Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

A framework for metamorphic malware analysis and real-time detection

Shahid Alam [a,*], *R.Nigel Horspool* [a], *Issa Traore* [b,**], *Ibrahim Sogukpinar* [c]

[a] *Department of Computer Science, University of Victoria, Victoria, BC, V8P5C2, Canada*
[b] *Department of Electrical and Computer Engineering, University of Victoria, Victoria, BC, V8P5C2, Canada*
[c] *Department of Computer Engineering, Gebze Institute of Technology, Gebze, Kocaeli, 41400, Turkey*

ARTICLE INFO

Article history:
Received 28 April 2014
Received in revised form
6 October 2014
Accepted 19 October 2014
Available online 1 November 2014

Keywords:
End point security
Malware analysis
Malware detection
Metamorphic malware
Window of difference
Control flow analysis
Heuristics
Data mining

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 paper presents a new framework named *MARD* for Metamorphic Malware Analysis and Real-Time 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%. 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.

* *Corresponding author.*
** *Corresponding author.* Tel.: +1 250 721 8697.
E-mail addresses: salam@cs.uvic.ca (S. Alam), nigelh@cs.uvic.ca (R.Nigel Horspool), itraore@ece.uvic.ca (I. Traore), ispinar@bilmuh.gyte.edu.tr (I. Sogukpinar).
http://dx.doi.org/10.1016/j.cose.2014.10.011
0167-4048/

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 (ITU, 2013) 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 about the impacts of cybercrime reports that worldwide losses due to malware attacks and phishing between July 2011 and July 2012 were $110 billion (Symantec, 2012b). According to the 2011 Symantec Internet security threat report (Symantec, 2012a) there was an 81% increase in the malware attacks over 2010, and 403 million new malware were 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 percent in 2012. With these increases and the anticipated future increases, these end points pose a new security challenge (Raschke, 2005) to the security professionals and researchers in industry and in academia, to devise new methods and techniques for malware detection and protection.

1.1. *Hidden malware*

Binary code obfuscation is a stealth technique that provides the capability to a malware for evading detection. 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 (OKane et al., 2011; Linn and Debray, 2003; Kuzurin et al., 2007; Collberg et al., 1998; Borello and Me, 2008). 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 (OKane et al., 2011): 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 (OKane et al., 2011) 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 same sequence of opcodes in the memory. This is also called *dynamic code obfuscation*. There are two kinds of metamorphic malware defined in (OKane et al., 2011) based on the channel of communication used: *Closed-world malware*, that does not rely on external communication and can generate the newly mutated code using either a binary transformer or a metalanguage, and *Open-world malware*, that can communicate with other sites on the Internet and update itself with new features.

1.2. *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 systems 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 paper our emphasis is on the second option.

As is clear from the above discussion out of the three malware groups mentioned, metamorphic malware are getting more complex and pose a special threat and new challenges to the end point security. None of the current techniques (Eskandari and Hashemi, 2012a, b; Faruki et al., 2012; Toderici and Stamp, 2013; Lee et al., 2010; Yin and Song, 2013; Rad et al., 2012; Song and Touili, 2012a; Vinod et al., 2012; Runwal et al., 2012; Leder et al., 2009; Ghiasi et al., 2012; Canfora et al., 2014; Austin et al., 2013; Shanmugam et al., 2013; Wong and Stamp, 2006) provide real-time detection for metamorphic malware. The number of new malware are increasing significantly and we need to automate the process of malware analysis and detection.

1.3. *Unknown metamorphic malware*

It is difficult to write a new metamorphic malware (Szor, 2005) 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 using pattern recognition techniques. On this assumption and motivation, we develop in this paper novel techniques to build a

behavioral signature and detect known and unknown metamorphic malware in real-time.

1.4. Contributions

Following are the contributions of this paper:

1. We propose a novel technique named Annotated Control Flow Graph (**ACFG**) that reduces the effects of obfuscations and provides efficient metamorphic malware detection. ACFG is built by annotating a control flow graph (CFG) of a binary program and is used for graph and pattern matching to analyze 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 and can handle malware with smaller CFGs, compared to other such techniques, without compromising the accuracy; (c) contains more information and hence provides more accuracy than a CFG.
2. We propose a novel technique named Sliding Window of Difference and Control Flow Weight (**SWOD-CFWeight**) that reduces the effects of obfuscations and provides real-time metamrophic malware detection. (a) SWOD is a window that represents differences in MAIL (Malware Analysis Intermediate Language) patterns distributions. MAIL patterns are a high level representation of opcodes and can be used in a similar manner (Alam et al., 2013), and hence make the analysis independent of the choice of compiler and the platform, compared to existing techniques that use opcodes for malware detection. (b) Unlike the current techniques that use opcodes for metamorphic malware detection, CFWeight captures the control flow semantics of a program to an extent that helps detect metamorphic malware in real-time.
3. We present a new framework named **MARD**[1] for Metamorphic Malware Analysis and Real-Time Detection. MARD uses the language MAIL and implements the above two proposed techniques, and enables detection automation, platform independence, and optimizations for real-time performance.

We conduct experimental evaluation of the proposed techniques, using an existing dataset of 5305 metamorphic malware and benign samples, a variety of samples based in terms of file size, number of ACFGs per sample and ACFG size.

The remainder of the paper is organized as follows. Section 2 describes the previous research efforts for detecting metamorphic malware. Section 3 describes MARD. Sections 4 and 5 describe the two new proposed techniques. In Section 6 we evaluate the performance of the two proposed techniques using an existing dataset and compare them with other techniques. Section 7 concludes the paper.

[1] The framework presented in this paper is a major extension of our previous work proposed in the conference paper (Alam et al., 2014).

2. Related works

This Section discusses the previous research efforts for detecting malware. We cover only recent academic research efforts that claim to detect or intend to extend their approaches to detect metamorphic malware. We divide these research efforts into three groups based on the type of analysis performed for malware detection: control flow analysis, information flow analysis, and opcode-based analysis.

2.1. Control flow analysis

Song and Touili (2012a) presented a method that uses model-checking to detect metamorphic malware. A program is modeled and malware's behaviors are specified using a formal language. Therefore the behavior of a program can be checked without executing the program. They build a CFG (Aho et al., 2006) of a binary program, which contains information about the register and the memory location values at each control point of the program.

Model-checking is time consuming and can run out of memory as was the case with the previous technique (Song and Touili, 2012b) by the same authors. Times reported in the paper range from few seconds (for 10 instructions) to over 250 s (for 10,000 instructions). Since real-world applications are much bigger than the samples tested, we believe that the proposed system cannot be used as a real-time malware detector.

Faruki et al. (2012) used API call-gram to detect malware. 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 call-gram. The call-gram becomes the input to a pattern matching engine. 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.

Eskandari and Hashemi (2012a, b) presented a method that uses 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. The implementation is not fully automatic and cannot be used as a real-time malware detector. The techniques used do not detect metamorphic malware, but as mentioned in the paper this option will be explored in the future.

Lee et al. (2010) proposed a technique that checks similarities of 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 (Garey and Johnson, 1990). So to reduce the call graph they divided these system calls into 128 groups (32 objects $\times$ 4 behaviors). This reduced the processing (from binary to code graph) time but also 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.

2.2. *Information flow analysis*

Yin and Song (2013) presented a method that uses dynamic taint analysis to automatically detect if an unknown sample exhibits malicious behavior or not. Their design consisted of four engines: taint engine, test engine, malware detection engine and malware analysis engine. Taint engine tracks the flow of information (all actions taken by the system as taint graphs) of the whole system. This information is used to detect malware from unknown samples by comparing the information against a set of defined policies.

The system implemented called *Panorama* is a plugin of an emulator. *Panorama* provides a fine grained whole system analysis but is not completely automatic and human analysis is needed to make more accurate detections. Running a sample in an emulator to detect malware has its own overheads. The paper lacks detailed information about the system performance and overheads.

Ghiasi et al. (2012) extended the idea of value set analysis (VSA) (Balakrishnan et al., 2005) proposed in Leder et al. (2009) for malware detection. Value set analysis is a static analysis technique that keeps track of the propagation and changes of values throughout an executable. Leder et al. (2009) track only register and stack values for efficiency reasons. These values are matched to reference list of value sets, generated from the infected files. Based on the matching a similarity score is computed which is used for detecting or classifying the malware. Ghiasi et al. (2012) track the register values for each API call in dynamic analysis setting. To collect register values, malware binaries are run and traced inside a controlled environment, which cannot be used as a real-time detector.

2.3. *Opcode-based analysis*

Wong and Stamp (2006) provides a good introduction to malware generation and detection, and served as a benchmark for comparison in several other studies (Toderici and Stamp, 2013; Runwal et al., 2012; Lin and Stamp, 2011; Baysa et al., 2013; Deshpande et al., 2014) on metamorphic malware. They analyzed 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. An 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 analyzed in Wong and Stamp (2006) are, G2, MPCGEN, NGVCK and VCL32. Based on the results, NGVCK (also used in this paper) outperforms other generators. VCL32 and MPCGEN have nearly similar morphing ability, and the malware programs generated by G2 (also used in this paper) 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.

Toderici and Stamp (2013) presented a technique that uses chi-squared (χ^2) test (Weisstein, Last accessed: October 30, 2014) to detect metamorphic malware. Their method is based on the observation that different compilers use different subset of instructions (opcodes). 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 instructions generated by a metamorphic malware generator. This technique is specific to a malware generator and is not platform independent. A χ^2 statistical test as described in Filiol and Josse (2007) 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 benign. The system implemented use a closed source disassembler, to disassemble the executables and is not fully automatic.

Runwal et al. (2012) presented a technique that uses 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 there is an edge from a node to a successor opcode. The edge is given a weight based on the frequency of occurrence of the successor opcode. This graph is directly compared, using matrices, with the graph of a 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 a malware is detected otherwise the program is benign.

Rad et al. (2012) proposed a method that uses histogram of instructions opcode to detect metamorphic malware. A histogram based on the frequency of instruction opcodes is built for each file and is compared against the already built histograms of malware samples to detect the file as malware or benign. The similarity between two histograms is measured using the Minkowski-form distance (Kruskal, 1964). The system implemented extracts opcodes from a binary file and uses MATLAB to generate a histogram of these opcodes, and is not fully automatic.

Austin et al. (2013) proposed a model that uses HMMs to point out 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.

Shanmugam et al. (2013) presented an opcode-based similarity measure inspired by substitution cipher cryptanalysis (Jakobsen, 1995) to detect metamorphic malware. They obtained promising results. A score is computed using an analog of Jackobsens algorithm (Jakobsen, 1995) that measures the distance between the opcode sequence of a given program and the opcode statistics for a malware program. A small distance indicates malware.

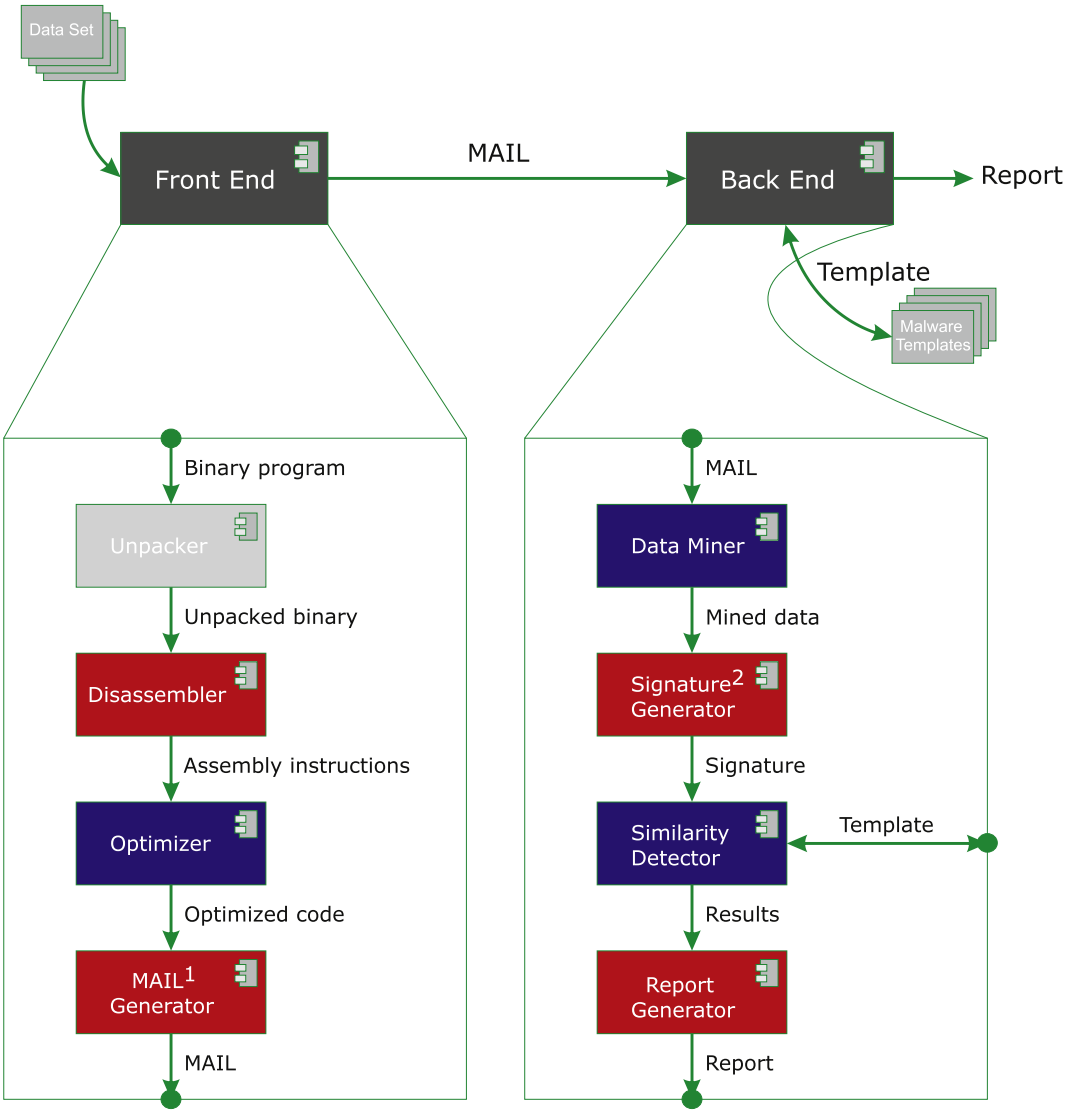


Fig. 1 – High level overview of MARD.

Vinod et al. (2012) used 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, and then align them into multiple sequences for comparison and detection. The authors assume that in a metamorphic malware some of the machine opcodes are replaced by equivalent machine opcodes but a complete change is impossible to keep the same functionality. 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. Experimental evaluation using the three types of signatures mentioned above yielded the following results. Single signature achieved a high detection rate (91%) but a very high false positive rate (52%). Group signature achieved a low detection rate (72.2%) but a very low false positive rate (0.01%). 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. Likewise it is unclear whether the proposed system be used as a real-time detector.

Recently, Canfora et al. (2014) presented a technique, that computes and uses frequency of program instructions present 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 occurence matrix (IOM) for a program. IOM associates each

opcode with the number of unique instructions that contain the opcode and 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 a malware from a benign program. However, the paper lacks details about the runtime performance of the proposed technique. Furthermore the paper does not provide any information about the distribution of the testing data. Likewise, it is unclear whether the proposed technique can also detect unknown malware.

3. MARD framework

3.1. Framework overview

Fig. 1 gives an overview of our proposed framework for Metamorphic Malware Analysis and Real-Time Detection (MARD). First a training dataset is built, also called *Malware Templates* in Fig. 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 Sections 4 and 5) the *Similarity Detector* (Fig. 1) detects the presence of malware in the program, using the *Malware Templates*. All the steps as shown in Fig. 1 are completley automated. There is no manual intervention during the complete 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 a view that if a program cannot be unpacked by the current available unpackers then the program is a malware, and also any good available unpacker can be interfaced with MARD.

3.2. Intermediate language

Most existing malware use binaries to infiltrate a computer system. Binary analysis is the process of automatically analyzing the structure and behavior of a binary program. We use binary analysis for malware detection.

To achieve platform independence, automation and optimization a binary program is first disassembled and translated to an intermediate language. We introduce in Alam et al. (2013), a new intermediate language for malware analysis named MAIL. In Alam (2013b), we explain in detail with examples how to translate an ×86 and an ARM assembly program into a MAIL program. For indirect jumps and calls (branches whose target is unknown or cannot be determined by static analysis) only a change in the source code can change them, so it is safe to ignore these branches for malware analysis where the change is only carried out in the machine (binary) code. We ignore indirect jumps and mark them as UNKNOWN when translating them to MAIL. The MAIL program is then annotated with patterns (called MAIL Patterns) for malware analaysis and detection.

3.3. Patterns for annotation

MAIL can be used to annotate 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 for pattern matching and representing opcodes/instructions in a program at a higher level during malware detection. The same pattern can be assigned to more than one statement in a MAIL program. There are in total 21 patterns in the MAIL language described 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; all the new statements when created are assigned this default value.

3.4. Characteristics

Some of the major characteristics and advantages of the MARD framework are as follows:

Platform independence: To make modern compilers (Aho et al., 2006; Muchnick, 1997) 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 ×86, 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 ×86 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.

3.5. Dataset

The dataset used in this paper for the experiments consists of total 5305 sample programs. Out of the 5305 programs, 1020 are metamorphic malware samples collected from three different resources (Rad et al., 2012; Runwal et al., 2012; Madenur Sridhara and Stamp, 2013), 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 1.

The dataset contains programs compiled with different compilers (Visual C++, Visual C#, Visual Basic and GCC) for Windows (32 and 64 bits, PE format) and Linux (32 and 64 bits, ELF format) operating systems. The size of the malware samples range from 1 KB to 299 KB and the size of the benign samples range from 9 bytes–10 MB. The 1020 malware samples belong to the following three metamorphic *family* of viruses: Next Generation Virus Construction Kit (NGVCK) (NGVCK, Last accessed: October 30, 2014), Second Generation Virus Generator (G2) (G2, Last accessed: October 30, 2014) and Metamorphic Worm (MWOR) generated by metamorphic generator (Madenur Sridhara and Stamp, 2013). NGVCK generates malware that is highly metamorphic, and contains obfuscations that range from simple to complex. They include dead code insertion, instruction reordering and insertion of indirect jumps. Malware generated by G2 is modestly metamorphic. MWOR uses two kinds of obfuscations: dead code insertion and equivalent instruction substitution. Out of 1020 malware samples, 270 are NGVCK, 50 are G2, and 700 are MWOR (each 100 with a padding ratio of 0.5, 1.0, 2.0, 2.5, 3.0, 3.5 and 4.0).

Since the six approaches (Wong and Stamp, 2006; Runwal et al., 2012; Toderici and Stamp, 2013; Rad et al., 2012; Austin et al., 2013; Shanmugam et al., 2013), as discussed in Section 2, are the only existing opcode-based approaches capable of detecting metamorphic malware, we have also included some of the malware samples used in their evaluations in our dataset in addition to other malware samples. This allows comparison of our results with their results in a more objective way.

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

4. ACFG detection technique

Current techniques (Eskandari and Hashemi, 2012a, b; Anju et al., 2010; Song and Touili, 2012a; Vinod et al., 2009; Bruschi et al., 2006; Cesare and Xiang, 2011; Guo et al., 2010; Kirda et al., 2006; Flake, 2004) that use CFG for malware detection are either compute intensive or have poor detection rates and cannot handle malware with smaller CFGs. We propose, in this paper, a new technique named Annotated Control Flow Graph (ACFG) that can enhance the detection of metamorphic malware and can handle malware with smaller CFGs.

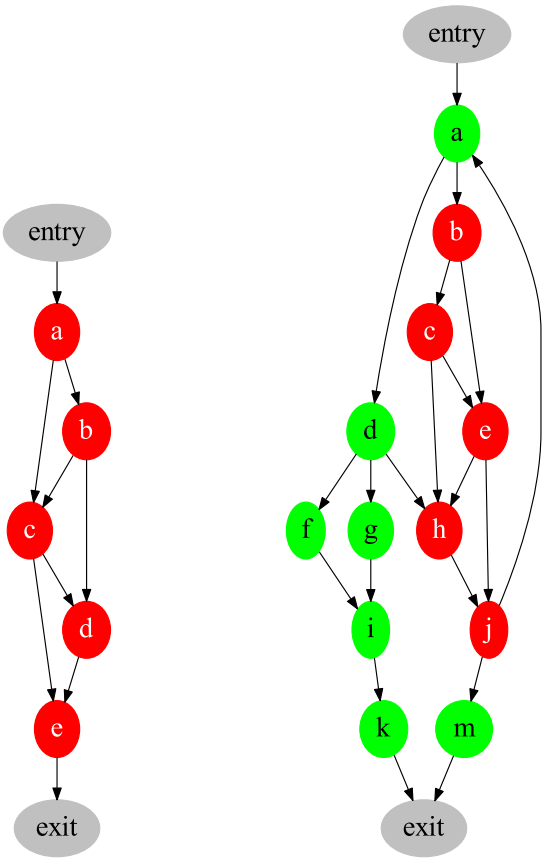
4.1. Definitions

Before describing the proposed technique in detail, we first provide the following definitions related to ACFG:

Definition 1. A **Basic block** is a sequence of instructions with a single entry and single exit points. There is no instruction after the first instruction that is the target instruction of a

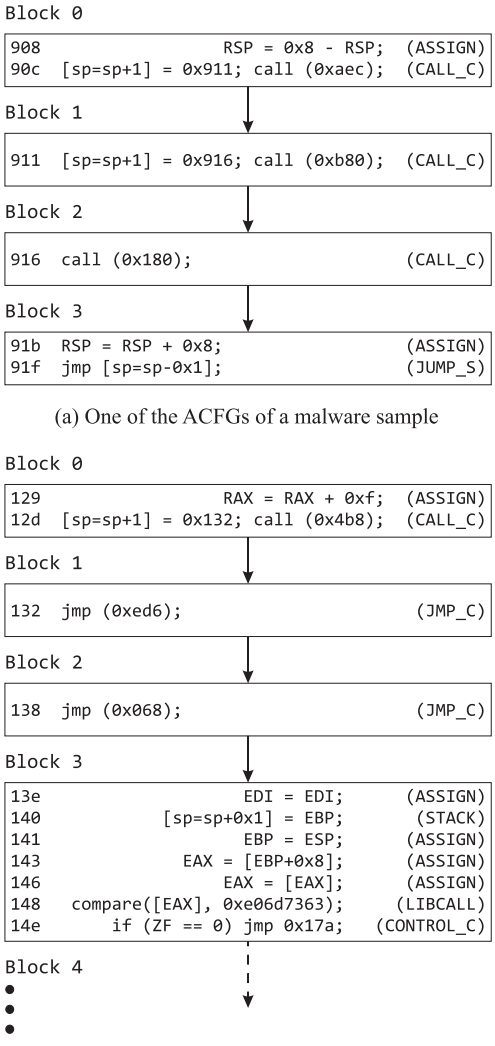
Table 1 – Dataset distribution based on the size of each program sample.

Malware samples (1020)		Benign program samples (4285)	
Range of size (bytes)	Number of samples	Range of size (bytes)	Number of samples
1343–1356	50	9–19,997	817
3584–4096	35	20,336–29,984	406
8192–16,384	215	30,144–49,800	492
29,356–35,324	200	50,144–119,936	935
36,864–40,772	102	120,296–980,840	1514
40,960–46,548	101	1,001,936–1,553,920	50
52,292–57,828	200	1,606,112–3,770,368	48
67,072–69,276	101	4,086,544–5,771,408	12
70,656–74,752	4	6,757,888–8,947,200	4
271,872–299,520	12	9,074,688–10,124,353	7



(a) A malware sample (b) The malware embedded inside a benign program

Fig. 2 – An example of subgraph matching. The graph in Figure (a) is matched as a subgraph of the graph in Figure (b).



(a) One of the ACFGs of a malware sample

(b) One of the ACFGs of a benign program

Fig. 3 – 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).

jump instruction, and only the last instruction can jump to a different block.

Definition 2. A **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 **Control Flow Graph (CFG)** is a directed graph $G = (V, E)$, where V is a set of basic blocks and E is a 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.

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 $G \cong H$.

4.2. *Metamorphic malware detection using ACFG*

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

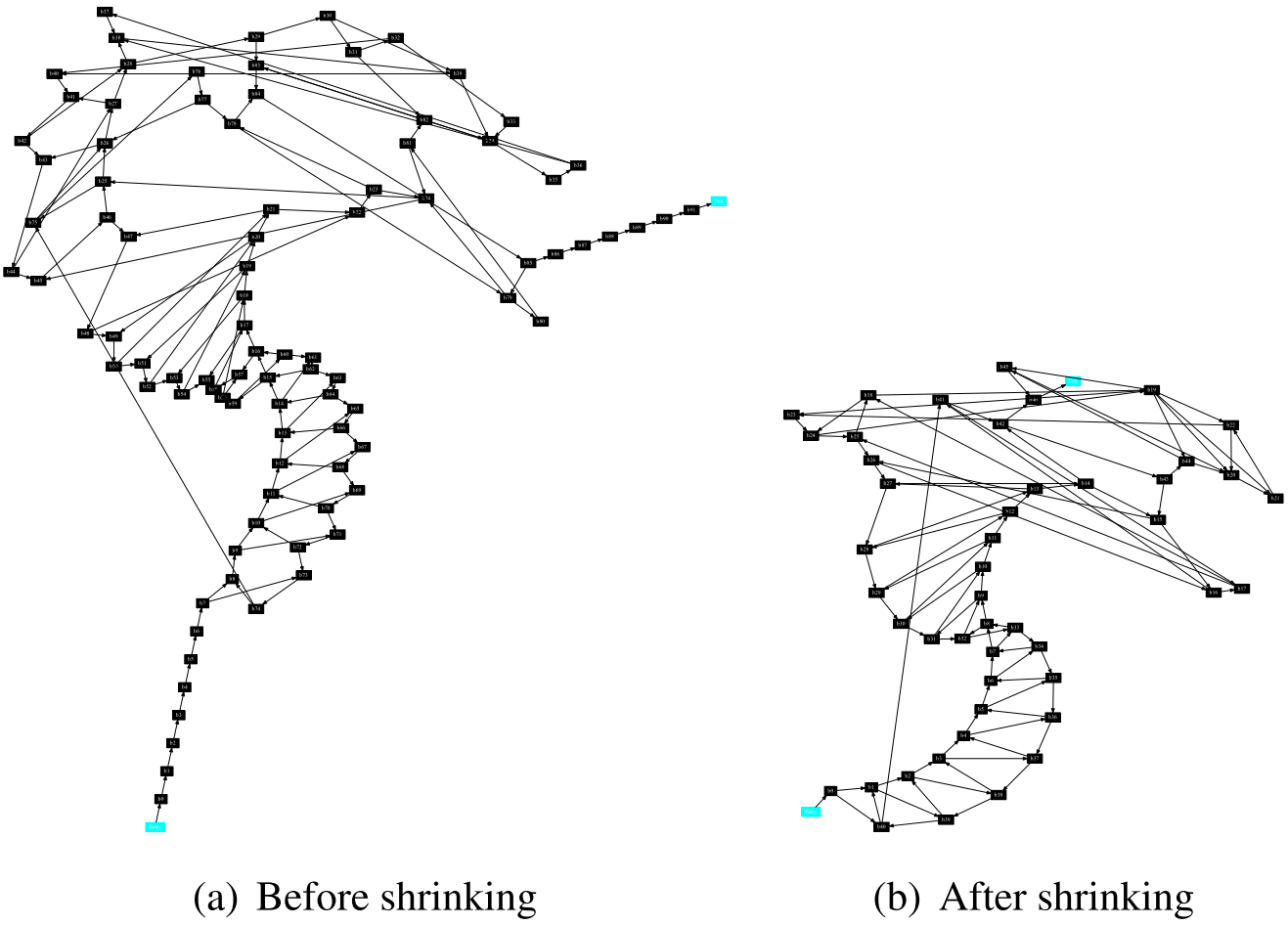


Fig. 4 – Examples of ACFGs. One of the functions of one of the samples of the MWOR class of malware, before and after shrinking. The ACFG has been reduced from 92 nodes to 47 nodes.

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.

4.2.1. *Subgraph matching*

In our malware detection approach, graph matching is defined in terms of **subgraph isomorphism** (Gross and Yellen, 2005). Given the input of two graphs, *subgraph isomorphism* determines if one of the graphs contains a subgraph that is isomorphic to the other graph, and is denoted as $\cong$. An example of subgraph matching is shown in Fig. 2. Based on this definition of *subgraph isomorphism* 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.

4.2.2. *Pattern matching*

Very small graphs when matched against a large graph can produce a false positive. Likewise to alleviate the impact of small graphs on detection accuracy, we integrate a *Pattern Matching* sub-component within the *Subgraph Matching* component, which is part of the *Similarity Detector* shown in Fig. 1. Every statement in ACFG is assigned a *pattern* as explained in Section 3.3. 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 ACFG 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 Fig. 3. One of the ACFGs of a malware sample, shown

Table 2 – 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	Optimization level 1		Optimization level 2	
	Total instructions 4045	Total instructions 1880		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 that the frequency of distribution of the 9 Opcodes change from 21.05% to upto 151.91% from optimization level 0 to optimization level 1, and from 14.04% to upto 276.20% from optimization level 0 to optimization level 2, whereas the frequency of distribution of MAIL Pattern *ASSIGN* changes only by 3.22% and 8.91%.

in Fig. 3(a), is *isomorphic* to a subgraph of one of the ACFGs of a benign program, shown in Fig. 3(b). The benign program is not detected as a malware, because not all the statements have the same pattern.

4.3. *Runtime optimization of Subgraph Matching*

The *Subgraph Matching* component matches the 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 but provide some options for optimization. We use two techniques to improve the runtime, namely, parallelization and ACFG reduction, described in the following.

4.3.1. *Parallelization*

Because of the ubiquitousness of multicores in the host machines (also called the end points) and to optimize the runtime, we decided to use Windows threads to parallelize the *Subgraph Matching* component. We developed Equation (1) empirically (Alam et al., 2014) to compute the maximum number of threads to be used by the *Subgraph Matching* component. We also give an option to the user to choose the maximum number of threads used by the tool for the *Subgraph Matching* component. The percentage of CPU utilization on average was 3 times more with threads than without threads. This was also confirmed by the improvement in the runtime (Alam et al., 2014).

$$\text{Threads} = (\text{NC})^3 \quad (1)$$

where NC = Number of CPUs/Cores

4.3.2. *ACFG reduction*

One of the advantages of using ACFG is that it contains annotations for malware detection. Our detection method uses 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 be reduced to very small number of blocks) we still get a good detection rate because of the combination of the two techniques.

To reduce the number of blocks (nodes) in an ACFG for runtime optimization 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 merged only if the merging does not change the control flow of a 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 will be merged.

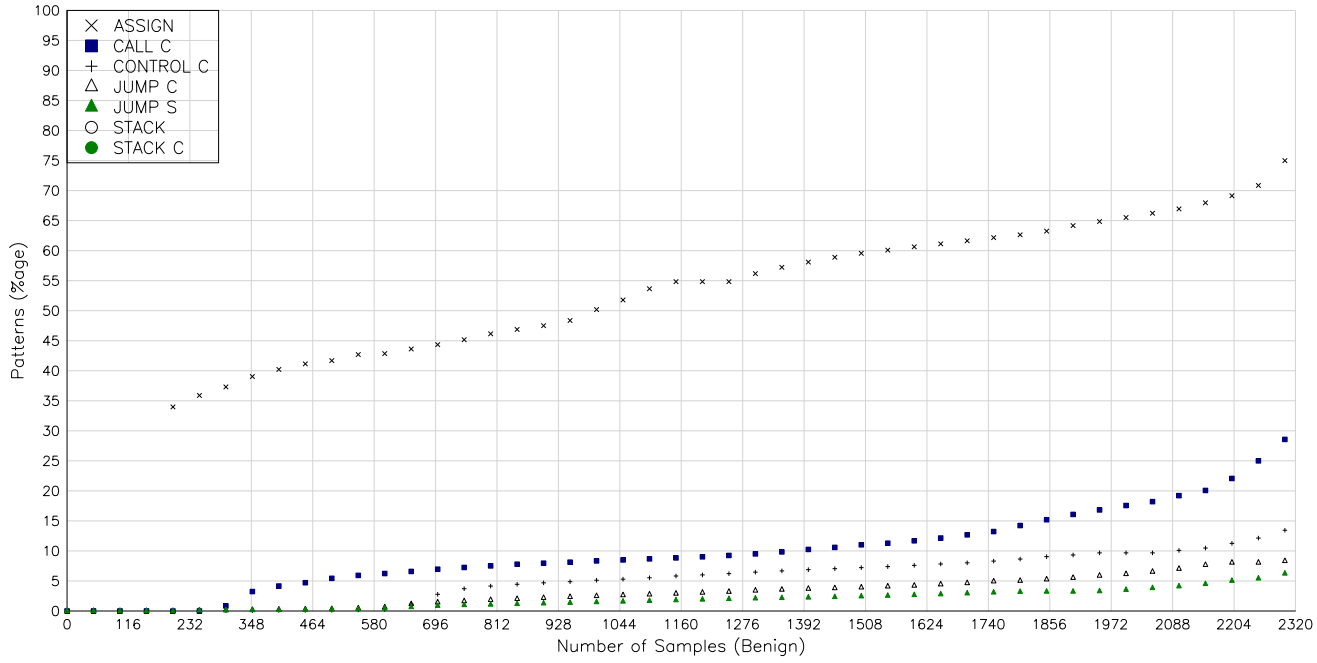
Fig. 4 shows examples of ACFGs, from the dataset described in this paper, before and after shrinking. The shrinking does not change the shape of a graph. As we can see in the Figure, the shapes of the graphs before and after shrinking are the same. More of these examples are available at (Alam, 2013a).

We were able to substantially reduce the number of nodes per ACFG (in total a 90.6% reduction), as shown in Table 4. This reduced the runtime on average by 2.7 times (for smaller dataset) and 100 times (for larger dataset), and still achieved a detection rate of 98.9% with a false positive rate of 4.5% as latter shown in Table 6.

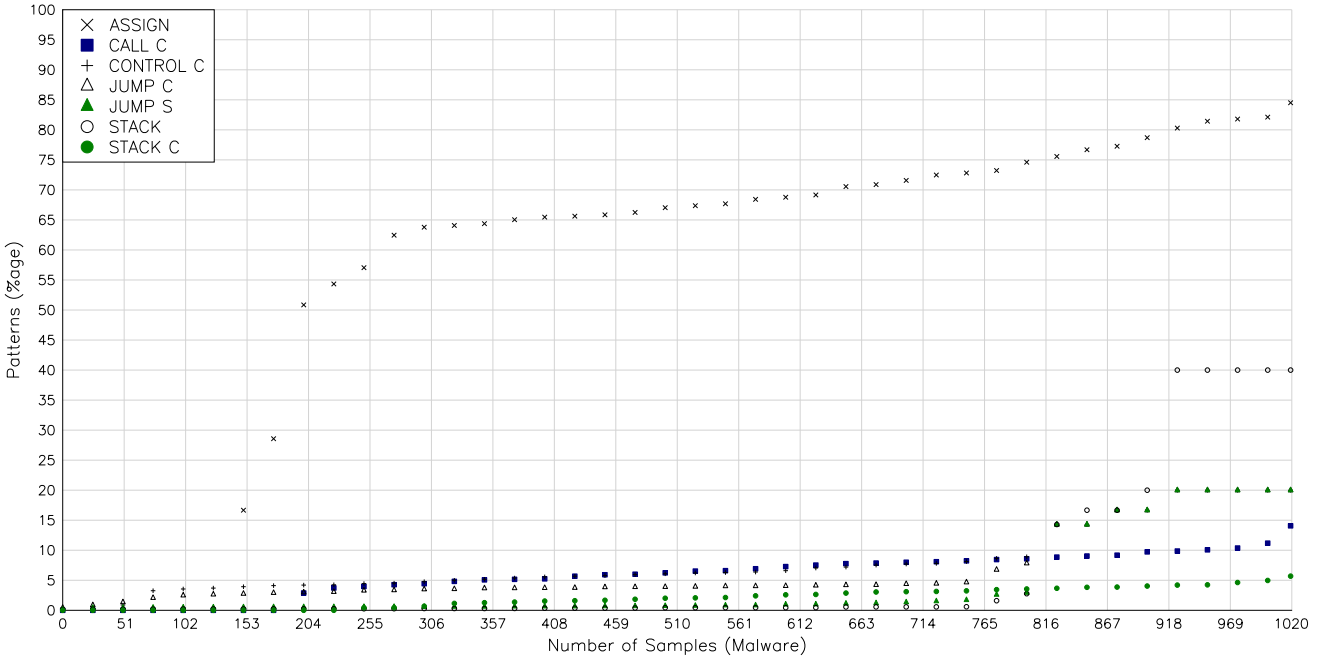
5. SWOD-CFWeight detection technique

5.1. *Rationale and overview*

Techniques based on behavior analysis (Faruki et al., 2012; Yin and Song, 2013; Song and Touili, 2012a; Vinod et al., 2012; Ghiasi et al., 2012; Bruschi et al., 2006; Guo et al., 2010; Kirda et al., 2006; Flake, 2004) are used to detect metamorphic malware, but are compute intensive and are not suitable for real-time detection. A subset of other techniques (Runwal et al., 2012; Toderici and Stamp, 2013; Rad et al., 2012; Santos et al., 2013; Shabtai et al., 2012; Wong and Stamp, 2006; Austin et al., 2013; Shanmugam et al., 2013) that use opcodes



(a) MAIL pattern distributions for benign samples



(b) MAIL pattern distributions for malware samples

Fig. 5 – MAIL pattern distributions based on the percentage of the MAIL patterns in each sample in the dataset.

to detect malware, such as (Runwal et al., 2012; Toderici and Stamp, 2013; Rad et al., 2012; Wong and Stamp, 2006; Austin et al., 2013; Shanmugam et al., 2013), have been shown to detect metamorphic malware. One of the main advantages of using opcodes for detecting malware is that detection 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.

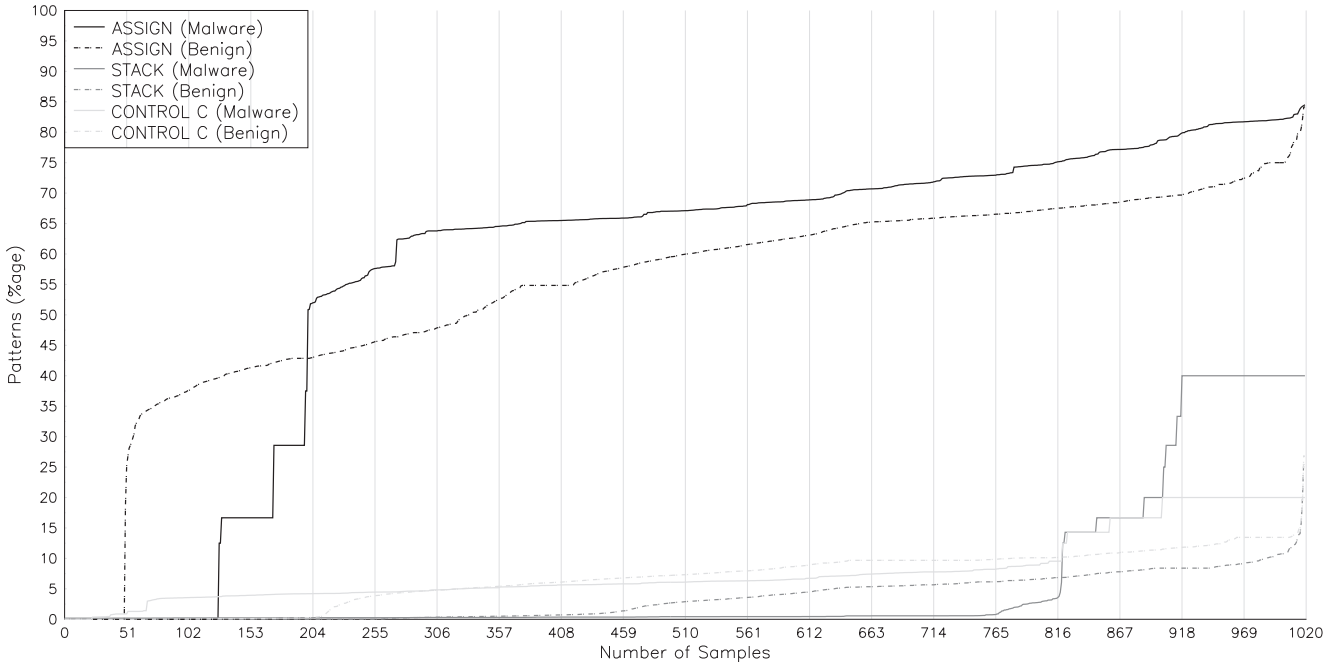


Fig. 6 – Superimposing three of the MAIL Patterns distributions from Fig. 5.

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 help in selecting an appropriate number of features and reduce the runtime cost.

We illustrate our approach using MAIL, that adequately addresses some of the problems described above. Table 2 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, which correspond to tokens, present in MAIL 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.

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 (Alam et al., 2013) for effective metamorphic malware detection.

5.2. *Statistical analysis of MAIL pattern distributions*

We conducted an empirical study of MAIL pattern distributions in order to investigate how MAIL pattern frequency differs between a malware and a benign sample, and define new metrics for metamorphic malware detection.

We used a subset of the dataset presented in Section 3.5 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 Fig. 5. 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 Fig. 5 show the number of samples on the *X*-axis and the Patterns' percentage on the *Y*-axis. On average the occurrence of MAIL pattern *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*.

To show the difference between the malware and benign MAIL pattern distributions we superimpose the two Fig. 5(a) and (b) using all the values of the MAIL Pattern distributions. For clarity and readability only three Patterns, *ASSIGN*, *CONTROL_C* and *STACK*, are superimposed. Fig. 6 shows the superimposition of these MAIL patterns distributions.

The graph in Fig. 6 is divided using perpendicular lines. When the two plots (malware and benign) of a pattern horizontally divide the space between two perpendicular lines, this division is called one pocket of the window. There are significant pockets of windows in MAIL pattern *ASSIGN* that show significant difference between malware and benign samples. There are few pockets of windows in MAIL patterns *CONTROL_C* and *STACK* that show significant difference between malware and benign samples. We use this observation and the differences as a motivation to develop the (*SWOD*) metric based on MAIL patterns, which we define formally in Section 5.3.1.

Out of the seven patterns shown in Fig. 5 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 provide the rationale for defining the *CFWeight* as a metric that captures the amount of change in the control flow of a program.

5.3. *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 underly 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.

5.3.1. *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 i-th malware sample and P_jM_i be the percentage of the j-th MAIL Pattern in M_i. Similarly, Let B_i be the i-th benign sample and P_jB_i be the percentage of the j-th MAIL Pattern in B_i. We compute the MAIL Pattern distributions in a MAIL program, as follows:

$$P_jM_i = \frac{p_j}{\sum_{i=1}^{N} p_i} \times 100 \quad (2)$$

$$P_jB_i = \frac{p_j}{\sum_{i=1}^{N} p_i} \times 100 \quad (3)$$

where p_j and p_i are the number of times the j-th and i-th 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 ($N = 21$) (Alam et al., 2013).

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 sorted in decreasing order: $PD_{bj} = \{P_jB_1, P_jB_2, P_jB_3, \ldots\ldots P_jB_l\}$.

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:

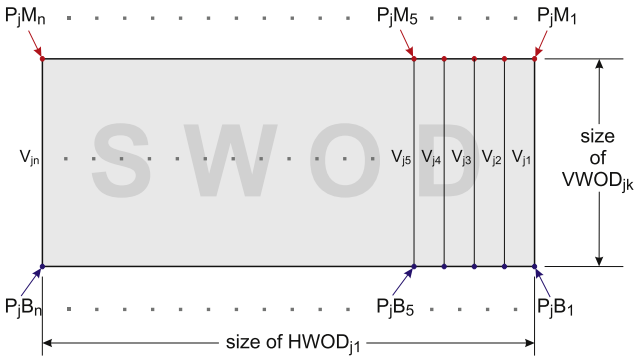


Fig. 7 – Sliding Window of Difference ($SWOD_{j1}$) as defined in *Definition 10*. $HWOD_{j1} = \{V_{j1}, V_{j2}, V_{j3}, \ldots V_{jn}\}$, where V_{j1}, V_{j2}, V_{j3}, ... V_{jn}, are the *VWODs*.

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 $HWOD_{jx} = \{VWOD_{j[(x-1)\times n+1]}, VWOD_{j[(x-1)\times n+2]}, VWOD_{j[(x-1)\times n+3]}, \ldots., VWOD_{j[x\times n]}\}$, where $\mathbf{n} = \lceil l/N \rceil$. 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 $SWOD_{jx}$.[2]

[2] In other words, *HWO* D_{jx} is a *SWO* D_{jx} if and only if it satisfies the *minratio*.

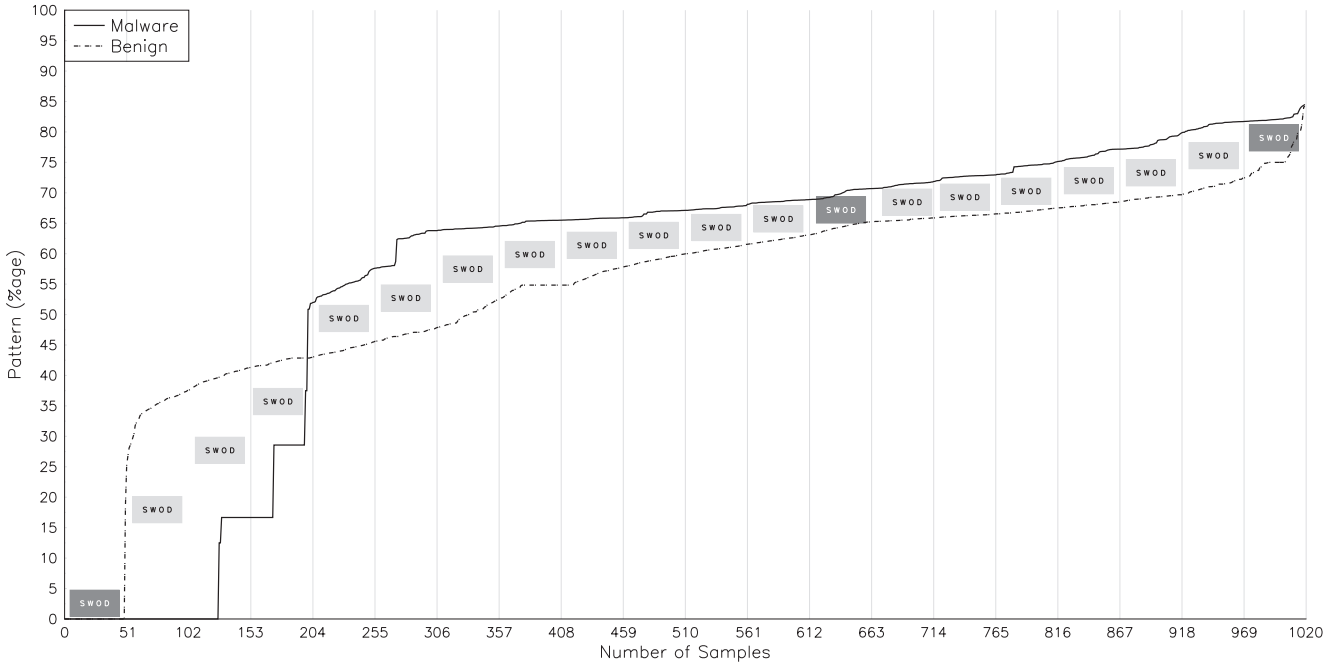


Fig. 8 – Sliding Window of Differences (*SWODs*) for the MAIL Pattern ASSIGN.

Definition 11. : The *SWOD* for pattern P_j denoted $SWOD_j$ is the union set of all the $SWOD_{jx}$: $SWOD_j = \cup_{1 \le x \le N} SWOD_{jx}$

The determination of the SWOD depends on the values of *minsize* and *minratio*, which 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* for an optimal performance of the malware detector. A graphical depiction of a *SWOD* is shown in Fig. 7.

Fig. 8 shows an example of *SWOD* for the MAIL Pattern *ASSIGN*, which is 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 $n = 51$ (1020/20).

According to the definitions presented above at least 30 ($51 \times 0.60 \approx 30$) of the *VWODs* should have a value$\geq$5, for the example of *SWODs* shown in Fig. 8. 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 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. This sorting is also useful for detecting a malware mixed with a benign program. For finding the real *SWODs* (as explained above) we go through candidate *SWODs* in the lists, as shown in lines through 13–16 in Algorithm 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.

5.3.2. *Control flow weight and MAIL program signature*

We assign to each statement in a MAIL program a partial weight computed from a 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. A back jump's (possibly a loop) weight is double the length of the jump.
5. A jump whose target is outside the function is assigned a weight equal to the distance (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 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.

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

Fig. 9 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

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 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.

5.3.3. *Signature matching and malware detection*

To detect if a new program is a 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 $S_i = [s_{ik}]_{1 \le k \le p}$ and $S_j = [s_{jk}]_{1 \le k \le q}$ denote the signature vectors, sorted in decreasing order, of two different programs, such that $p \le q$.

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:

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

Two weights s_{ik} and s_{jk} are matching if and only if $d_{ijk} \le \varepsilon_1$, where $\varepsilon_1 = 3 \times minsize$.

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 $y/r \times 100 \ge \varepsilon_2$, where $\varepsilon_2 = minratio/2.25$.

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 Fig. 9 there are 3 weights with the value 3, so we store a 3 at index 3. There is only 1 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, there is a successfull match of the signature of the MAIL program with the signature of the malware sample M_{12} and hence the program is tagged as a malware. The comparison stops after the first successfull match with the signature of a malware sample. The lines without a × (12 of them, $\ge \varepsilon_2$) show a match (percentage of difference is $\le \varepsilon_1$) between the corresponding weights, and the lines marked with a × (6 of them) show a no-match (percentage of difference is $> \varepsilon_1$) between the corresponding weights.

5.3.4. *Complexitiy analysis*

Time complexity of Algorithm 1: There are two outer loops in function *ComputeWeight()*. The 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*

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

Require: *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 store the Signature for easy and fast comparison. e.g: The Signature: 11111444777777777 is stored as: 05003009 i.e: there are 5 1's, 3 4's and 9 7's in the Signature.

second loop $= N(nm + nb + nm \log m + b \log b) + N\,min(m, b)$, and is simplified to **O(*N n* log *n*)**, where *N* is the number of Patterns in MAIL, and *n* is either the number of malware 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 **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 (Cormen et al., 2009).

6. Evaluation, analysis and comparison

In this Section we evaluate the correctness and the efficiency of our proposed techniques, and present results, discussions and analysis of this evaluation. We also compare the proposed framework with other such malware detection systems.

6.1. Evaluation metrics

We use **k-fold cross validation** to estimate the performance of our techniques. In k-fold cross validation the original sample is divided into *k* equal size subsamples. One of the samples is used for testing and the remaining $k-1$ samples are used for training. The cross validation process is then repeated *k* times with each of the *k* subsamples used exactly once for validation. Before evaluating the proposed techniques, we first define the following evaluation metrics:

True positive (**TP**) is the number of malware 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 that are classified as benign.

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

$$\text{Precision} = \frac{TP}{TP + FP} \quad \text{Accuracy} = \frac{TP + TN}{P + N}$$

where *P* and *N* are the total number of malware and benign programs respectively. Now we define the mean maximum

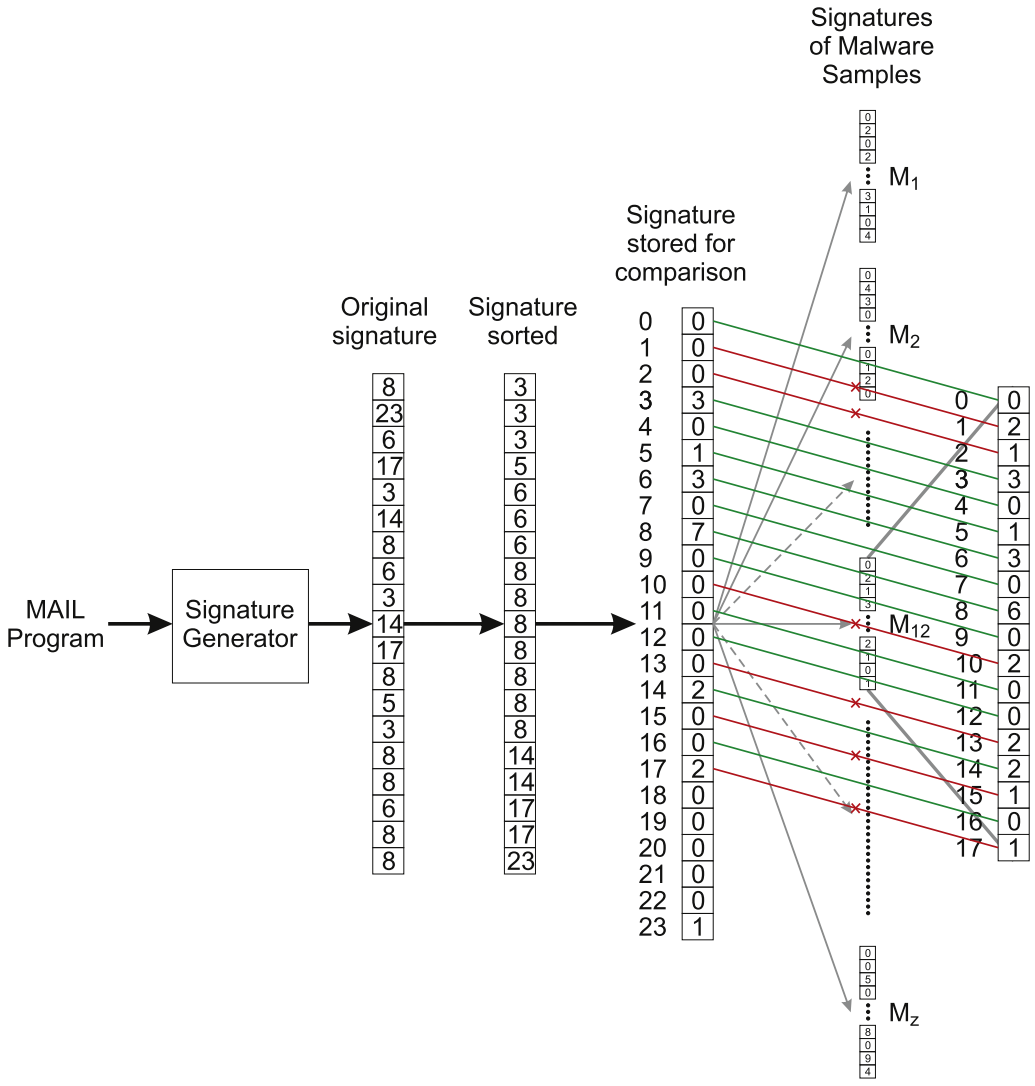


Fig. 9 – Malware detection using MAIL program signatures.

precision (**MMP**) and mean maximum accuracy (**MMA**) for k-fold cross-validation as follows:

$$MMP = \frac{1}{k}\sum_{i=1}^{k} \text{Precision}_i \tag{4}$$

$$MMA = \frac{1}{k}\sum_{i=1}^{k} \text{Accuracy}_i \tag{5}$$

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:

Table 3 – Dataset distribution based on the number of 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

Table 4 – Dataset distribution based on the size (number of nodes) for each 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	60,284	1–10	242,240
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

$$TPR = \frac{TP}{P} \quad FPR = \frac{FP}{N} \tag{6}$$

6.2. Performance of MARD with ACFG

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

6.2.1. Dataset based on ACFGs

Graph matching is a time consuming technique, so we did not use all the 5305 samples in the dataset for evaluating ACFG. We selected only 3350 sample Windows and Linux programs. Out of the 3350 programs, 1020 are metamorphic malware samples described in Section 3.5. The dataset distribution based on the number of ACFGs for each sample, and size of the CFG after normalization is shown in Tables 3 and 4 The normalizations carried out are removal of NOP, instructions that are not required for malware analysis (when translating to MAIL (Alam, 2013b)) and other junk instructions.

The dataset contains a variety of programs with ACFGs, ranging from simple to complex for testing. As shown in Table 3 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 have 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 4. The sizes of the ACFGs of the malware samples range from 1 node to 301 nodes, and the size of the ACFG of the benign programs range from 1 node to 521 nodes.

Table 5 – 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 8.028 s with ACFG shrinking and 40.4288 s 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.

Table 6 – 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 426.588 s with ACFG shrinking and over 125,400 s (over 34 h) 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 h) reported is just for one run of the experiment without ACFG shrinking.

6.2.2. Experiments

The experiments were run on the following machine: Intel Core i5 CPU M 430 (2 Cores) @ 2.27 GHz with 4 GB RAM, operating system Windows 8 professional. To get the accurate and best results out of the current 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. For both these experiments we use a threshold value of 25%. In the following two Sections we present and describe these two experiemnts.

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, 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. The obtained results are listed in Table 5. The low DR (94%) when using a smaller training dataset (25 samples, i.e, 10% of the malware samples), is because of the variability present in metamorphic malware samples used in the experiments, and the DR improved (99.6%) when a larger training dataset (125 samples, i.e, 50% of the malware samples) is used.

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. The obtained results are listed in Table 6. The DR improved from 97% when the size of the training set is 204 (20% of the malware samples) to 98.9% when we used a training dataset of 510 samples (50% of the malware samples).

6.2.3. Comparison with others

Table 7 gives a comparison of ACFG with the research efforts of detecting malware (including metamorphic malware) discussed in Section 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.

Table 7 – Comparison of ACFG with the metamorphic malware detection techniques based on control and information flow analysis.

System	Analysis type	DR	FPR	Data set size Benign/Malware	Real time	Platform
MARD-ACFG	Static	98.9%	4.5%	2330/1020	✓	Win & Linux 64
API-CFG (Eskandari and Hashemi, 2012a, b)	Static	97.53%	1.97%	2140/2305	✗	Win 32
Call-Gram (Faruki et al., 2012)	Static	98.4%	2.7%	3234/3256	✗	Win 32
Code-Graph (Lee et al., 2010)	Static	91%	0%	300/100	✗	Win 32
DTA (Yin and Song, 2013)	Dynamic	100%	3%	56/42	✗	Win XP 64
Model-Checking (Song and Touili, 2012a)	Static	100%	1%	8/200	✗	Win 32
MSA (Vinod et al., 2012)	Static	91%	52%	150/1209	✗	Win 32
VSA-1 (Leder et al., 2009)	Dynamic	100%	0%	25/30	✗	Win 32
VSA-2 (Ghiasi et al., 2012)	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 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.

Table 7 reports the best DR results achieved by these detectors. Out of the 10 systems, ACFG clearly shows superior results and, unlike others is fully automatic, supports 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.

6.3. *Performance of MARD with* SWOD-CFWeight

We carried out an empirical study to analyze 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 6.2.

6.3.1. *Experiments*

First we compute the values of *minsize* and *minratio*, as defined in *Definition 7* and *Definition 9* 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. 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 k) k-fold cross validations. For example for a fair comparison we used the same k-fold cross validation as used in Section 6.2 while comparing *SWOD-CFWeight* with *ACFG*.

6.3.2. *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 4 GB RAM, operating system Windows 8 professional. The results of all the experiments are listed in Table 8.

The first four rows in Table 8 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-

Table 8 – 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 are $\leq$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.

Table 9 – Comparison of *SWOD-CFWeight* with the malware detection techniques based on opcode analysis.

Technique	Analysis type	DR	FPR	Dataset size Benign/Malware	Real time	Platform
MARD-SWOD-CFWeight	Static	99.08%	0.93%	1101/250	✓	Win & Linux 64
Opcode-HMM-Wong (Wong and Stamp, 2006)	Static	~ 90%	~ 2%	40/200	✗	Win & Linux 32
Chi-Squared (Toderici and Stamp, 2013)	Static	~ 98%	~ 2%	40/200	✗	Win & Linux 32
Opcode-HMM-Austin (Austin et al., 2013)	Static	93.5%	0.5%	102/77	✗	Win & Linux 32
Opcode-SD (Shanmugam et al., 2013)	Static	~ 98%	~ 0.5%	40/800	✗	Linux 32
Opcode-Graph (Runwal et al., 2012)	Static	100%	1%	41/200	✗	Win & Linux 32
Opcode-Histogram (Rad et al., 2012)	Static	100%	0%	40/60	✗	Win & Linux 32
Opcode-Seqs-Santos (Santos et al., 2013)	Static	96%	6%	1000/1000	✗	Win 32
Opcode-Seqs-Shabtai (Shabtai et al., 2012)	Static	~ 95%	~ 0.1%	20,416/5677	✗	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.

malware applications. The high FP rates of *SWOD-CFWeight* (row four and onwards) are due to the size of the *SWOD* used in the experiments, as explained below.

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 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 of 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 effects the performance of the detector. The dataset used in the last row contains additional benign files whose size range from 1 MB–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 make this claim certain, in future work it will be validated with more such experiments.

6.3.3. *Comparison with others*

Table 9 gives a comparison of *SWOD-CFWeight* with the existing opcode-based malware detection approaches discussed in Section 2 except *Opcode-Seqs-Santos* and *Opcode-Seqs-Shabtai*. None of the prototype systems implemented 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* 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. Like other malware detection techniques, it may be possible to evade detection by *SWOD*, such as, an advanced adversary aware of this system, by adding code (padding) to a malware program can alter the difference between malware and benign MAIL pattern distributions.

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 execution flow of the program. *CFWeight* proposed in this paper includes this information to an extent that helps detect metamorphic malware.

Table 9 reports the best DR result achieved by the detection techniques discussed in Section 2 and others. Out of the 6 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.

7. Conclusion and future work

In this paper, we have presented a new metamorphic malware detection framework named *MARD* that implements the two novel techniques proposed in this paper named *ACFG* and *SWOD-CFWeight*, and 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 the proposed techniques, clearly shows results in the top, and unlike others is fully automatic, supports 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.

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 from other techniques (that also use control flow semantics) and results in a high detection rate for metamorphic malware 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 dataset. 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.

REFERENCES

Aho AV, Lam MS, Sethi R, Ullman JD. Compilers: principles, techniques, and tools. 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.; 2006.

Alam S. Examples of CFGs before and after shrinking. 2013. http://www.cs.uvic.ca/~salam/PhD/cfgs.html.

Alam S. MAIL: malware analysis intermediate language. 2013. http://www.cs.uvic.ca/~salam/PhD/TR-MAIL.pdf.

Alam S, Horspool RN, Traore I. MAIL: malware analysis intermediate language – a step towards automating and optimizing malware detection. In: Security of Information and Networks. SIN'13. New York, NY, USA: ACM SIGSAC; November 2013.

Alam S, Horspool RN, Traore I. MARD: a framework for metamorphic malware analysis and real-time detection. In: Advanced Information Networking and Applications, Research Track – Security and Privacy. AINA'14. Washington, DC, USA: IEEE Computer Society; May 2014.

Anju SS, Harmya P, Jagadeesh N, Darsana R. Malware detection using assembly code and control flow graph optimization. In: A2CWiC, 2010. New York, NY, USA: ACM; 2010. 65:1–65:4.

Austin TH, Filiol E, Josse S, Stamp M. Exploring hidden markov models for virus analysis: a semantic approach. In: 2013 46th Hawaii International Conference on System Sciences (HICSS); Jan 2013. p. 5039–48.

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

Baysa D, Low R, Stamp M. Structural entropy and metamorphic malware. J Comput Virol Hacking Tech 2013;9(4):179–92.

Borello J-M, Me L. Code obfuscation techniques for metamorphic viruses. J Comput Virol 2008;4(3):211–20.

Bruschi D, Martignoni L, Monga M. Detecting self-mutating malware using control-flow graph matching. In: DIMVA. Berlin, Heidelberg: Springer-Verlag; 2006. p. 129–43.

Canfora G, Iannaccone A, Visaggio C. Static analysis for the detection of metamorphic computer viruses using repeated-instructions counting heuristics. J Comput Virol Hacking Tech 2014;10(1):11–27.

Cesare S, Xiang Y. Malware variant detection using similarity search over sets of control flow graphs. In: TrustCom, 2011; November 2011. p. 181–9.

Collberg C, Thomborson C, Low D. Manufacturing cheap, resilient, and stealthy opaque constructs. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL'98. New York, NY, USA: ACM; 1998. p. 184–96.

Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to algorithms. 3rd ed. The MIT Press; 2009.

Deshpande S, Park Y, Stamp M. Eigenvalue analysis for metamorphic detection. J Comput Virol Hacking Tech 2014;10(1):53–65.

Eskandari M, Hashemi S. A graph mining approach for detecting unknown malwares. J Vis Lang Comput Jun. 2012a;23(3):154–62.

Eskandari M, Hashemi S. ECFGM: enriched control flow graph miner for unknown vicious infected code detection. J Comput Virol Aug. 2012b;8(3):99–108.

Faruki P, Laxmi V, Gaur MS, Vinod P. 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: ACM SIGSAC; 2012.

Filiol E, Josse S. A statistical model for Undecidable Viral detection. J Comput Virol 2007;3:65–74.

Flake H. Structural comparison of executable objects. In: Flegel U, Meier M, editors. DIMVA. Vol. 46 of LNI. GI; 2004. p. 161–73.

G2. Second generation virus generator, http://vxheaven.org/vx.php?id=tg00 [last accessed 30.10.14].

Garey MR, Johnson DS. Computers and intractability; a guide to the theory of NP-completeness. New York, NY, USA: W. H. Freeman & Co.; 1990.

Ghiasi M, Sami A, Salehi Z. Dynamic malware detection using registers values set analysis. In: Information Security and Cryptology; 2012. p. 54–9.

Gross JL, Yellen J. Graph theory and its applications. Discrete mathematics and its applications. 2nd ed. Chapman & Hall/CRC; 2005.

Guo H, Pang J, Zhang Y, Yue F, Zhao R. Hero: a novel malware detection framework based on binary translation. In: ICIS, 2010, Vol. 1; oct. 2010. p. 411–5.

ITU. The world in 2013: ICT Facts and figures. © ITU; 2013.

Jakobsen T. A fast method for cryptanalysis of substitution ciphers. Cryptologia 1995;19(3):265–74.

Kirda E, Kruegel C, Banks G, Vigna G, Kemmerer RA. Behavior-based Spyware detection. In: Proceedings of the 15th Conference on USENIX Security Symposium. Volume 15. USENIX–SS'06. Berkeley, CA, USA: USENIX Association; 2006.

Kruskal J. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. Psychometrika 1964;29:1–27.

Kuzurin N, Shokurov A, Varnovsky N, Zakharov V. On the concept of software obfuscation in computer security. In: Proceedings of the 10th International Conference on Information Security. ISC'07. Berlin, Heidelberg: Springer-Verlag; 2007. p. 281–98.

Leder F, Steinbock B, Martini P. Classification and etection of metamorphic malware using value set analysis. In: MALWARE, 2009; oct. 2009. p. 39–46.

Lee J, Jeong K, Lee H. Detecting metamorphic malwares using code graphs. In: SAC, 2010. New York, NY, USA: ACM; 2010. p. 1970–7.
Lin D, Stamp M. Hunting for undetectable metamorphic viruses. J Comput Virol 2011;7(3):201–14.
Linn C, Debray S. Obfuscation of executable code to improve resistance to static disassembly. In: ACM CCS. New York, NY, USA: ACM; 2003. p. 290–9.
Madenur Sridhara S, Stamp M. Metamorphic worm that carries its own morphing engine. J Comput Virol Hacking Tech 2013;9(2):49–58.
Muchnick SS. Advanced compiler design and implementation. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 1997.
NGVCK. Next generation virus construction kit, http://vxheaven.org/vx.php?id=tn02 [last accessed 30.10.14].
OKane P, Sezer S, McLaughlin K. Obfuscation: the hidden malware. IEEE Secur Priv Sep. 2011;9(5):41–7.
Rad B, Masrom M, Ibrahim S. Opcodes histogram for classifying metamorphic portable executables malware. In: ICEEE; September 2012. p. 209–13.
Raschke T. The new security challenge: endpoints. © International Data Corporation; 2005.
Runwal N, Low RM, Stamp M. Opcode graph similarity and metamorphic detection. J Comput Virol May 2012;8(1–2):37–52.
Santos I, Brezo F, Ugarte-Pedrero X, Bringas PG. Opcode sequences as representation of executables for data-mining-based unknown malware detection. Inf Sci 2013;231(0):64–82 [data Mining for Information Security].
Shabtai A, Moskovitch R, Feher C, Dolev S, Elovici Y. Detecting unknown malicious code by applying classification techniques on opcode patterns. Secur Inform 2012;1(1):1–22.
Shanmugam G, Low RM, Stamp M. Simple substitution distance and metamorphic detection. J Comput Virol Hacking Tech 2013;9(3):159–70.
Song F, Touili T. Efficient malware detection using model-checking. In: Giannakopoulou D, Mry D, editors. FM: Formal Methods. Vol. 7436 of Lecture Notes in Computer Science. Berlin Heidelberg: Springer; 2012a. p. 418–33.
Song F, Touili T. Pushdown model checking for malware detection. In: Flanagan C, Knig B, editors. Tools and Algorithms for the Construction and Analysis of Systems. Vol. 7214 of Lecture Notes in Computer Science. Berlin Heidelberg: Springer; 2012b. p. 110–25.
Symantec C. Internet security threat report – 2011 Trends. © Symantec Corporation; April 2012b. p. 17.
Symantec C. Norton cybercrime report. ©Symantec Corporation; August 2012b. http://www.symantec.com.
Szor P. The art of computer virus research and defense. Addison-Wesley Professional; 2005.
Toderici A, Stamp M. Chi-squared distance and metamorphic virus detection. J Comput Virol 2013:1–14.
Vinod P, Laxmi V, Gaur M, Chauhan G. MOMENTUM: metamorphic malware exploration techniques using MSA signatures. IIT; March 2012. p. 232–7.
Vinod P, Laxmi V, Gaur MS, Kumar GP, Chundawat YS. Static CFG analyzer for metamorphic malware code. In: Proceedings of the 2nd International Conference on Security of Information and Networks. SIN'09. New York, NY, USA: ACM SIGSAC; 2009. p. 225–8.
Weisstein EW. Chi-Squared Test. In: MathWorld – A Wolfram Web Resource. Wolfram Research Inc. http://mathworld.wolfram.com/Chi-SquaredTest.html, [last accessed 30.10.14].
Wong W, Stamp M. Hunting for metamorphic engines. J Comput Virol 2006;2:211–29.
Yin H, Song D. Privacy-breaching behavior analysis. In: Automatic malware analysis. SpringerBriefs in Computer Science. New York: Springer; 2013. p. 27–42.

Shahid Alam is currently a PhD student in the Computer Science Department at University of Victoria, BC. He received his MASc degree from Carleton University, Ottawa, ON, in 2007. He has more than 5 years of experience working in the software industry. His research interests include programming languages, compilers, software engineering and binary analysis for software security. Currently he is looking into applying compiler, binary analysis and artificial intelligence techniques to automate and optimize malware analysis and detection.

Nigel Horspool is a Professor of computer science at the University of Victoria. He received an M.Sc degree and a Ph.D. in Computer Science from the University of Toronto in 1972 and 1976, respectively. From 1976 until 1983, he was an Assistant Professor and then an Associate Professor in the School of Computer Science at McGill University in Montreal. He joined the Computer Science Department at the University of Victoria in 1983. His research interests are mostly concerned with the compilation and implementation of programming languages. He is the author of the book *C Programming in the Berkeley UNIX Environment* and co-author of the book *C# Concisely*.

Issa Traore has been with the faculty of the Electrical and Computer Engineering Department of the University of Victoria since 1999, where he is currently a Professor. Dr. Traore is also the founder and Director of the Information Security and Object Technology (ISOT) Lab (www.isot.ece.uvic.ca). He obtained in 1998 a PhD in Software Engineering from the Institute Nationale Polytechnique of Toulouse, France. His main research interests are biometrics technologies, intrusion detection systems, and software security.

Ibrahim Sogukpinar received his B.Sc. degree in Electronic and Communications Engineering from Technical University of Istanbul in 1982, and his M.Sc. degree in Computer and Control Engineering from Technical University of Istanbul in 1985. He received his Ph.D. degree in Computer and Control Engineering from Technical University of Istanbul in 1995. Currently he is the head of the Computer Engineering Department at Gebze Institute of Technology. His main research areas are information security, computer networks, applications of information systems and computer vision.