Malware Variant Detection Using Similarity Search over Sets of Control Flow Graphs

Silvio Cesare and Yang Xiang
School of Information Technology
Deakin University
Burwood, Victoria 3125, Australia
{scesare, yang}@deakin.edu.au

Abstract—Static detection of polymorphic malware variants plays an important role to improve system security. Control flow has shown to be an effective characteristic that represents polymorphic malware instances. In our research, we propose a similarity search of malware using novel distance metrics of malware signatures. We describe a malware signature by the set of control flow graphs the malware contains. We propose two approaches and use the first to perform pre-filtering. Firstly, we use a distance metric based on the distance between feature vectors. The feature vector is a decomposition of the set of graphs into either fixed size k-subgraphs, or q-gram strings of the high-level source after decompilation. We also propose a more effective but less computationally efficient distance metric based on the minimum matching distance. The minimum matching distance uses the string edit distances between programs’ decompiled flow graphs, and the linear sum assignment problem to construct a minimum sum weight matching between two sets of graphs. We implement the distance metrics in a complete malware variant detection system. The evaluation shows that our approach is highly effective in terms of a limited false positive rate and our system detects more malware variants when compared to the detection rates of other algorithms.

Keywords—computer security; malware classification; static analysis; control flow; structuring; decompilation

I. INTRODUCTION

A. Motivation

Malicious software presents a significant challenge to modern desktop computing. According to the Symantec Internet Threat Report [1], 499,811 new malware samples were received in the second half of 2007. F-Secure additionally reported, “As much malware [was] produced in 2007 as in the previous 20 years altogether” [2]. Detection of malware before it adversely affects computer systems is highly desirable. Static detection of malware is still the dominant technique to secure computer networks and systems against untrusted executable content.

Detecting malware variants improves signature based detection methods. The size of signature databases is growing exponentially, and detecting entire families of related malicious software can prevent the blowout in the number of stored malware signatures.

B. Background

Malware classification and detection can be divided into the tasks of detecting novel instances of malware, and detecting copies or variants of known malware. Both tasks require suitable feature extraction, but the class of features to be extracted is often dependant on which problem is trying to be solved. Detecting novel samples primarily uses statistical machine learning. On the contrary, malware variant detection uses the concept of similarity searching to query a database of known instances. These similarity queries or nearest neighbour searches are known in machine learning as instance-based learning. Instance-based learning uses distance functions to show dissimilarity and hence similarity between objects. If the distance function has the mathematical properties of a metric, then algorithms exist that enable more efficient searching than an exhaustive set of queries over the database.

Traditional and commercial malware detection systems have predominantly utilised static string signatures [3-4] to query a database of known malware instances. Static string signatures capture sections of the malware’s raw file content that uniquely identifies them. String signatures have been employed because they have desirable performance characteristics that enable real-time use [5]. However, string signatures perform poorly when faced with polymorphic malware variants. Exact string matching also ineflectively handles closely related but non-identical signatures.

Polymorphic malware variants have the property that the byte level content of the malware changes between instances. This can be the result of source code modifications or self mutation and obfuscation to the malware. Signatures that rely on fixed byte level content are unable to capture the invariant characteristics between these polymorphic instances.

Control flow has also been used to overcome the limitations of byte level and instruction level classification [6]. Control flow has the desirable property that instruction level changes do not affect the resulting flowgraphs. Control flow is observed to be more invariant in polymorphic malware [7].

Control flow represents the execution path a program may take. Control flow information appears in two main forms. The call graph represents the inter-procedural control flow. The intra-procedural control flow is represented as a
set of control flow graphs with one graph per procedure. Most malware analysis using control flow has focused on analysing the call graph.

The challenge of using graphs to show similarity is that accurately measuring similarity such as when using the graph edit distance does not perform in polynomial time. Therefore, research must investigate methods that make using graphs feasible for large scale malware detection. The real or near real-time constraints of Antivirus software make this challenge even more significant. The challenge increases again when complex graph based objects are considered such as the set of graphs signature our research investigates.

C. Innovation

Our work is based on control flow classification but we make the following contributions:

- We propose a system that performs similarity searching of sets of control flow graphs. We perform the search in close to real-time in the expected case. No other system has demonstrated near real-time performance for this use of control flow based signature.
- We propose using fixed size k-subgraphs to construct a feature vector approximating a set of graphs. Using a vector representation improves efficiency significantly and has not been used before.
- We also propose the novel use of a polynomial time algorithm to generate q-gram features of decompiled control flow graphs to construct a feature vector. These features are shown to have more accuracy than k-subgraphs and can be constructed faster than k-subgraphs. K-subgraph feature construction is not known to take polynomial time. Q-grams of decompiled graphs have not been used before for malware classification.
- We propose a distance metric between two sets of graphs based on the minimum matching distance. The minimum matching distance uses the linear sum assignment problem. It has been used previously with sets of vectors, but not sets of graphs. The minimum matching distance has not been used before in malware classification.
- We implement these ideas in a complete prototype system and perform an evaluation on a set of benign binaries and on real malware, including those malware that are packed and polymorphic. The evaluation demonstrates the system is effective and fast enough for potential desktop adoption.

D. Structure of the Paper

The structure of this paper is as follows: Section II describes the related work in static malware classification. Section III defines the malware classification problem and our approach. Section IV describes the unpacking and general static analysis component of the system. Section V describes the pre-filtering stage used in classification. This is a course grained classification process. Section VI describes the fine grained classification algorithms. Section VII performs an evaluation using benign and malicious samples. Section VIII summarizes and concludes the paper.

II. RELATED WORK

The related work reviewed in this paper is limited to static classification approaches. Dynamic approaches examining run-time behaviour are not closely related to this paper and are therefore not examined.

Automatic extraction of string signatures has been proposed and employed by commercial Antivirus software [3-4]. In these systems, all possible signatures of fixed size are extracted and then culled to eliminate strings that appear in benign samples. However, polymorphic malware make string signatures prone to failure when the byte level content changes due to mutation, recompilation, and source code modification of the malware.

Code normalization has been investigated to canonize malware before string scanning [8-9] and improve detection rates of malware variants. In [8], static analysis eliminated superfluous control flow. Additionally, instruction sequences that had no effect were also removed using a decision procedure to perform the analysis. Malware normalization improves detection but does not always effectively canonize a program to a unique form which affects the effectiveness and efficiency of the system.

Fingerprinting malware based on opcode distributions has shown to be partly effective [10]. An improved approach was to examine byte sequences and opcode sequences in an n-gram analysis. N-grams and n-perms were proposed to identify similarity between malicious programs to build evolutionary trees [11]. By extracting relevant n-grams as particular features, feature vectors could be compared in order to construct similarity. Machine learning and classification extended these systems to detect unknown malware in [12-13]. These systems improve the effectiveness of static string signatures, but instruction level classification has similar problems when the instruction stream changes to any significant degree.

Malware classification using the basic blocks of a program has been investigated in [14]. A basic block represents a sequence of instructions without an intervening control transfer instruction. The edit distance between basic blocks identifies similarity. Existence of a basic block in a malicious sample can be determined using an inverted index or bloom filters. Previous research demonstrated that this is effective at detecting some malware variants, but is not effective when byte and instruction level changes occur. In the systems that implemented basic block classification, efficiency was not shown to be sufficient for desktop adoption.

The ordering of system API calls of a program can be extracted and used for malware classification. [15] proposed using association mining of API calls to detect unknown malicious programs. The problem of this approach is that code packing can obscure the API calls [16].

An approach to malware classification that is more resistant to byte and instruction level changes is by using control flow as a feature. Combining data flow analysis and
control flow analysis was proposed in [17-18]. Annotations were made to the control flow graphs to incorporate abstractions of the instructions and data flow. These annotated flowgraphs were compared to signatures, or automata, that describe the malware. If the malware signature is present in the query program, a malware instance can be detected. In [19], value set analysis was used as a specific data flow analysis to construct signatures. Our control flow graph based system is more effective at detecting modified variants because it allows for errors in the signature generation.

Interprocedural control flow using the call graphs of a program have been compared to show similarity to existing malware [6, 20-22]. Procedures or nodes in the call graph are fingerprinted and common nodes between two program call graphs are identified. By identifying matching nodes, an approximation is made to the graph edit distance. The edit distance allows similarity to be measured. An approach to transform the interprocedural control flow information into a context free grammar, allowing for homomorphism testing using string equality was also proposed in [23], but this technique does not allow for approximate matches.

An alternative approach to using the call graph is using the control flow graphs. Ignoring the edges in the call graph and identifying exact or isomorphic control flow graphs was proposed for a real-time system and shown to be effective [24]. Intraprocedural control flow using the control flow graphs have been compared by decompiling the graph to source code and using the string edit distance [25], but was not as efficient as exact matching. Identifying isomorphisms and the maximum common subgraph of a whole program control flow graph using tree automata was proposed in [26].

A worm detection system fingerprinted control flow using subgraphs of a small and fixed size as a feature in [7].

Our research employs a novel approach to finding the distance between malware based on individual distances or similarity between control flow graphs. We allow for efficiently approximate matching of sets of graphs whereas previous research only provided efficient exact matching. Our research additionally extends the concept of use subgraphs as a feature by incorporating it into larger classification system based representing a set of graphs as feature vectors and consequently using similarity searching. We also demonstrate a more effective feature using the q-grams of decompiled control flow graphs.

III. PROBLEM STATEMENT AND OUR APPROACH

A. Problem Statement

New programs that are discovered on the host system are inspected to determine if they are malicious or benign. Unknown malware are detected by calculating their similarity to existing malware. A high similarity identifies a malicious variant. Existing malware are collected from honeypots and other malicious sources to construct a database of malware signatures. The described malware variant detection problem is equivalent to the software similarity search problem.

The software similarity problem is to determine if program p is a copy or derivative of program q. It is based on the definition proposed in [27]. A workflow is shown in Fig. 1. The software similarity problem is extended to operate over a database of programs. We use the nearest neighbour search.

B. Our Approach

Our approach builds a signature or birthmark of a malware based on the set of control flow graphs it has. We compare signatures using distance metrics to show similarity. We take advantage of the efficiency of using feature vectors to represent a program’s control flow information. Because a feature vector is a course grained view of the program’s control flow and introduces false detections, we incorporate an additional and more accurate representation using the minimum matching problem to compare two sets of graphs.

Malware is first unpacked to remove obfuscations. Control flow is reconstructed and the control flow graphs decompiled and structured into strings. Malware variants are detected by identifying existing malware the query programs are related to. Pre-filtering is used to provide a list of potentially related malware. The pre-filtering algorithm is based on constructing a feature vector to represent the query programs and malware. Either of two algorithms can be used to extract features. Firstly, subgraphs of size k are used to represent features. Alternatively, q-grams are extracted from the strings representing the structured graphs. Using either algorithm for feature extraction, the most relevant features are used to construct a feature vector. The pairwise similarity between two feature vectors employs a distance function on the pair of vectors. Vectors that are close to each other are indexed to the same bucket. To identify candidates with high similarity to existing malware, a metric similarity search is performed using Vantage Point trees [28].

To compare the query to the candidate malware, a more accurate pairwise distance function is used. Each control flow graph from one program is assigned a unique mapping to a flowgraph from the other program. This mapping intuitively shows the flowgraphs represent the same procedure. The mapping is assigned a weight and the mappings chosen by considering it as an optimization problem. The mappings are chosen to minimize the sum of all weights associated with the mappings. The weight is the distance between flowgraphs and is based on the string distance between structured graphs. This sum weight is
known as the minimum matching distance and is known to be metric. Metric Access Methods using DBM trees [29] are used to perform a similarity search.

IV. UNPACKING AND STATIC ANALYSIS

A. Unpacking

The query program may have its real contents hidden using the code packing transformation [30]. Code packing encrypts, compresses, or obfuscates the code by dynamically generating the original program at runtime. This obfuscation layer is removed using automated unpacking. The unpacking process employs application level emulation as proposed in previous research [25].

B. Disassembly and Control Flow Reconstruction

In our system, an unpacked program is disassembled using speculative disassembly [31]. The disassembly is translated to an intermediate language using the Wire static analysis framework. The control flow is reconstructed into control flow graphs for each procedure [25] based on the intermediate code. This and the remaining components of the static analysis are architecture independent.

The control flow graphs are normalized to eliminate unnecessary jumps such as when a unconditional branch is used to divide a basic block in two. This is typically done by a malware for the purpose of changing its byte level content and static string signature. The result after control flow reconstruction is the set of control flow graphs associated with each identified program procedure.

C. Structuring

Structuring is a reverse engineering and decompilation technique to transform a control flow graph into its high level source code representation. We use a structuring algorithm to transform the control flow graphs into strings. The intuition is that similar control flow graphs are structured into similar strings [25]. The structuring algorithm we use is based on the algorithm used in the DCC decompiler [32]. The grammar for the resulting string is shown in Fig. 2. Fig. 3 illustrates the relationship between a control flow graph, a structured graph, and the resulting string representation.

V. PRE-FILTERING

To reduce the search space for potentially related malware, we use an initial similarity search to select candidate malware variants. We construct and search for feature vectors that are associated with malware. We propose two methods to extract features for the feature vector using either k-subgraphs or q-grams of structured control flow. Q-grams are more efficient and evaluation shows that they generate more accurate results. The use of approximating a set of graphs by a vector is a novel contribution of this paper.

A. The K-Subgraph Feature

Using subgraphs of size k to characterize control flow has been investigated in previous literature [7]. Subgraphs of size k are those subgraphs in the control flow graph which have k nodes. We use each possible subgraph of size k in the control flow graphs as features of the program. Our novel contribution is the use of these features in the construction of a feature vector which is subsequently used in a similarity search.

For each control flow graph, we construct a depth first spanning tree to eliminate cycles. We then perform a traversal of all possible paths in the tree where the traversal is terminated when k nodes have been visited [7].

Given a subgraph of size k, the graph is transformed into a unique and canonical representation using the Bliss open-source toolkit [33]. A canonical graph labeling is formed and the adjacency matrix of the resulting graph is stored as a string. This string represents a feature of the malware. Graph canonization is not known to take polynomial time in the general case.

B. The Control Flow Q-Gram Feature

Q-grams can be employed to represent control flow if the control flow graph is modeled as a string. We use structuring to generate the strings. A q-gram is any character sequence
in the string of length q, constructed as a sliding window. For q-grams to be an effective feature, the strings must satisfy the property that similar control flowgraphs have similar strings. Each possible q-gram in the string represents a feature. Constructing the strings and the q-grams can be done in polynomial time and is more efficient than using k-subgraphs. The use of q-grams on the structured control flow graphs is a novel contribution of this paper.

C. Feature Selection

The number of possible and distinct features in a program is large. To reduce the number of distinct features to a feasible number, the set of the 500 most frequent features are selected from a training set of malicious and benign programs. Both program classes are used because it is our intuition that there is no significant classification difference in control flow between malicious and benign programs. This intuition forms the basis for our instance-based learning approach to classification. These features represent dimensions in a program’s feature vector, and the frequency of a particular feature represents the dimension’s magnitude. For the remaining features not in the 500 most frequent, they are ignored when constructing the feature vector of a program.

D. Feature Vector Distance

To calculate the pairwise similarity between two feature vectors, a distance metric is employed. Many distance metrics are possible including the Euclidean distance. We use the Manhattan distance because of its efficiency when compared to the more traditional Euclidean distance. The Manhattan distance is also known as the city block or \( L_1 \) distance. For \( n \)-dimensional vectors \( p \) and \( q \), the Manhattan distance is:

\[
d_1(p,q) = \|p - q\|_1 = \sum_{i=1}^{n} |p_i - q_i|
\]

E. Indexing and Searching the Feature Vectors

We group the feature vectors into buckets. To group the feature vectors, the neighbours of each feature vector that are equal to or exceed the similarity of 0.6 are placed in the same bucket. This threshold was chosen empirically through experimentation.

We pre-filter malware variants by performing a range nearest neighbour similarity search to our query feature vector. Given database \( D \), query \( q \), and threshold \( t \), the set of nearest neighbours \( R \) is:

\[
R = \{ r \in D \} : 1 - \frac{d(r,q)}{|q|} \geq t
\]

The results of the similarity search are candidate matches that can be used in the subsequent stage of comparing programs using the assignment problem. In our prototype, we use a Vantage Point Tree [28] for indexing the feature vectors used in the pre-filtering stage using the GDBI Arboretum library [34].

VI. Malware Classification

We propose a more accurate distance function to be applied to candidate malware variants after their identification in the pre-filtering stage. This improved distance is based on the distance between the control flow graphs’ structured strings and is a variant of the minimum matching distance.

A. String Distance Metrics

The Levenshtein distance provides the minimum number of edit operations, or distance, to transform one string to the other. Sequence alignment algorithms also provide suitable string distances. The Smith-Waterman algorithm is an optimal local alignment algorithm that we use to score the distance between structured strings in our prototype.

B. A Distance Function for Programs Based On the Linear Sum Assignment Problem

The linear sum assignment problem is to bijectively match distinct pairings of elements between two sets. Each match or assignment has an associated weight. The assignments are made such that, the sum of the weights are minimized. The linear sum assignment problem is also known as a minimum weight perfect matching.

For each program examined by our malware classification system, there exists an associated set of control flow graphs. Each set is represented as a set of structured strings. The assignment problem is used to match control flow graphs between sets. The intuition is that these matched control flow graphs are shared characteristics between malware variants. The weight of the assignment is the string metric or distance between those strings. We construct a matrix containing the weights of all possible pairings between two programs’ sets of structured strings. If the number of elements in each set is not identical, then the elements that cannot be paired to existing elements are paired to the null element. The weight of this pairing is equivalent to the size of the element’s string.

We define the distance between programs as the minimal cost function generated by a solution to the assignment problem using the matrix of weights. The use of the assignment problem is a novel contribution used in our system to show the distance between programs. This cost is a variation of the minimum matching distance [35] which is known to be metric.

Formally, let two programs \( P_1 \) and \( P_2 \) be defined as sets of control flow graphs and let \( S \) be a structuring function.

We first normalize the size of the sets making them equal. The additional elements, \( b \), used in the normalization process are placeholders and not used for any other purpose.

\[
M_1 = S(P_1)
M_2 = S(P_2)
M'_1 = \{ a_i \in M_1 \} \cup \{ b \} : 1 \leq i \leq |M_1| < j \leq |M_2|
M'_2 = \{ a_i \in M_2 \} \cup \{ b \} : 1 \leq i \leq |M_2| < j \leq |M_1|
\]

The function \( ed(a,b) \) is defined as the distance between strings. The distance, \( d \), between the programs is found as follows:

\[
C : M'_1 \times M'_2 \rightarrow \mathbb{R}
\]
Find a bijection such that the distance, $d$ is minimized.

C. Solutions to the Assignment Problem

The assignment problem can be solved optimally using the Munkres or Hungarian algorithm [36] in time $O(N^3)$. Although an optimal solution is available, for programs that have a large number of control flow graphs, the time complexity required of $O(N^3)$ is impractical. In these cases when the number of nodes is greater than 300, we use a heuristic solution based on a greedy assignment. The greedy assignment matches an element from one set by selecting the element from the other set with the lowest associated weight. The time complexity is $O(N(N+1)/2)$. The greedy solution performs more efficiently, but the program distance it identifies is often significantly higher than the optimal solution.

D. Similarity Search of Malware

The similarity between two objects is given by:

$$s(p, q) = 1 - \frac{d(p, q)}{|q|} \quad \text{iff} \quad d(p, q) < |q|$$

A threshold for similarity, $t$, is chosen as 0.6. The threshold was chosen manually after an empirical evaluation. We then use this to identify any nearest neighbour $p$ to the query $q$ in the set of malware, $E$, returned by our pre-filtering process.

$$\exists p: p \in E, 1 - \frac{d(p, q)}{|q|} \geq t, d(p, q) < |q|$$

This is the system’s final result. We implement this any nearest neighbour search using a DBM-Tree [29] in a modified GBDI metric access method library [34].

VII. Implementation and Evaluation

A. Implementation

Our implementation is built as a set of modules in the Malwise malware and static analysis framework. Malwise consists of approximately 100,000 LOC of C++ and its features include unpacking using application level emulation and static analysis. The modules we developed to perform malware classification consist of approximately 3,000 LOC of C++. The workflow of the Malwise malware classification system is shown in Fig. 4. Emulation is used to perform unpacking. However, the classification process uses only static analysis and that is the focus of our current work.

**Effectiveness**

To perform an evaluation of the classification system 17,430 real malware with unique MD5 hashes were collected between 02-01-2009 and 8-12-2009 from honeypots in the mwcollect Alliance [37] network. From these malware, 15,398 were found to be valid object files for Windows Vista – the remaining binaries were invalid, specific to Windows XP, and not able to be processed by our prototype’s unpacking system. In addition to the malware, we employed the use of 1601 benign binaries, which were obtained from the Windows system directory and the Cygwin [38] executable directories. The system we used to evaluate the prototype classification system was an Intel Q6600 Quad...
The classification algorithm chosen was the Ronor family of malware. The similarity between malware that belongs to the same family is generally quite low, and any similarity found should be non-matching samples. We expect that similarity found will identify how effective the classification algorithm chosen empirically through experimental testing.

The prototype system requires training to select the 500 most common q-grams and k-subgraphs, 1769 malware and 1601 benign binaries were used in the training set to generate features.

1) Evaluation of False Positives in Pre-filtering

To evaluate the accuracy of the q-gram and k-subgraph classification algorithms we first constructed a database of 10,000 malware signatures. Then, we found the similarities between each of 10,000 malware and 280 benign binaries from the windows system directory. This evaluation is to identify how effective the pre-filtering stage is at filtering non-matching samples. We expect that similarity found should be generally quite low, and any similarity found above or equal to 0.6 identifies a false positive. The size of the q-gram was 4. The size of the k-subgraph was 10 as recommended in the existing literature. Better selections of the size k were not investigated. The threshold of 0.6 was chosen empirically through experimental testing.

The evaluation shown in Table I demonstrates that false positives, or collisions, occur using this pre-filtering algorithm with either feature. The q-gram feature is shown to generate considerably less collisions and false positives compared to using k-subgraphs of size 10. For this reason, we excluded using k-subgraphs as part of the classification process in further evaluations.

2) True Positives of the System Compared to Previous Research

The next evaluation we performed was to examine the similarity matrices for our complete classification algorithms on a known family of related malware.

This evaluation incorporates all elements of our system and is the main evaluation we performed on the true positive detection rate of the system. The system ideally identifies high similarity between malware that belongs to the same family. The malware chosen was the Ronor family of malware.

### Table I. False positives using k-subgraphs and q-grams.

<table>
<thead>
<tr>
<th>Similarity</th>
<th>K-Subgraphs</th>
<th>Q-Grams</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1302161</td>
<td>2334251</td>
</tr>
<tr>
<td>0.1</td>
<td>463170</td>
<td>413667</td>
</tr>
<tr>
<td>0.2</td>
<td>356345</td>
<td>40055</td>
</tr>
<tr>
<td>0.3</td>
<td>285202</td>
<td>7899</td>
</tr>
<tr>
<td>0.4</td>
<td>200326</td>
<td>3790</td>
</tr>
<tr>
<td>0.5</td>
<td>129790</td>
<td>327</td>
</tr>
<tr>
<td>0.6</td>
<td>46320</td>
<td>11</td>
</tr>
<tr>
<td>0.7</td>
<td>10784</td>
<td>0</td>
</tr>
<tr>
<td>0.8</td>
<td>5883</td>
<td>0</td>
</tr>
<tr>
<td>0.9</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>1.0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Table II. Similarity matrices for Ronor malware.

<table>
<thead>
<tr>
<th>ao</th>
<th>b</th>
<th>d</th>
<th>e</th>
<th>g</th>
<th>k</th>
<th>m</th>
<th>q</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>ao</td>
<td>0.44</td>
<td>0.28</td>
<td>0.27</td>
<td>0.28</td>
<td>0.55</td>
<td>0.44</td>
<td>0.44</td>
<td>0.47</td>
</tr>
<tr>
<td>b</td>
<td>0.44</td>
<td>0.27</td>
<td>0.27</td>
<td>0.27</td>
<td>0.51</td>
<td>1.00</td>
<td>1.00</td>
<td>0.58</td>
</tr>
<tr>
<td>d</td>
<td>0.28</td>
<td>0.27</td>
<td>0.27</td>
<td>0.27</td>
<td>0.56</td>
<td>0.27</td>
<td>0.27</td>
<td>0.27</td>
</tr>
<tr>
<td>e</td>
<td>0.27</td>
<td>0.27</td>
<td>0.27</td>
<td>0.27</td>
<td>0.59</td>
<td>0.27</td>
<td>0.27</td>
<td>0.27</td>
</tr>
<tr>
<td>g</td>
<td>0.28</td>
<td>0.27</td>
<td>0.56</td>
<td>0.59</td>
<td>0.27</td>
<td>0.27</td>
<td>0.27</td>
<td>0.27</td>
</tr>
<tr>
<td>k</td>
<td>0.55</td>
<td>0.51</td>
<td>0.27</td>
<td>0.27</td>
<td>0.27</td>
<td>0.51</td>
<td>0.51</td>
<td>0.75</td>
</tr>
<tr>
<td>m</td>
<td>0.44</td>
<td>1.00</td>
<td>0.27</td>
<td>0.27</td>
<td>0.27</td>
<td>0.51</td>
<td>1.00</td>
<td>0.58</td>
</tr>
<tr>
<td>q</td>
<td>0.44</td>
<td>1.00</td>
<td>0.27</td>
<td>0.27</td>
<td>0.27</td>
<td>0.51</td>
<td>1.00</td>
<td>0.58</td>
</tr>
<tr>
<td>a</td>
<td>0.47</td>
<td>0.58</td>
<td>0.27</td>
<td>0.27</td>
<td>0.27</td>
<td>0.75</td>
<td>0.58</td>
<td>0.58</td>
</tr>
</tbody>
</table>

### Table III. Malware detection rates.

<table>
<thead>
<tr>
<th>Classification Algorithm</th>
<th>Klez</th>
<th>Netsky</th>
<th>Ronor</th>
<th>Frethem</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Maximum</strong></td>
<td>36</td>
<td>49</td>
<td>81</td>
<td>289</td>
</tr>
<tr>
<td>Exact</td>
<td>20</td>
<td>29</td>
<td>17</td>
<td>139</td>
</tr>
<tr>
<td>Heuristic Approximate</td>
<td>20</td>
<td>27</td>
<td>43</td>
<td>144</td>
</tr>
<tr>
<td>Q-Grams</td>
<td>20</td>
<td>31</td>
<td>79</td>
<td>226</td>
</tr>
<tr>
<td>Optimal Distance</td>
<td>22</td>
<td>46</td>
<td>73</td>
<td>220</td>
</tr>
<tr>
<td>Q-Grams + Optimal Distance</td>
<td>20</td>
<td>43</td>
<td>73</td>
<td>217</td>
</tr>
</tbody>
</table>

### Table IV. False positives.

<table>
<thead>
<tr>
<th>Classification Algorithm</th>
<th>False Positives</th>
<th>FP Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q-Grams</td>
<td>10</td>
<td>0.62</td>
</tr>
<tr>
<td>Q-Grams + Optimal Distance</td>
<td>7</td>
<td>0.43</td>
</tr>
</tbody>
</table>

Core 2.4GHz PC with 4G of memory running 32-Bit Windows Vista Home Premium, Service Pack 1.

The prototype system requires training to select the 500 most common q-grams and k-subgraphs, 1769 malware and 1601 benign binaries were used in the training set to generate features.
malware to replicate previous research [6, 24-25]. The family of malware variants was identified by an Antivirus vendor and may not necessarily have been entirely trustworthy. We obtained the malware from the Offensive Computing malware database [39]. We compared the q-gram classification algorithm and the assignment problem classification algorithm. Additionally, we made comparison to algorithms proposed in previous research. We compared our system to a real-time flowgraph based classification system that uses exact or isomorphic testing of control flow graphs in [24]. We expect our approximate matching algorithm to detect more variants than the exact matching system. The second comparison was to a previously proposed system that uses an approximate control flow graph matching algorithm in [25]. The previously proposed system uses an alternative heuristic algorithm based on greedy matching and string metrics of the structured control flow graphs. The system we compared against does not employ the assignment problem or a program distance metric.

The results are shown in Table II. Identified malware variants have similarities exceeding or equal to 0.6. Identified variants additionally have their table cells highlighted. The results show that our prototype detects more malware variants in this family of malware than existing systems.

The Netsky, Roron, and Klez, and Frethem malware were chosen to continue the evaluation of variant detection. For each malware family, the maximum number of possible variants is listed, along with the detection rates of our algorithm and existing algorithms [24-25], Table III lists the results. Ideally, the number of variants detected would be the maximum, meaning all variants were related to each other. Our system detects many but not all variant relationships. It may be that some of the variants are quite distinct. The classification algorithms we proposed in this research are shown to be highly effective and detect more malware than previous systems. Looking at the Frethem malware family, our algorithm detects 217 variant relationships, while the next best system detects 144 variants.

3) Evaluation of the System’s False Positives

We next evaluate the number of false positives generated by the system. The number of false positives gives indication of how the distance functions perform using non similar programs. We perform the evaluation using a malware database size of 10,000. We classify the set of 1601 benign programs and expect that any identified malware is a false positive. The evaluation demonstrates false positives when using the q-gram as is also demonstrated in Table IV. For a database size of 10,000 the false positive rate is shown to be less than 1%. We also show that using the assignment problem solution in conjunction with the q-gram classification results in fewer false positives.

We suspect the reason for the remaining false positives is because we do not eliminate statically linked functions from the analysis. Programs that share the same statically linked objects have a high similarity, even when the programs are generally unrelated.

C. Efficiency

To evaluate the efficiency of our system, we record the execution time to classify each of 1601 benign programs and 15,398 malware. The malware database is pre-populated with 10,000 malware signatures. We evaluate the complete processing time of the system including unpacking, disassembly, control flow reconstruction and analysis. The processing times for the malware and benign programs are shown in Table V. The median time for processing malicious samples is 0.84 seconds. The maximum time taken is 585 seconds and may have resulted from excessive memory consumption causing thrashing. Some candidate buckets were large due to a high number of related malware variants, resulting in higher than average pairwise comparisons using the less efficient distance function. Unpacking binaries using emulation may also cause significant overhead in some cases. If applied in a desktop environment, the analysis may need to flag such binaries that impact performance and whitelist known benign programs that would otherwise cause false positives. In practice, we do not see these edge cases as reducing the effectiveness if they are handled in these ways. The median time for processing the benign binaries is 0.06s. Classifying only the Windows system programs has a median time of 0.15s. Processing benign programs is the expected case and performs more quickly than classifying malware due to the extra overhead of unpacking. The slowest time is 8.06 seconds which is still reasonable for industrial deployment. Our system improves the performance in classifying benign programs compared to the less effective exact matching algorithm proposed in [24] which has a median Windows system directory processing time of 0.25s. This is not due to classification performance, which is almost identical, but due to improvements to efficiency in the static analysis component. The general results indicate that the speed of classification may warrant the system suitable for real-time use for desktop Antivirus or on an Email gateway system.

<table>
<thead>
<tr>
<th>% Samples</th>
<th>Benign Time(s)</th>
<th>Malware Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.02</td>
<td>0.16</td>
</tr>
<tr>
<td>20</td>
<td>0.02</td>
<td>0.28</td>
</tr>
<tr>
<td>30</td>
<td>0.03</td>
<td>0.30</td>
</tr>
<tr>
<td>40</td>
<td>0.03</td>
<td>0.36</td>
</tr>
<tr>
<td>50</td>
<td>0.06</td>
<td>0.84</td>
</tr>
<tr>
<td>60</td>
<td>0.09</td>
<td>0.94</td>
</tr>
<tr>
<td>70</td>
<td>0.13</td>
<td>0.97</td>
</tr>
<tr>
<td>80</td>
<td>0.25</td>
<td>1.03</td>
</tr>
<tr>
<td>90</td>
<td>0.56</td>
<td>1.31</td>
</tr>
<tr>
<td>100</td>
<td>8.06</td>
<td>585.16</td>
</tr>
</tbody>
</table>

V. BENIGN AND MALICIOUS BINARY PROCESSING TIME.
VIII. CONCLUSION

Malware can effectively be characterized by its control flow. We proposed a malware classification system using approximate matching of control flow graphs. We used techniques to extract q-grams and k-subgraphs of sets of control flow graphs and created feature vectors. From these feature vectors we were able to construct an efficient distance metric and similarity search. We also used the assignment problem and the string distance to construct a distance metric between programs. We implemented these algorithms in a prototype and performed an evaluation of the system. Our evaluation showed that our work more effectively detected malware than previous comparable systems. The number of false positives was low, and the efficiency of the prototype demonstrated that the system could be used on a desktop system or Email gateway.

REFERENCES