantec Corporation (http://www.symantec.com)*, August 2012.

[89] Corporation Symantec. Internet Security Threat Report - 2011 Trends. © *Symantec Corporation*, 17, April 2012.

[90] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[91] GCC Team. *GCC: The GNU Compiler Collection*. http://gcc.gnu.org, 2013.

[92] Zynamics team at Google. *BinNavi: Binary Code Reverse Engineering Tool © Google Inc.* http://www.zynamics.com/binnavi.html, Last accessed: August 16, 2014.

[93] AnnieH. Toderici and Mark Stamp. Chi-squared Distance and Metamorphic Virus Detection. *Journal in Computer Virology*, pages 1 – 14, 2013.

[94] P. Vinod, V. Laxmi, M.S. Gaur, and G. Chauhan. MOMENTUM: Metamorphic Malware Exploration Techniques Using MSA Signatures. In *IIT*, pages 232 – 237, March 2012.

[95] Eric W. Weisstein. Chi-Squared Test. In *MathWorld - A Wolfram Web Resource*. Wolfram Research Inc., `http://mathworld.wolfram.com/Chi-SquaredTest.html`, Last accessed: August 16, 2014.

[96] Wing Wong and Mark Stamp. Hunting for Metamorphic Engines. *Journal in Computer Virology*, 2:211–229, 2006.

[97] Heng Yin and Dawn Song. Privacy-Breaching Behavior Analysis. In *Automatic Malware Analysis*, SpringerBriefs in Computer Science, pages 27–42. Springer New York, 2013.

[98] Qinghua Zhang. *Polymorphic and Metamorphic Malware Detection*. PhD thesis, North Carolina State University, 2008.

[99] Qinghua Zhang and D.S. Reeves. MetaAware: Identifying Metamorphic Malware. In *ACSAC, 2007*, pages 411 – 420, December 2007.

[100] Zeljka Zorz. *New wave of Shylock Trojan targets bank customers*. `http://www.net-security.org/malware_news.php?id=2592`, Last accessed: August 16, 2014.