Annotated Control Flow Graph for Metamorphic Malware Detection

SHAHID ALAM1*, ISSA TRAORE2 AND IBRAHIM SOGUKNAR3

1Department of Computer Science, University of Victoria, Victoria, BC, Canada
2Department of Electrical and Computer Engineering, University of Victoria, Victoria, BC, Canada
3Department of Computer Engineering, Gebze Institute of Technology, Gebze, Kocaeli, Turkey
*Corresponding author: salam@cs.uvic.ca

Metamorphism is a technique that mutates the binary code using different obfuscations and never keeps the same sequence of opcodes in the memory. This stealth technique provides the capability to a malware for evading detection by simple signature-based (such as instruction sequences, byte sequences and string signatures) anti-malware programs. In this paper, we present a new scheme named Annotated Control Flow Graph (ACFG) to efficiently detect such kinds of malware. ACFG is built by annotating CFG of a binary program and is used for graph and pattern matching to analyse and detect metamorphic malware. We also optimize the runtime of malware detection through parallelization and ACFG reduction, maintaining the same accuracy (without ACFG reduction) for malware detection. ACFG proposed in this paper: (i) captures the control flow semantics of a program; (ii) provides a faster matching of ACFGs and can handle malware with smaller CFGs, compared with other such techniques, without compromising the accuracy; (iii) contains more information and hence provides more accuracy than a CFG. Experimental evaluation of the proposed scheme using an existing dataset yields malware detection rate of 98.9% and false positive rate of 4.5%.

Keywords: annotated control flow graph; static binary analysis; malware detection; optimizations

Received 24 August 2014; revised 7 November 2014
Handling editor: Steven Furnell

1. INTRODUCTION

End point security is often the last defense against a security threat. An end point can be a desktop, server, laptop, kiosk or mobile device that connects to a network (Internet). Recent statistics by the International Telecommunications Union [1] 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 [2]. 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.

In the early days, malware writers were hobbyists, however professionals have now become part of this group, either for the financial gains [3] attached to it, or for political and military reasons. One of the basic techniques used by a malware writer is obfuscation [4]. 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 detectable by simple signature-based detectors. These signature-based detectors work on simple signatures such as byte sequences, instruction sequences and string signatures (pattern of a malware that uniquely identifies it). However, they lack information about the semantics or behavior of the malicious program. To counter these detectors the obfuscation techniques evolved, producing metamorphic malware that use stealthy mutation techniques [4–8]. Likewise, to address effectively the challenges posed by metamorphic malware, we need to develop new methods and techniques to analyze the behavior of a program and make a better detection decision with few false positives.
The main goal of this paper is to extract behavioral and structural information from a program to detect the presence of malware in the program. Control flow analysis (CFA) [9, 10] is one of the techniques used in compilers for program analysis and optimization. The CFA of a program is expressed as a control flow graph (CFG). The CFG of a program represents all the paths that can be taken during the program execution. Current techniques [11–20] that use CFG for malware detection are compute intensive (others have poor detection rate) and cannot handle malware with smaller CFGs.

It is difficult to write a new metamorphic malware [21] and in general malware writers reuse old malware. To hide detection the malware writers change the obfuscations (syntax) more than the behavior (semantic) of such a new metamorphic malware. If an unknown metamorphic malware uses all or some of the same class of behaviors as are used by the training dataset (set of old metamorphic malware) then it is possible to detect these types of malware.

On the assumption and the motivation described above, we propose a new technique named Annotated Control Flow Graph (ACFG) that can enhance the detection of metamorphic malware and can handle malware with smaller CFGs. We also optimize the runtime of malware detection through parallelization and ACFG reduction, making the comparison (matching with other ACFGs) faster, while maintaining 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.

ACFG1 proposed in this paper: (i) captures the control flow semantics of a program; (ii) provides a faster matching of ACFGs and can handle malware with smaller CFGs, in comparison with other such techniques, without compromising the accuracy; (iii) contains more information and hence provides more accuracy than a CFG.

The rest of the paper is structured as follows. In Section 2, we discuss related research efforts that use CFG for metamorphic malware detection. In Section 3, we describe in detail the patterns used for CFG annotation, illustrate how a binary program can be translated to an ACFG and introduce our malware detection approach. In Section 4, we show how parallelization and ACFG reduction reduces the runtime of a malware detector. In Section 5, we evaluate the performance of ACFG using an existing malware dataset and compare it with other such techniques. We finally conclude in Section 6.

2. RELATED WORKS

This section discusses the previous research efforts that use CFG for metamorphic malware detection. We cover only academic research efforts that claim or will extend their detector to detect metamorphic malware.

In [13], the authors use an intermediate language called CFGO-IL to simplify transformation of a program in the x86 assembly language to a CFG. After translating a binary program to CFGO-IL, the program is optimized to make its structure simpler. The optimizations also remove various malware obfuscations from the program. These optimizations include dead code elimination, removal of unreachable branches, constant folding and removal of fake conditional branches inserted by malware. The authors developed a prototype malware detection tool using CFGO-IL that take advantage of the optimizations and the simplicity of the language. The size of a CFGO-IL program tends to increase compared with the original assembly program and hence the CFG of the program. As mentioned in the paper [13], the technique proposed is not able to discriminate malware from a benign program if the size of the CFG is less than 100 nodes. ACFG can handle such smaller CFGs.

The method described in [14] uses model-checking to detect metamorphic malware. Model-checking techniques check if a given model meets a given specification. The paper [14] used a pushdown system to build a model that takes into account the behavior of the stack. They use IDA Pro [24] (a close source disassembler) to build a CFG of a binary program. This CFG contains information about the register and the memory location values at each control point of the program, and is translated into a pushdown system. The pushdown system stores the control points and the stack of the program.

Model-checking is time consuming and it can sometimes run out of memory as was the case with the previous technique [25] by the same authors. The times reported range from a few seconds (for a program consisting of 10 instructions) to over 250 s (for a program consisting of 10,000 instructions). Real-life applications are much larger than the samples tested.

In [15], the authors compare the instructions of a basic block in CFG of the original malware, with that of its variants. The comparison is performed using the longest common subsequence [26]. Unlike, other techniques they do not compare the shape of two CFGs, and hence is more time efficient, but, as the preliminary results show, produce a greater number of false positives. This work is still in progress and therefore the complete results are not available.

The technique described in [16] is the closest technique to the work presented in this paper. After disassembling a program they perform normalization on the disassembled program. They then build an interprocedural CFG of the normalized disassembled program. For malware detection, CFG of a benign program is checked to find if it contains a subgraph that is isomorphic to CFG of a malware program. The results reported in the paper are preliminary and are incomplete.

The method presented in [11, 12] uses CFG for visualizing the control structure and representing the semantic aspects of a

---

1The work presented in this paper is an expansion of the previously published work in [22, 23].
Annotated Control Flow Graph for Metamorphic Malware Detection

3. MALWARE ANALYSIS AND DETECTION USING ACFG

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

**Definition 1.** A basic block is a maximal sequence of instructions with a single entry and a single exit point. There is no instruction after the first instruction that is the target instruction of a jump instruction, and only the last instruction can jump to a different block. Instructions that can start a basic block include the following. The first instruction, target of a branch or a function call, and a fall through instruction, i.e. an instruction following a branch, a function call or a return instruction. Instructions that can end a basic block include the following. A conditional or unconditional branch, an instruction following a branch or a function call or a return instruction.

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

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

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

3.1. Patterns for Annotation

We have recently proposed a new intermediate language named MAIL (Malware Analysis Intermediate Language) [23]. There are 21 Patterns in the MAIL language. They are used to build an ACFG of a program that can be used for graph and pattern matching during malware detection. In this section we describe in detail these patterns as follows:

- **ASSIGN:** An assignment statement, e.g. EAX=EAX+ECX;
- **ASSIGN_CONSTANT:** An assignment statement including a constant, e.g. EAX=EAX+0x10;
- **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. For example, 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 including a constant, e.g. EAX = [SP=SP–0x1];
- **STACK_CONSTANT:** A stack statement including a constant, e.g. [SP=SP+0x1] = 0x432516;
- **TEST:** A test statement, e.g. EAX and ECX;
- **TEST_CONSTANT:** A test statement including a constant, e.g. EAX and 0x10;
- **UNKNOWN:** Any unknown assembly instruction that cannot be translated.
- **NOTDEFINED:** The default pattern, e.g. all the new statements when created are assigned this default value.

where EAX, EBX, ECX are the general purpose registers, ZF and CF are the zero and carry flags, respectively, and SP is the stack pointer.
3.2. Approach overview

Almost all 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.

A binary program is first disassembled (for this paper, we assume that the binary is unpacked) and translated to a MAIL program. We then build a CFG of the MAIL program and annotate it with patterns described above. This annotated CFG (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 malware or not. This approach is very useful in detecting known malware but may not be able to detect unknown malware.

To detect 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 a signature, we divide the 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 malware, i.e. if a percentage (compared with some predefined threshold computed empirically) of the number of ACFGs involved in a malware signature match with the signature of a program then the program is classified as malware.

3.3. ACFG construction

To translate a binary program, we first disassemble the binary program into an assembly program, and then translate this assembly into an ACFG. We use a sample program, actually a malware sample, to illustrate the steps involved in the translation as shown in Fig. 1. This example shows how an assembly program is translated into an ACFG.

The binary analysis of the function shown in Fig. 1 identifies five blocks in this function labeled 18–25. There are two sets of columns separated by →. The first set of columns lists the hex dump of the function and the second set of columns lists the corresponding translated MAIL statements with annotations.

The example shows the data embedded inside the code section in block 27. This block is used to store, load and process data by the function as pointed out by the underlined addresses in the blocks. There are five instructions that change the control flow of the program as indicated by the arcs in the figure. There are two back edges 20 ← 22 and 19 ← 23, which indicate the presence of loops in the program. The jump in block 24 indicates a jump out of the function.

Each MAIL statement in a block in the ACFG is annotated with a type also called a pattern. There are a total of 21 patterns as explained in Section 3.1 that are used to build the ACFG. For example, an assignment statement with a constant value and an assignment statement without a constant value are two different patterns. Jump statements can have up to three patterns and so on.

3.4. Subgraph matching

Before explaining the subgraph matching technique used in this paper for malware detection, we first define graph isomorphism [27] as follows:

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

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

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

An example of isomorphism is shown in Fig. 2. The edges of graphs \( G \) and \( H_1 \) are not consistent, e.g. edge \( (00, 10) \) in graph \( G \) is not mapped to any edges in graph \( H_1 \), therefore graphs \( G \) and \( H_1 \) are not isomorphic. Whereas the edges of graphs \( G \) and \( H_2 \) are consistent, therefore graphs \( G \) and \( H_2 \) are isomorphic.

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 (NP refers to nondeterministic polynomial time) problem [28]. An ACFG of a program is usually a sparse graph, therefore it is possible to compute the isomorphism of two ACFGs in a reasonable amount of time.

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

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

After the binary analysis performed in Section 3.3 we obtain a set of ACFGs (each corresponding to a separate function) of a program as shown in Fig. 1. To detect if a program contains a malware we compare the ACFGs of the program with the ACFGs of known malware samples from our training database. If a percentage of the ACFGs of the program, greater than a predefined threshold, match one or several of the ACFGs of a malware sample (from the database) then the program will be classified as malware. As part of the machine learning process,
FIGURE 1. Disassembly and translation to ACFG of one of the functions of a malware sample. The arcs are used to represent the control flow of the program and show the transfer of control from one block to another block.
the threshold is computed empirically; by running the detector on the training malware samples with different values of thresholds; and picking a threshold with the best detection rate. An example of malware detection using subgraph matching is shown in Fig. 3.

3.5. 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. 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 annotations/patterns, present in the ACFG, 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) annotations, although there could be differences in the corresponding statement blocks.

An example of Pattern Matching of two isomorphic ACFGs is shown in Fig. 4. One of the ACFGs of a malware sample, shown in Fig. 4a, is isomorphic to a subgraph of one of the ACFGs of a benign program, shown in Fig. 4b. Considering these two ACFGs as a match for malware detection will produce a wrong result, a false positive. The statements in the benign program do not match with the statements in the malware sample. To reduce this false positive we have two options: (i) we can match each statement exactly with each other or (ii) assign patterns to these statements for matching. Option (i) will not be able to detect unknown malware samples and is time consuming, so we use option and (ii) in our approach, which in addition to reducing false positives has the potential of detecting unknown malware samples.

For a successful pattern matching we require all the statements in the matching blocks to have the same patterns. In Fig. 4, only the statements in block 0 satisfy this requirement. The statements in all the other blocks do not satisfy this requirement, therefore these ACFGs fail the pattern matching.

4. RUNTIME OPTIMIZATION

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.

4.1. Parallelization

Figure 5 shows an example of matching ACFGs for malware detection as explained in Section 3.2. The Malware program has 4 ACFGs (functions) represented as \{m_1, m_2, m_3, m_4\} and the benign program has nine ACFGs (functions) represented as \{b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9\}. Each ACFG of the malware program is matched with ACFGs of the benign program. The example has the following successful matches: \(m_1 \rightarrow b_5\), \(m_2 \rightarrow b_4\) and \(m_4 \rightarrow b_9\). The three ACFGs of the malware program successfully match with the three ACFGs out of the total nine ACFGs of the benign program, i.e. over 25% match. Using a threshold of 25% the benign program is detected as malware.

The example discussed above, is matching one benign sample with one malware sample. Similarly, each testing sample is matched with all the training malware samples in the dataset. Based on these matchings, there are two opportunities to parallelize the Subgraph Matching component.

1. Each ACFG matching in a sample is independent of the other matchings, these matchings can be performed in parallel.
2. Each sample can be processed independently of the other samples, the processing of the samples can be performed in parallel.
Multicore processors are becoming 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 [29], a processor with 48 cores on a chip. Keeping in view this ubiquitoussness of multicore in the host machines (also called the end points), the two opportunities listed above and to optimize the runtime, we decided to use Windows threads to parallelize the Subgraph Matching component. The following paragraphs give an overview of our thread implementation approach.

We create two sets of threads. One for each sample under test in the dataset, called the global threads, and the other for each ACFG matching in the sample, called the local threads, i.e. each sample runs in its own thread and this thread creates children threads, one for each ACFG in the sample. A local thread manager (LTM) is implemented to synchronize between the set of threads local to a sample. The LTM counts the number of successful matches and sends a signal to all other local threads to stop if the count crosses the threshold, i.e. the sample is malware and hence there is no need for further ACFG matching. A global thread manager (GTM) is implemented to synchronize between the global threads created for each sample. The GTM and LTM count and keep the total number of threads in the system to MAX_THREAD. The value of MAX_THREAD is computed using Equation (1).

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

where NC = Number of CPUs/Cores.

### 4.2. 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 a 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.

Figure 6 shows an example of ACFG, from the dataset component using different number of threads.

### TABLE 1. Runtime optimization by parallelizing the Subgraph Matching component using different number of threads.

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>2 Cores Runtime reduced by (times)</th>
<th>4 Cores Runtime reduced by (times)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3.20</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5.92</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>7.20</td>
<td>32</td>
</tr>
<tr>
<td>16</td>
<td>5.72</td>
<td>64</td>
</tr>
<tr>
<td>32</td>
<td>5.64</td>
<td>128</td>
</tr>
<tr>
<td>250</td>
<td>5.41</td>
<td>250</td>
</tr>
<tr>
<td>8</td>
<td>14.87</td>
<td>64</td>
</tr>
</tbody>
</table>

Machines used:

Intel Core i5 CPU M 430 (2 Cores) @ 2.27 GHz with 4 GB of RAM and Windows 8 Professional installed

Intel Core 2 Quad CPU Q6700 (4 Cores) @ 2.67 GHz with 4 GB of RAM and Windows 7 Professional installed.

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.
FIGURE 6. ACFG of one of the functions of one of the samples of the MWOR class of malware, before and after shrinking. The ACFG has been reduced from 92 nodes to 47 nodes. (a) Before shrinking and (b) after shrinking.

in the figure, the shape of the graphs before and after shrinking are the same. More of these examples are available at [30].

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 a smaller dataset) and 100 times (for a larger dataset), and still achieved a detection rate of 98.9% with a false positive rate of 4.5% as is latter shown in Table 6.

5. EXPERIMENTS

We conducted an experiment to evaluate the performance of our malware detection technique. The evaluation was carried out using the prototype implementation of our detector named MARD (for Metamorphic malware Analysis and Real-time Detection) [22]. MARD fully automates the malware analysis and detection process, without any manual intervention during a complete run. We present, in this section, the evaluation metrics, experimental settings, obtained results and a comparison of ACFG with other such techniques.

5.1. Performance metrics

Before evaluating the performance of our malware detection technique, 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:

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

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

$$MMP = \frac{1}{n} \sum_{i=1}^{n} Precision_i$$  \hspace{1cm} (2)

$$MMA = \frac{1}{n} \sum_{i=1}^{n} Accuracy_i$$  \hspace{1cm} (3)

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

$$TPR = \frac{TP}{P}$$  \hspace{1cm} (4)

$$FPR = \frac{FP}{N}$$  \hspace{1cm} (5)

### 5.2. Dataset

The dataset used for the experiments consists of total 3350 sample Windows and Linux programs. Out of the 3350 programs, 1020 are metamorphic malware samples collected from three different resources [31–33], and the other 2330 are benign programs. The 1020 malware samples belong to the following three metamorphic family of viruses: Next Generation Virus Generation Kit (NGVCK) [34], Second Generation Virus Generator (G2) [35] and Metamorphic Worm (MWOR) generated by metamorphic generator [31]. NGVCK and MWOR family of viruses are further divided into two and seven Classes, respectively. This Class distribution is shown in Table 2.

The dataset distribution based on the number of ACFGs for each sample, and size of the ACFG after normalization and shrinking is shown in Tables 3 and 4, respectively. The normalizations carried out are removal of NOP (no operation) 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 ranges from 2 to 1272 and the number of ACFGs per benign program ranges from 0 to 1148. Some of the Windows 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 ACFGs. The sizes of these ACFGs are shown in Table 4. The size of the ACFGs of the malware samples range from 1 to 301 nodes, and the size of the ACFGs of the benign programs range from 1 to 521 nodes.

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

### 5.3. 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 4 GB 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 ACFG reduction, both these experiments were carried out with and without ACFG reduction. We were able to complete all the runs (10 times) with and without ACFG reduction. We were able to complete all the runs (five times) without ACFG reduction for the smaller dataset. We completed all the runs (five times) with ACFG reduction with the larger dataset but completed only one run (took over 34 h to complete) without ACFG 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

<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 [31].
TABLE 3. Dataset distribution based on the number of ACFGs for each program sample.

<table>
<thead>
<tr>
<th>Malware samples (1020)</th>
<th>Benign program samples (2330)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of ACFGs</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 4. Dataset distribution based on the size (number of nodes) for each ACFG after normalization and shrinking.

<table>
<thead>
<tr>
<th>Malware samples (1020)</th>
<th>Benign program samples (2330)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of nodes</td>
<td>Number of ACFGs</td>
</tr>
<tr>
<td>1–10</td>
<td>60 284</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>

Total number of nodes before ACFG shrinking = 4 908 422. Total number of nodes after ACFG shrinking = 462 974 (90.6% reduced). Runtime on average reduced by 2.7 times (for smaller dataset) and 100 times (for larger dataset).

sections, we present and describe these two experiments with the results.

5.3.1. 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 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 a 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.

On average it took MARD 15.2037 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 5. Malware detection results for smaller dataset.

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

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 5. The DR improved from 94% when the size of the training set is 25–99.6% when we used a training dataset of 125 samples.

5.3.2. 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 five times and each time a 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 five different runs.

On average it took MARD 946.5824 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.

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 5. The DR improved from 94% when the size of the training set is 25–99.6% when we used a training dataset of 125 samples.

TABLE 6. Malware detection results for larger dataset.

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

On average it took MARD 946.5824 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.

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 5. The DR improved from 94% when the size of the training set is 25–99.6% when we used a training dataset of 125 samples.
TABLE 7. Summary and comparison with ACFG of the metamorphic malware analysis and detection systems discussed in Section 2 and others.

<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>ACFG</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 [11, 12]</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 [36]</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>_CFG-IL [13]</td>
<td>Static</td>
<td>100</td>
<td>12</td>
<td>100/100</td>
<td>×</td>
<td>Win 32</td>
</tr>
<tr>
<td>Chi-squared [37]</td>
<td>Static</td>
<td>~98</td>
<td>~2</td>
<td>40/200</td>
<td>×</td>
<td>Win &amp; Linux 32</td>
</tr>
<tr>
<td>Code-graph [38]</td>
<td>Static</td>
<td>91</td>
<td>0</td>
<td>300/100</td>
<td>×</td>
<td>Win 32</td>
</tr>
<tr>
<td>DTA [39]</td>
<td>Dynamic</td>
<td>100</td>
<td>3</td>
<td>56/42</td>
<td>×</td>
<td>Win XP 64</td>
</tr>
<tr>
<td>Opcode-HMM-Wong [40]</td>
<td>Static</td>
<td>~90</td>
<td>~2</td>
<td>40/200</td>
<td>×</td>
<td>Win &amp; Linux 32</td>
</tr>
<tr>
<td>Opcode-HMM-Austin [41]</td>
<td>Static</td>
<td>93.5</td>
<td>0.5</td>
<td>102/77</td>
<td>×</td>
<td>Win &amp; Linux 32</td>
</tr>
<tr>
<td>Opcode-SD [42]</td>
<td>Static</td>
<td>~98</td>
<td>~0.5</td>
<td>40/800</td>
<td>×</td>
<td>Linux 32</td>
</tr>
<tr>
<td>Opcode-graph [33]</td>
<td>Static</td>
<td>100</td>
<td>1</td>
<td>41/200</td>
<td>×</td>
<td>Win &amp; Linux 32</td>
</tr>
<tr>
<td>Model-checking [14]</td>
<td>Static</td>
<td>100</td>
<td>1</td>
<td>8/200</td>
<td>×</td>
<td>Win 32</td>
</tr>
<tr>
<td>MSA [43]</td>
<td>Static</td>
<td>91</td>
<td>52</td>
<td>150/1209</td>
<td>×</td>
<td>Win 32</td>
</tr>
<tr>
<td>VSA-1 [44]</td>
<td>Static</td>
<td>100</td>
<td>0</td>
<td>25/30</td>
<td>×</td>
<td>Win 32</td>
</tr>
<tr>
<td>VSA-2 [45]</td>
<td>Dynamic</td>
<td>98</td>
<td>2.9</td>
<td>385/826</td>
<td>×</td>
<td>Win XP 64</td>
</tr>
</tbody>
</table>

Real-time here means the detection is fully automatic and finishes in a reasonable amount of time as shown in Tables 5 and 6. Some of the reasons why other techniques do not provide real-time detection are mentioned in Section 2. Moreover, none of the other techniques compute or mention their runtime for malware detection. 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 for the similarity score in the paper.

5.4. Comparison with others

Table 7 gives a comparison of ACFG with the research efforts of detecting the metamorphic malware discussed in Section 2 and others, that do not use CFG and claim to detect metamorphic malware. None of the prototype systems implemented can be used as a real-time detector. Furthermore, only a few systems that claim a perfect detection rate were validated using small datasets.

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

Table 7 reports the best DR results achieved by these detectors. Out of the 15 systems, ACFG clearly shows superior results (the perfect results in Table 7 should be validated with more samples) and, unlike others is fully automatic, supports metamorphic malware detection for 64 bit Windows (PE binaries) and Linux (ELF binaries) platforms and has the potential to be used as a real-time detector.

We have also compared our malware detection with four commercial antimalware programs, including AVG AntiVirus, Microsoft Security Essentials, McAfee and Kaspersky. We used G2 and NGVCK malware families (320 out of the 1020 malware samples, Table 2) for testing these antimalware programs. Out of the 320 malware samples, Microsoft Security Essentials detected 20 (6.25%), McAfee detected 268 (83.75%), Kaspersky detected 268 (83.75%) and AVG detected 274 (85.63%). The combined results of these antimalware programs tagged all the 320 samples as malware, however, none of the individual programs achieved a DR over 86%, whereas ACFG achieved a much higher DR.

6. CONCLUSION

In this paper, we have presented a new technique named ACFG, and shown through experimental evaluation its effectiveness in metamorphic malware analysis and detection and the ability to handle malware with smaller CFGs. We have also optimized the runtime of malware detection through parallelization and ACFG reduction, that makes the comparison (matching with other ACFGs) faster, while 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. Since metamorphic malware is a challenging test case, we believe that our proposed technique can also be effectively
used for the detection of other malware and would like to carry out such experiments in the future. Our future work will also consist of strengthening our existing algorithms by investigating and incorporating more powerful pattern recognition techniques.

REFERENCES


SECTION D: SECURITY IN COMPUTER SYSTEMS AND NETWORKS
THE COMPUTER JOURNAL, 2014


