Abstract—Because of the financial and other gains attached with the growing malware industry, there is a need to automate the process of malware analysis and provide real-time malware detection. To hide a malware, obfuscation techniques are used. One such technique is metamorphism encoding that mutates the dynamic binary code and changes the opcode with every run to avoid detection. This makes malware difficult to detect in real-time and generally requires a behavioral signature for detection. In this paper we present a new framework called MARD for Metamorphic Malware Analysis and Real-Time Detection, to protect the end points that are often the last defense, against metamorphic malware. MARD provides: (1) automation (2) platform independence (3) optimizations for real-time performance and (4) modularity. We also present a comparison of MARD with other such recent efforts. Experimental evaluation of MARD achieves a detection rate of 99.6% and a false positive rate of 4%.

Keywords— End Point Security; Metamorphism; Malware Analysis and Detection; Control Flow Analysis; Automation;

I. INTRODUCTION

End point protection 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 ITU (International Telecommunications Union) [14] show that the number of Internet users in the world have increased from 20% in 2006 to 35% (almost 2 billion in total) in 2011. According to the 2011 Symantec Internet security threat report [8] 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. With these increases and the anticipated future increases, these end points pose a new security challenge [23] to the security professionals and researchers in industry and in academia, to devise new methods and techniques for malware detection and protection.

A broad definition of malware, also called malicious code, is used in the literature that includes viruses, worms, spywares and trojans. Here we use one of the earliest definitions by Gary McGraw and Greg Morrisett [19]: Malicious code is any code added, changed, or removed from a software system in order to intentionally cause harm or subvert the intended function of the system. A malware carries out activities such as setting up a back door for a bot, setting up a keyboard logger and stealing personal information etc.

Antimalware software detects and neutralizes the effects of a malware. There are two basic detection techniques [13]: anomaly-based and signature-based. Anomaly-based detection technique uses the knowledge of the behavior of a normal program to decide if the program under inspection is malicious or not. Signature-based detection technique uses the characteristics of a malicious program to decide if the program under inspection is malicious or not. Each of the techniques can be performed statically (before the program executes), dynamically (during or after the program execution) or both statically and dynamically (hybrid).

Detecting whether a given program is a malware is an undecidable problem [6], [16]. Antimalware software detection techniques are limited by this theoretical result. Malware writers exploit this limitation to avoid detection.

In the early days the malware writers were hobbyists but now the professionals have become part of this group because of the financial gains [5] attached to it.

One of the basic techniques used by malware writers is obfuscation [15]. Such a technique obscures a code to make it difficult to understand, analyze and detect malware embedded in the code.

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. Currently there are three categories of obfuscation techniques [21]: 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 [21] 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 products.

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 opcode with each run of the infected program and does not use any encryption or decryption. The malware never keeps the same sequence of opcodes in the memory. This is also called *dynamic code obfuscation*. There are two kinds of metamorphic malware defined in [21] based on the channel of communication used: Closed-world malware, that do not rely on external communication and can generate the newly mutated code using either a binary transformer or a metalanguage. Open-world malware, that can communicate with other sites on the Internet and update themselves with new features.

As is clear from the above discussion out of the three malware groups mentioned above, metamorphic malware are getting more complex and pose a special threat and new challenges to the end point security. We propose in this paper a new framework for real-time detection of metamorphic malware using *subgraph isomorphism*. The proposed framework is named MARD for Metamorphic Malware Analysis and 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 provides protection from malware in two ways:

1) They can provide protection by detecting a 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. The general problem of detecting metamorphic malware is NP-complete [26]. That means there exist some metamorphic malware whose detection is NP-complete.

The remainder of the paper is organized as follows. Section II describes MARD in detail. In Section III we show how parallelization and CFG reduction reduces the runtime of MARD. In Section IV we evaluate the performance of MARD using an existing malware dataset. Section V describes the most recent efforts of detecting metamorphic malware and compares them to MARD. We conclude in Section VI.

## II. The MARD Framework

Our primary goal is to detect automatically in real-time both known and unknown malware. We present in this section the design principles underlying the MARD framework and describes the proposed malware detection approach.

### A. MARD Design

Most existing malware use binaries to infiltrate a computer system. Binary analysis is the process of automatically analysing 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 [3], a new intermediate language for malware analysis named MAIL. MAIL is the acronym for Malware Analysis Intermediate Language. In [2], we explain in detail with examples how to translate a x86 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. We then build a CFG (control flow graph) of the annotated MAIL program.

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

**Platform independence:** To make modern compilers [1], [20] platform independent they are divided into two major components: a *front end* and a *back end*. The design of the MARD framework follows the same principle. In compilers the same C++ *front end* can be used with different *back ends* to generate code for different platforms such as x86, ARM and PowerPC etc. In the case of MARD the same *back end* can be used with different *front ends* to detect malware for different platforms (Windows and Linux etc). For example, we can implement a *front end* for the PE (Windows executable) files and another *front end* for the ELF (Linux executable) files. Both these *front ends* generate their output in our intermediate language (i.e. MAIL) that is used by the *back end*. Programs compiled for different architectures such as Intel x86 and ARM (the two most popular architectures) can be translated to MAIL. Examples of such translations are described in [2]. 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 CFG reduction to optimize the runtime of MARD.

### B. Malware Detection

After a program sample is translated to MAIL, an annotated control flow graph (CFG) for each function in the program is built. Instead of using one large CFG as signature, we divide a program into smaller CFGs, with one CFG per function. A program signature is then represented by the set of corresponding (smaller) CFGs. 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 CFGs involved in a malware signature match with the signature of a program then the program is classified as a malware. Using a simple threshold-based classifier also considerably reduces the runtime of MARD and facilitates real-time detection.

Figure 1 gives an overview of the MARD framework. First, a training dataset is built, also called Malware Templates in Figure 1, using the malware training samples. After a program (sample) is translated to MAIL and to ACFGs (annotated CFGs) the Similarity Detector (Figure 1) detects the presence of malware in the program, using the Malware Templates as described above. All the steps as shown in Figure 1 are completely automated.

Similarity detector in MARD is comprised of two sub-components: a subgraph matching component and a pattern matching component. We describe these two components in the following.

1) **Subgraph Matching**: In our malware detection approach, graph matching is defined in terms of subgraph isomorphism. Given the input of two graphs, subgraph isomorphism determines if one of the graphs contains a subgraph that is isomorphic to the other graph. Generally, subgraph isomorphism is an NP-Complete problem [7]. A

![Figure 1. High Level Overview of the MARD Framework](image)

After the binary analysis we obtain a set of CFGs (each corresponding to separate function) of a program. To detect if a program contains a malware we compare the CFGs of the program with the CFGs of known malware samples from our training database. If a percentage of the CFGs of the program, greater than a predefined threshold, match one or several of the CFGs of a malware sample (from the database) then the program will be classified as a malware.

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. Every statement in MAIL is assigned a pattern as explained in [3]. If a CFG of a malware sample matches with a CFG of a program (i.e. the two CFGs are isomorphic), then we further use the patterns, assigned to MAIL statements, to match each statement in the matching nodes of the two CFGs. 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.

### Figure 1. High Level Overview of the MARD Framework

**Front End**
- MAIL (Malware Analysis Intermediate Language)
- **Back End**
- MAIL
- ACFG
- **Detector**
- Similarity
- Report
- **Generator**
- ACFG
- Template
- **Optimizer**
- Template
- **Disassembler**
- Unpacked binary
- **Graph Miner**
- Mined graph
- **Report**
- **Optimizer**
- Optimized code
- **Report Generator**
- MAIL
- **Generator**
- MAIL
- **Unpacker**
- Unpacked binary
- Assembly instructions
- **Template**
- **Report**
- **Template**

1. MAIL = Malware Analysis Intermediate Language
2. ACFG = Annotated Control Flow Graph

**NOTE**: The component "Unpacker" is not implemented in this version of the Malware Detector.
III. Runtime Optimization

There are two components in the MARD framework that consumes most of its runtime; one is the Translator (part of the MAIL Generator in Figure 1) and the other is the Subgraph Matching (part of the Similarity Detector in Figure 1). The Translator reads each assembly instruction one by one and translates it to the MAIL language. The translation time increases with the size of the binary, and provide limited avenue for optimization. The Subgraph Matching component matches the CFG 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 of MARD, namely, parallelization and CFG reduction, described in the following.

A. Parallelization

Multicore processors are becoming popular nowadays. All the current desktops, laptops and even the energy efficient small mobile devices contain a multicore processor. Intel has recently announced its Single-Chip Cloud Computer [18], a processor with 48 cores on a chip. Keeping in view this 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 carried out an experiment using different number of threads ranging from 2 to 250. We used 250 malware samples and 30 benign applications each with a different size of CFG. The main reason for this experiment was to develop an empirical model to compute the number of threads to be used in the Subgraph Matching component. The experiment was run on machines with 2 and 4 Cores. Details of the machines used and the results of this experiment are shown in Table I. The percentage of CPU utilization on average was 3 times more with threads than without threads. This is also confirmed by the improvement in the runtime.

<table>
<thead>
<tr>
<th>Number of Threads</th>
<th>Runtime Reduced By</th>
<th>Number of Threads</th>
<th>Runtime Reduced By</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3.20 times</td>
<td>4</td>
<td>3.98 times</td>
</tr>
<tr>
<td>4</td>
<td>5.92 times</td>
<td>8</td>
<td>4.11 times</td>
</tr>
<tr>
<td>8</td>
<td>7.20 times</td>
<td>32</td>
<td>5.95 times</td>
</tr>
<tr>
<td>16</td>
<td>5.72 times</td>
<td>64</td>
<td>7.76 times</td>
</tr>
<tr>
<td>32</td>
<td>5.64 times</td>
<td>128</td>
<td>7.53 times</td>
</tr>
<tr>
<td>250</td>
<td>5.41 times</td>
<td>250</td>
<td>6.37 times</td>
</tr>
<tr>
<td>8</td>
<td>14.87 times</td>
<td>64</td>
<td>14.53 times</td>
</tr>
</tbody>
</table>

Machines used:
Intel Core i5 CPU M 430 (2 Cores) @ 2.27 GHz with 4GB of RAM and Windows 8 Professional installed
Intel Core 2 Quad CPU Q6700 (4 Cores) @ 2.67 GHz with 4GB of RAM and Windows 7 Professional installed

As we increased the number of threads the reduction in the runtime also increased upto a certain number (8 in 2 Cores and 64 in 4 Cores) of threads, after which the reduction in the runtime started decreasing. As we increased the number of benign samples from 30 to 1387 (the last row in Table I) the runtime only reduced by almost 2 times. Beside other factors such as the number of CPUs/Cores and the memory available to the application, the other reasons for this reduction in the runtime are the fact that more time was spent on thread management and the increased sharing of information among threads.

Based on this experiment we developed Equation 1 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.
where NC = Number of CPUs/Cores

B. CFG Reduction

One of the advantages of using MAIL is that it provides patterns 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 a CFG (it is possible for a CFG 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 a CFG for runtime optimization we carried out CFG reduction, also called CFG shrinking. We reduce the number of blocks in a CFG 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 a CFG, 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.

Figures 3, 4 and 5 show examples of CFGs, from the dataset described in this paper, before and after shrinking. Figures 3 and 4 are CFGs of functions of two different malware samples and Figure 5 is a CFG of a function of a benign sample. The shrinking does not change the shape of a graph. As we can see in the Figures, the shapes of the graphs before and after shrinking are the same. More of these examples are available at http://www.cs.uvic.ca/~salam/PhD/cfgs.html.

![Example of a CFG, of one of the functions of one of the samples of the MWOR class of malware, before and after shrinking. The CFG has been reduced from 92 nodes to 47 nodes.](image)

![Example of a CFG, of one of the functions of the Windows disk free space utility program df.exe, before and after shrinking. The CFG has been reduced from 894 nodes to 283 nodes.](image)

(1) \[ Threads = (NC)^3 \]

We were able to substantially reduce the number of nodes per CFG (in total a 90.6% reduction), as shown in Table III. This reduced the runtime of MARD on average by 2.7 times (for smaller dataset) and 100 times (for larger dataset), and still achieved a detection rate of 99.6% with a false positive rate of 4% as latter shown in Table VI.

IV. MARD PERFORMANCE

We carried out an empirical study to analyse the correctness and efficiency of MARD. We present, in this section, the evaluation metrics, the empirical study, obtained results and analysis.

A. Performance Metrics

Before evaluating the performance of MARD, we first define the following performance 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:

\[
Precision = \frac{TP}{TP + FP} \quad \text{Accuracy} = \frac{TP + TN}{P + N} \quad (2)
\]

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

\[
MMP = \frac{1}{n} \sum_{i=1}^{n} Precision_i \quad (3)
\]

\[
MMA = \frac{1}{n} \sum_{i=1}^{n} Accuracy_i \quad (4)
\]

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

\[
TPR = \frac{TP}{P} \quad (5)
\]

\[
FPR = \frac{FP}{N} \quad (6)
\]

Receiver Operator Characteristic (ROC) Curve is a graphical plot of depicting performance of a binary classifier. It is a two dimensional plot, where TPR is plotted on the Y-axis and FPR is plotted on the X-axis, and hence depicts the trade-offs between benefits (TP) and costs (FP). We want a higher TPR and a lower FPR, so a point in ROC space to the top left corner is desirable.

**B. Dataset**

The dataset used for the experiments consists of total 3350 sample Windows programs. Out of the 3350 programs, 1020 are metamorphic malware samples collected from three different resources [22], [24], [17], and the other 2330 are benign programs. The dataset distribution based on the number of CFGs for each program sample is shown in Tables II and III respectively. The normalizations carried out are removal of NOP and other junk instructions.

The dataset contains a variety of programs with CFGs ranging from simple to complex for testing. As shown in Table II the number of CFGs per malware sample ranges from 2 to 1272 and the number of CFGs per benign program ranges 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 (i.e. they cannot be executed) and as a result they have 0 node CFGs. The sizes of these CFGs are shown in Table III. The size of the CFGs of the malware samples range from 1 to 301 nodes, and the size of the CFGs of the benign programs range from 1 to 521 nodes.

The 1020 malware samples belong to the following three metamorphic family of viruses: Next Generation Virus Gen-

<table>
<thead>
<tr>
<th>Table II</th>
<th>DATASET DISTRIBUTION BASED ON THE NUMBER OF CONTROL FLOW GRAPHS (CFGs) FOR EACH PROGRAM SAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malware samples (1020)</td>
<td>Benign program samples (2330)</td>
</tr>
<tr>
<td>Number of CFGs</td>
<td>Number of Samples</td>
</tr>
<tr>
<td>2</td>
<td>250</td>
</tr>
<tr>
<td>5 – 32</td>
<td>204</td>
</tr>
<tr>
<td>33 – 57</td>
<td>222</td>
</tr>
<tr>
<td>58 – 84</td>
<td>133</td>
</tr>
<tr>
<td>85 – 133</td>
<td>105</td>
</tr>
<tr>
<td>140 – 249</td>
<td>94</td>
</tr>
<tr>
<td>133 – 1272</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table III</th>
<th>DATASET DISTRIBUTION BASED ON THE SIZE (NUMBER OF NODES) FOR EACH CFG AFTER NORMALIZATION AND SHRINKING</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malware samples (1020)</td>
<td>Benign program samples (2330)</td>
</tr>
<tr>
<td>Number of Nodes</td>
<td>Number of CFGs</td>
</tr>
<tr>
<td>1 – 10</td>
<td>60284</td>
</tr>
<tr>
<td>11 – 20</td>
<td>1652</td>
</tr>
<tr>
<td>21 – 39</td>
<td>2177</td>
</tr>
<tr>
<td>41 – 69</td>
<td>288</td>
</tr>
<tr>
<td>70 – 96</td>
<td>207</td>
</tr>
<tr>
<td>104 – 183</td>
<td>254</td>
</tr>
<tr>
<td>221 – 301</td>
<td>2</td>
</tr>
</tbody>
</table>

Runtime on average reduced by 2.7 times (for smaller dataset) and 100 times (for larger dataset) respectively. The normalizations carried out are removal of NOP and other junk instructions.

The dataset contains a variety of programs with CFGs ranging from simple to complex for testing. As shown in Table II the number of CFGs per malware sample ranges from 2 to 1272 and the number of CFGs per benign program ranges 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 (i.e. they cannot be executed) and as a result they have 0 node CFGs. The sizes of these CFGs are shown in Table III. The size of the CFGs of the malware samples range from 1 to 301 nodes, and the size of the CFGs of the benign programs range from 1 to 521 nodes.

The 1020 malware samples belong to the following three metamorphic family of viruses: Next Generation Virus Gen-
eration Kit (NGVCK) (http://vxheaven.org/vx.php?id=tn02), Second Generation Virus Generator (G2) (http://vxheaven.org/vx.php?id=tg00) and Metamorphic Worm (MWOR) [17] generated by metamorphic generator. NGVCK and MWOR family of viruses are further divided into two and seven Classes respectively. This Class distribution is shown in Table IV.

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

MWOR uses two morphing techniques: dead code insertion and equivalent instruction substitution. Padding ratio is the ratio of the number of dead code instructions to the core instructions of the malware. Padding ratio of 0.5 means that the malware has half as much dead code instructions as core instructions [17].

This variety of CFGs and malware Classes in the samples provides a good testing platform for the proposed malware detection framework.

C. Threshold Computation

As explained above, to classify a program as benign or malware we compare to some predefined threshold the percentage of its CFGs that match malware CFGs from the training set. To compute this threshold value we carried out an experiment with MARD and selected 1387 sample Windows programs for malware analysis and detection. Out of these 1387 programs, 250 are metamorphic malware samples, and the other 1137 are benign programs. We ran the experiment with different values of the threshold ranging from 20% to 90%. Table V lists the obtained results. As it can be noted the best results are obtained for threshold values of 20% – 25%. We decided to use 25% as threshold value for malware detection.

<table>
<thead>
<tr>
<th>Threshold (%)</th>
<th>DR (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>99.2</td>
</tr>
<tr>
<td>25</td>
<td>99.2</td>
</tr>
<tr>
<td>30</td>
<td>93.2</td>
</tr>
<tr>
<td>40</td>
<td>86.4</td>
</tr>
<tr>
<td>50</td>
<td>82.8</td>
</tr>
<tr>
<td>60</td>
<td>76</td>
</tr>
<tr>
<td>70</td>
<td>76</td>
</tr>
<tr>
<td>80</td>
<td>76</td>
</tr>
<tr>
<td>90</td>
<td>76</td>
</tr>
</tbody>
</table>

D. Empirical Study

In this Section we discuss the experiments, and present the performance results obtained. The experiments were run on the following machine: Intel Core i5 CPU M 430 (2 Cores) @ 2.27 GHz with 4GB RAM, operating system Windows 8 professional.

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.

To find the runtime improvement by CFG reduction, both these experiments were carried out with and without CFG reduction. We were able to complete all the runs (10 times) with and without CFG reduction for the smaller dataset. We completed all the runs (5 times) with CFG reduction with the larger dataset but completed only one run (took over 34 hours to complete) without CFG reduction. This is one of the other reasons to use a smaller dataset with a larger dataset, to get accurate results for runtime improvement. In the following two Sections we present and describe these two experiments with the results.

1) Experiment with Smaller Dataset Using 10-fold Cross Validation: Out of 3350 Windows programs we selected randomly 1351 Windows programs. Out of these 1351 programs, 250 are metamorphic malware samples belonging to Classes NGVCK_1 and NGVCK_2, and the other 1101 are benign programs.

The 10-fold cross validation was conducted by selecting 25 malware samples out of the 250 malware to train our detector. The remaining 225 malware samples along with the 1101 benign programs were then used to test the detector. These two steps were repeated 10 times and each time...
different set of 25 malware samples were selected for training and the remaining samples for testing. The overall performance results were obtained by averaging the results obtained in the 10 different runs.

Using a threshold value of 25%, we conducted further evaluation by increasing the size of the training set from 25 samples to 125 malware samples (50% of the malware samples). The obtained results are listed in Table VI. The DR improved from 94% when the size of the training set is 25 to 99.6% when we used a training dataset of 125 samples.

<table>
<thead>
<tr>
<th>Training set size</th>
<th>DR</th>
<th>FPR</th>
<th>MMP</th>
<th>MMA</th>
<th>Real-Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>94%</td>
<td>3.1%</td>
<td>0.86</td>
<td>0.96</td>
<td>✓</td>
</tr>
<tr>
<td>125</td>
<td>99.6%</td>
<td>4%</td>
<td>0.85</td>
<td>0.97</td>
<td>✓</td>
</tr>
</tbody>
</table>

Real-time here means the detection is fully automatic and finishes in a reasonable amount of time. On average it took MARD 15.2037 seconds with CFG shrinking and 40.4288 seconds without CFG 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 CFG shrinking.

2) Experiment with Larger Dataset Using 5-fold Cross Validation: The 5-fold cross validation was conducted by selecting 204 malware samples out of the 1020 malware to train our detector. The remaining 816 malware samples along with the 2330 benign programs in our dataset were then used to test the detector. These two steps were repeated 5 times and each time different set of 204 malware samples were selected for training and the remaining samples for testing. The overall performance results were obtained by averaging the results obtained in the 5 different runs.

<table>
<thead>
<tr>
<th>Training set size</th>
<th>DR</th>
<th>FPR</th>
<th>MMP</th>
<th>MMA</th>
<th>Real-Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>204</td>
<td>97%</td>
<td>4.3%</td>
<td>0.91</td>
<td>0.96</td>
<td>✓</td>
</tr>
<tr>
<td>510</td>
<td>98.9%</td>
<td>4.5%</td>
<td>0.91</td>
<td>0.97</td>
<td>✓</td>
</tr>
</tbody>
</table>

Real-time here means the detection is fully automatic and finishes in a reasonable amount of time. On average it took MARD 946.5824 seconds with CFG shrinking and over 125400 seconds (over 34 hours) without CFG 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 CFG shrinking. The time (over 34 hours) reported is just for one run of the experiment without CFG shrinking.

Using a threshold value of 25%, we conducted further evaluation by increasing the size of the training set from 204 samples to 510 malware samples (50% of the malware samples). The obtained results are listed in Table VI. The DR improved from 97% when the size of the training set is 204 to 98.9% when we used a training dataset of 510 samples.

3) ROC Graphs: ROC graph plots for both the above experiments are shown in Figure 6. As expected both of them are in the top left corner of the graph. The ROC for larger dataset has more precision, i.e; more number of ROC points are closer to each other and this is also confirmed by the MMP listed in Table VII.

![ROC graph plots](image)

Figure 6. ROC graph plots

V. RELATED WORK

This section discusses the previous research efforts on detecting metamorphic malware.

A recent effort presented in [29] uses dynamic taint analysis (DTA) to automatically detect if an unknown sample exhibits malicious behavior or not. A so-called taint engine
tracks the flow of information of the whole system, which is compared against a set of defined policies. A proof of concept was implemented as a plugin of an emulator, and evaluated using 46 malware and 56 benign samples for testing the system. The evaluation yielded DR=100% and FPR=3%.

However, while the proposed detector provides a fine-grained whole system analysis, it is not completely automatic; human input is needed to make more accurate detections, and to specify the policies.

Furthermore, as the detector runs in an emulator, it probably takes a considerable amount of time for detection. Therefore it cannot be used for real-time malware detection.

In [25] the authors use model-checking to detect metamorphic malware. Model-checking techniques check if a given model meets a given specification. 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. The proposed approach was evaluated with 200 malware and 8 Windows benign programs, yielding DR=100% with FPR=12.5%.

Model-checking is time consuming and sometimes it can run out of memory. Likewise the proposed approach is not suitable for real-time detection.

A technique described in [12] uses value set analysis (VSA) [4] for detecting metamorphic malware, where malware binaries are run and traced inside a controlled environment to collect register values. Experimental evaluation using a dataset consisting of 826 files contaminated with 7 specimens of malware and 385 benign files, achieved DR=98% and FPR=2.9%.

In [11], a technique is presented that uses 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 evaluation of the approach was conducted using a dataset consisting of 2550 benign and 3639 malware files, and achieved in the best case DR=98.4% and FPR=2.7%. The proposed system is not fully automated and cannot be used as a real-time detector.

Table VIII gives a summary of all the systems’ research efforts discussed above and a few additional detectors not covered here due to space limitation. 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.

VI. CONCLUSION AND FUTURE WORK

In this paper, we have presented a new metamorphic malware detection framework named MARD 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. Table VIII reports the best DR results achieved by these detectors. Out of the 10 systems, MARD 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. Currently we are carrying out further research into using similar techniques for web malware analysis and detection.

REFERENCES


Table VIII
SUMMARY AND COMPARISON WITH MARD OF THE METAMORPHIC MALWARE ANALYSIS AND DETECTION SYSTEMS DISCUSSED IN SECTION V

<table>
<thead>
<tr>
<th>System</th>
<th>Analysis Type</th>
<th>DR</th>
<th>FPR</th>
<th>Data Set Size</th>
<th>Real-Time</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>MARD</td>
<td>Static</td>
<td>98.9%</td>
<td>4.5%</td>
<td>2330 / 1020</td>
<td>✓</td>
<td>Win &amp; Linux 64</td>
</tr>
<tr>
<td>API-CFG [9], [10]</td>
<td>Static</td>
<td>97.53%</td>
<td>1.97%</td>
<td>2140 / 2305</td>
<td>✗</td>
<td>Win 32</td>
</tr>
<tr>
<td>Call-Gram [11]</td>
<td>Static</td>
<td>98.4%</td>
<td>2.7%</td>
<td>3234 / 3256</td>
<td>✗</td>
<td>Win 32</td>
</tr>
<tr>
<td>Chi-Squared [27]</td>
<td>Static</td>
<td>89.74%</td>
<td>~10%</td>
<td>40 / 200</td>
<td>✗</td>
<td>Win 32</td>
</tr>
<tr>
<td>DTA [29]</td>
<td>Dynamic</td>
<td>100%</td>
<td>3%</td>
<td>56 / 42</td>
<td>✗</td>
<td>Win 64</td>
</tr>
<tr>
<td>Histogram</td>
<td>Static</td>
<td>100%</td>
<td>0%</td>
<td>40 / 60</td>
<td>✗</td>
<td>Win 32</td>
</tr>
<tr>
<td>Model-Checking [25]</td>
<td>Static</td>
<td>100%</td>
<td>1%</td>
<td>8 / 200</td>
<td>✗</td>
<td>Win 32</td>
</tr>
<tr>
<td>MSA [28]</td>
<td>Static</td>
<td>91%</td>
<td>52%</td>
<td>150 / 1209</td>
<td>✗</td>
<td>Win 32</td>
</tr>
<tr>
<td>Opcode-Graph [24]</td>
<td>Static</td>
<td>100%</td>
<td>1%</td>
<td>41 / 200</td>
<td>✗</td>
<td>Win 32</td>
</tr>
<tr>
<td>VSA [12]</td>
<td>Dynamic</td>
<td>98%</td>
<td>2.9%</td>
<td>385 / 826</td>
<td>✗</td>
<td>Win 64</td>
</tr>
</tbody>
</table>

The perfect results of 100% and 0% should be validated with more number of samples than tested in the papers. 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.


