Control Flow-based Malware Variant Detection

Silvio Cesare, Student Member, IEEE, Yang Xiang, Senior Member, IEEE, and Wanlei Zhou, Senior Member, IEEE

Abstract—Static detection of malware variants plays an important role in system security and control flow has been shown as an effective characteristic that represents polymorphic malware. In our research, we propose a similarity search of malware to detect these variants using novel distance metrics. We describe a malware signature by the set of control flow graphs the malware contains. We first experiment with string based signatures. We then try using vector and set of strings based signatures. 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 use this distance metric to perform pre-filtering. 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.

Index Terms—computer security, malware classification, static analysis, control flow, structuring, decompilation.

1 INTRODUCTION

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. In 2010, over 1.5 billion malicious code detections were identified [2] by the same vendor. F-Secure published, “As much malware [was] produced in 2007 as in the previous 20 years altogether” [3]. This trend is continuing and makes the detection of malware before it adversely affects computer systems highly desirable. To achieve this, 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.

1.1 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 [4, 5] 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 [6]. However, string signatures perform poorly when faced with polymorphic malware variants. Exact string matching also ineffectively 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.

To extend the capabilities of string based signatures, code normalization has been investigated [7, 8]. Using code normalization, programs are transformed into an ideally canonical representation to eliminate the mutations and obfuscations incurred by polymorphic variants. The normalized program is then analysed using existing string based malware detection systems. Code normalization improves the effectiveness of string based scanning,
but has not been widely adopted due to efficiency concerns.

Efficient real-time systems have been proposed that examine the run-time behaviour of programs to identify malicious behaviour [9]. Malicious behaviour can either conform to a policy of malicious intent, or reassemble the behaviour of a program instance, known in advance to be malicious. However, static detection of malware has advantages - it does not require conditional, untrusted or sandboxed execution of malware once the original contents of the malware are visible. Unpacking is the processing of revealing that code and typically occurs before the malware performs its malicious intent. Many Antivirus products implement static unpacking for known packers, and this accounts for the majority of samples. However, for novel packing techniques unpacking is often a dynamic process making effective static analysis against novel malware a hybrid approach. Additionally, snapshots of process images can be taken at runtime, thus avoiding the most common packing issues and can be used to statically identify if those processes belong to a known malware family.

A variety of algorithms have been employed to statically detect malware variants with superior classification compared to string based approaches. An n-gram is one of all possible fixed sized substring extracted from a larger string. Our work is directly related to the n-gram concept. N-grams of byte level, or instruction level content, utilising machine learning and classification has been proposed. However, n-grams are ineffective with polymorphic malware because of the changes the instruction level content.

More detailed program analysis techniques have been employed on the instruction level content to extract high level features. Data flow analysis reveals useful high level features that are more invariant than instruction content alone. Likewise, abstract interpretation using specific domains reveals desirable features. Efficiency still remains a concern for industrial usage.

Control flow has also been used to overcome the limitations of byte level and instruction level classification [10]. 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 [11].

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. Our work is based on the set of control flow graphs of the program. In some literature, the individual control flow graphs are merged together into a single interprocedural control flow graph (ICFG). However, for our work, we represent each procedure with a separate graph and therefore consider the set of graphs problem. In contrast, most malware analysis using control flow has focused on analysing a single call graph. The advantage of considering each control flow graph individually is that we can apply the decompilation technique of structuring which is not possible with the ICFG.

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.

1.2 Motivation
This work is motivated by several real-world applications that would benefit from control flow-based malware variant detection.

1.2.1 A replacement to traditional Antivirus
Traditional AV suffers from the inability to detect malware variants efficiently from large databases. Control flow is effective and our system makes such a system practically efficient when using large databases. Moreover, it would reduce the size of the database required on the end host due to requiring fewer samples to recognise a large malware family. Malware classification can help group samples together to create traditional AV signatures, and our work improves the performance of this, yet our work also presents a direction for a replacement of user-level AV to cope with today’s malware problem.

1.2.2 To cluster interesting samples
AV vendors need to know which malware families are significant enough that they require manual analysis. Our system could be used to identify variants and group them to their family. If many instances of a family are identified, then that family may require human analysis to determine what the real impact of the malware is. Moreover, interesting samples, such as state-sponsored malware, can be used as a query to retroactively find any other related malware from databases of unlabelled samples.

1.2.3 Incident Response
An accurate system that identifies what family of malware a sample belongs to could be used in incident response. If a sample is a polymorphic variant of known malware, but is as yet unidentified by AV, an analyst could identify the family it belongs and hence have insight into what disinfection procedures are required and what impact the malware has on an infected system.

1.3 Contributions
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 the Levenshtein distance, the NCD and the BLAST algorithms to perform similarity
comparisons using novel string based malware signatures.

- 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.

The structure of this paper is as follows: Section 2 describes the related work in static malware classification. Section 3 defines the malware classification problem and our approach. Section 4 describes the unpacking and general static analysis component of the system. Section 5 examines string based signatures. Section 6 describes the vector based pre-filtering stage used in classification. This is a coarse grained classification process. Section 7 describes the fine grained classification algorithms. Section 8 describes distance metrics and the nearest neighbour similarity search. Section 9 performs an evaluation using benign and malicious samples. Section 10 examines limitations and discusses points of interest. Section 11 summarizes and concludes the paper.

2 RELATED WORK

The related work reviewed in this paper is limited to static classification approaches because our work is based on static analysis. Dynamic approaches examining malware behaviour are important and the reader is advised to read the survey in [12]. BitBlaze [13] is an example dynamic system based on whole-system emulation which has contributed a number of works in malware analysis. In static analysis, automatic extraction of string signatures has been proposed and employed by commercial Antivirus software [4, 5]. 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 [7, 8] and improve detection rates of malware variants. In [7], 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 [14]. 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 [15]. By extracting relevant n-grams as particular features, feature vectors could be compared in order to construct similarity. N-perms re-examined n-grams and allowed for equivalency between grams irrespective of permutations of their elements. Machine learning and classification extended these systems to detect unknown malware in [16, 17]. 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 [18]. 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. [19] 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 [20].

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 [21, 22]. 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 [23], 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 [10, 24-26]. 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 [27], 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 [28]. Intraprocedural control flow using the control flow graphs have been compared by decompiling control flow graphs of different programs of different sizes. The nearest neighbour search is shown in Fig. 2

The distance function used in the nearest neighbour search is 
\[ d(p,q) = \begin{cases} 0 & \text{if } p = q \\ 1 & \text{otherwise} \end{cases} \]

Fig. 2. The software similarity search to detect malware.

3 PROBLEM STATEMENT AND OUR APPROACH

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 an extension of the definition proposed in [32]. A workflow is shown in Fig. 1.

Definition 1. A program \( q \) is a copy of program \( p \) if it is exactly the same as \( p \) or it is the result of a semantic preserving transformation (e.g., obfuscation, recompilation, or optimisation) over \( p \).

Definition 2. Programs \( p \) and \( q \) are similar if they are derived from the same software source.

Definition 3. Let \( p, q \) be programs. Let \( f \) be a method for extracting a set of characteristics extracted from \( p \). We say \( f(p) \) is a birthmark of \( p \), only if both of the following conditions hold.

- \( f(p) \) is obtained only from \( p \) itself
- \( p \) is a copy of \( q \) if \( f(p) = f(q) \)

Definition 4. Let \( p, q \) be programs or program components. Let \( f(p) \rightarrow a \) and \( f(q) \rightarrow b \) be the birthmarks extracted from \( p \) and \( q \). Let \( s(a,b) \rightarrow [0,1] \) be a similarity function and a value \( e < 1 \). The birthmarking system is resilient if \( p \) and \( q \) are similar and \( 1 - s(a,b) < e \).

Definition 5. Let \( p \) and \( q \) be independently written programs. The software birthmarking system is credible if the system can discriminate between the two programs; that is \( s(f(p),f(q)) < 1 - e \).

The software similarity problem is extended to operate over a database of programs. We use the nearest neighbour search.

Definition 6. Given a set of objects \( P \) and a query \( q \), and a range \( r > 0 \), the range nearest neighbours (rNN) query is to find a result set \( rNN \) that consists of objects such that for any \( p \in rNN \), \( p \in P \) and \( \text{dist}(p,q) \leq r \).

A slight variation is to find any nearest neighbour in range. This variation can improve performance.

Definition 7. Given a set of objects \( P \) and a query \( q \), and a range \( r > 0 \), the any range nearest neighbours (rNN) query is to find any object \( p \), such that \( p \in P \) and \( \text{dist}(p,q) \leq r \).

The distance function used in the nearest neighbour search is \( d(p,q) = 1 - s(p,q) \).

The nearest neighbour search is shown in Fig. 2

3.1 Our Approach

Our approach builds a signature or birthmark of a mal-

Fig. 1. The software similarity problem.
ware based on the set of control flow graphs it has. We compare signatures using distance metrics to show similarity. In our experiments we evaluate constructing strings to represent signatures and then use a variety of string metrics to show signature similarity. We also use a vector based signature which we observe is more effective and efficient than our string signatures. Finally, we add a set of strings signature which we observe as more accurate and is used to refine the vector based result.

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. Q-grams are equivalent to n-grams when using strings from decompiled control flow 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 [33].

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 [34] are used to perform a similarity search.

4 Unpacking and Static Analysis

4.1 Unpacking

The query program may have its real contents hidden using the code packing transformation [35]. 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 [29].

4.2 Control Flow

A basic block is a sequence of instructions that satisfy the following conditions:

- Execution flow can only enter the basic block through the first instruction.
- Execution flow can only exit the block at the last instruction.

A control flow graph is a directed graph representing the possible flow of execution within a procedure. The nodes in the graph represent basic blocks.

Definition 8. The control flow graph of procedure f is the directed graph C=(B,E) such that B is the set of basic blocks and E is the set of edges between them.

A program P is a set of control flow graphs \( P = \{c_1, \ldots, c_n\} \).

4.3 Disassembly and Control Flow Reconstruction

In our system, an unpacked program is disassembled using speculative disassembly [36]. 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 [29] 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 an 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.

4.4 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 [29]. This effectively forms a locality sensitive hash. The structuring algorithm we use is based on the algorithm used in the DCC decompiler [37]. The grammar for the resulting string is shown in Fig. 3. Using alphabet \( \Sigma \). Formally, for program P

---

**Fig. 3. The grammar of a structured string.**

- Execution flow can only enter the basic block through the first instruction.
- Execution flow can only exit the block at the last instruction.

A control flow graph is a directed graph representing the possible flow of execution within a procedure. The nodes in the graph represent basic blocks.

Definition 8. The control flow graph of procedure f is the directed graph C=(B,E) such that B is the set of basic blocks and E is the set of edges between them.

A program P is a set of control flow graphs P={c_1,...,c_n}.

4.3 Disassembly and Control Flow Reconstruction

In our system, an unpacked program is disassembled using speculative disassembly [36]. 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 [29] 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 an 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.

4.4 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 [29]. This effectively forms a locality sensitive hash. The structuring algorithm we use is based on the algorithm used in the DCC decompiler [37]. The grammar for the resulting string is shown in Fig. 3. Using alphabet \( \Sigma \). Formally, for program P

---

**Fig. 3. The grammar of a structured string.**

- Execution flow can only enter the basic block through the first instruction.
- Execution flow can only exit the block at the last instruction.

A control flow graph is a directed graph representing the possible flow of execution within a procedure. The nodes in the graph represent basic blocks.

Definition 8. The control flow graph of procedure f is the directed graph C=(B,E) such that B is the set of basic blocks and E is the set of edges between them.

A program P is a set of control flow graphs P={c_1,...,c_n}.

4.3 Disassembly and Control Flow Reconstruction

In our system, an unpacked program is disassembled using speculative disassembly [36]. 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 [29] 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 an 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.

4.4 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 [29]. This effectively forms a locality sensitive hash. The structuring algorithm we use is based on the algorithm used in the DCC decompiler [37]. The grammar for the resulting string is shown in Fig. 3. Using alphabet \( \Sigma \). Formally, for program P
and for a control flow graph $c$, let $P = \{c \in G\}$. A structuring function for a control flow graph is defined as $s$ and a structuring algorithm for program $P$ is defined as $S$.

$$S : P \rightarrow M$$

$$c \rightarrow s(c)$$

### 5 String Based Signatures

We first experimented with string based signatures. This approach was eventually discarded in favour of representing signatures using vectors, however insight into the malware detection problem is gained by examining these novel techniques.

#### 5.1 Feature Extraction

There is an associated string representing control flow for each procedure identified in the binary. These strings and ordered and concatenated to form a single string to represent the control flow of the entire binary. The substrings are delimitated by a specific character (eg `Z`). The novelty of our approach is to order and concatenate the control flow graph strings into a single unified string based signature, which allows us to use traditional string similarity metrics for malware classification. The order of the concatenated strings is determined by features of the procedure, which are used as sort keys. Procedures that have duplicate sets of keys are removed from the analysis. The keys in order of importance are:

- Number of IL instructions in procedure
- Length of string representing decompiled control flow graph
- Number of basic blocks in procedure
- Number of edges in control flow graph
- Number of procedure’s callers
- Number of procedure’s callees

#### 5.2 Indexing Using String Metric Access Methods

String metrics are proposed to show the similarity between a query signature and malware signatures. A similarity search over the malware database enables the malware variant classification. The string metric we propose is the Levenshtein or edit distance. The Levenshtein distance between two strings gives the minimum number of insertions, deletions and substitutions to transform one string to the other. The run-time complexity is $O(nm)$ where $n$ and $m$ are the lengths of the strings. The Levenshtein distance forms a metric. A metric allows efficient indexing and searching of objects. Sequence alignment methods also provide suitable string distances. The Smith-Waterman algorithm is an optimal local alignment algorithm. We propose using Metric Access Methods to perform a range similarity search. The similarity search finds all malware signatures similar to the query with at most $r$ edit operations to transform the query signature to the malware signature.

String metrics may also be used on the byte-level content of the unpacked malware. We evaluate the effectiveness of using byte-level content in Section 9, and compare it to our proposed signature of using decompiled control flow graphs.

### 5.3 Indexing Using Genome Strings and Blast

The Smith-Waterman algorithm gives the optimal local sequence alignment between two strings. The local sequence alignment seeks to provide an alignment between two strings taking into account the alignment of substrings. Local sequence alignment is used often in the field of Bioinformatics to identify similarity between genome sequences. It forms a metric allowing for Metric Access Methods for indexing and searching. The Smith-Waterman algorithm has quadratic run-time complexity like the Levenshtein distance. A quadratic running time has poor efficiency when the length of the strings becomes moderately large. The Basic Local Assignment Search Tool (BLAST) approximates the Smith-Waterman algorithm using a heuristic search. BLAST is used frequently to improve the efficiency of genome searches. We propose using off-the-shelf BLAST software to perform similarity searches of our malware signatures. To do this, we translate our control flow graph signatures to a protein string in the FASTA format to be used as input to the BLAST software. To construct a protein sequence, the decompiled string is translated character by character to a genome identifier. The BLAST algorithm does not employ distance metrics for the similarity search, but uses the notion of an expected value, which describes the statistical probability of the occurrence of a random signature.

The use of off-the-shelf genome similarity search software is a novel aspect used by our approach, and to the best of our knowledge has not been proposed in earlier research.

#### 5.4 Indexing Using the NCD Metric Access Method

We propose using the normalized compression distance (NCD) to perform a similarity search. The NCD utilises the notion of compressed objects being related to Kolmorogov complexity. The NCD takes note that when two objects are related, compressing the concatenated objects results in a blob of similar length to compressing only one of the objects. The NCD provides a measure of dissimilarity or distance between objects without explicit knowledge or representation of the internal structure of the objects in question. It is able to provide a distance measure using many existing compression algorithms without modification. The NCD is defined as:

$$NCD(x, y) = \frac{C(xy) - \min\{C(x), C(y)\}}{\max\{C(x), C(y)\}}$$

where $C(x)$ is the length of the compressed object, and $C(xy)$ is the length of the compressed concatenated objects.

For the NCD to perform effectively, the size of the objects must be less than the compressors window size. The NCD is a metric and so can employ the use of Metric Access Methods to index and search the signatures. To the best of our knowledge, Metric Access Methods have not been used in conjunction with the NCD and malware indexing by previous research.
6 Vector Based Signatures – Pre-filtering

To reduce the search space for potentially related malware, we use an initial similarity search to select candidate malware variants. We chose a vector based approach because during our evaluation we observed that this approach was more efficient and effective. Note that in our final system we use both vector based signatures for pre-filtering and the set of strings based signatures for refinement.

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.

6.1 The K-Subgraph Feature

Using subgraphs of size k to characterize control flow has been investigated in previous literature [11]. 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 [11].

Given a subgraph of size k, the graph is transformed into a unique and canonical representation using the Bliss open-source toolkit [40]. 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. An example of possible k-subgraph features from a control flow graph when k is 7 is shown in Fig. 4.

6.2 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.

6.3 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. Feature selection works by counting the number of times each feature occurs in the training set and then ranking them in descending order. The top 500 were our selected features. Frequency of features forms our feature selection and is reasonable considering we are performing a nearest neighbour search. If we were performing malware detection using binary classification then another form of feature selection would be more suitable, for example, Mutual Information. The number of features, 500, was chosen to replicate previous work used in n-gram classification. We noted no significance to the accuracy of the system when this number was increased further. We did notice that decreasing this number using dimensionality reduction did decrease the accuracy as explained in section 6.4. 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.

6.4 Dimensionality Reduction

To reduce the dimensionality of the feature vector obtained from the previous stage, Principal Component Analysis (PCA) [41] can be employed. Our pilot studies performed more effectively when PCA was not used. We do not consider dimensionality reduction any further. Fig. 8 illustrates the process of feature selection and dimensionality reduction.

6.5 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. This distance is also reportedly more robust for high dimensional data when compared to the Euclidean distance. The
more familiar cosine similarity measure is not used in our work because it is not a metric distance function and therefore does not allow for efficient database indexing. The Manhattan distance is also known as the city block or L1 distance. For n-dimensional vectors \( p \) and \( q \), the Manhattan distance is:

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

6.6 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:

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

The results of the similarity search are candidate matches that can be used in the subsequent stage of comparing programs using the assignment problem. The nearest neighbours of the query enable us to determine if those neighbours are variants of the query. All samples available are typically used for the queries once the database is created. The nearest neighbours of those queries, as described in the introduction, can identify polymorphic variants, group samples by their family, or enable incident response to identify clusters of infection. More discussions of the algorithms and implementation of the similarity search are given in Section 8.

7 Set of Strings Based Signatures – 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.

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

The linear sum assignment problem is to 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.

The linear sum assignment problem is formally defined as:

Given two sets, \( A \) and \( T \), of equal size, together with a weight function \( C : A \times T \rightarrow \mathbb{R} \). Find a bijection \( f : A \rightarrow T \) such that the cost function:

\[
\sum_{a \in A} C(a, f(a))
\]

is minimized.

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 use the Smith-Waterman algorithm. 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 [42] 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, \( b_r \) used in the normalization process are place holders and not used for any other purpose.

\[
M_1 = S(P_1)
\]

\[
M_2 = S(P_2)
\]

\[
M_1 = \{a_i \in M_1 \cup \{b_r\} \mid 1 \leq i \leq |M_1| < j \leq |M_2|
\]

\[
M_2 = \{a_i \in M_2 \cup \{b_r\} \mid 1 \leq i \leq |M_2| < j \leq |M_1|
\]

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

\[
C : M_1' \times M_2' \rightarrow \mathbb{R}
\]

\[
C(a, b) = \begin{cases} |a|, & \text{if } a \in M_1, b \notin M_2 \\ |b|, & \text{if } b \in M_2, a \notin M_1 \\ ed(a, b), & \text{if } a \in M_1, b \in M_2 \end{cases}
\]

Find a bijection \( f : M_1' \rightarrow M_2' \) such that the distance, \( d \), is minimized.

\[
d = \sum_{a \in M_1'} C(a, f(a))
\]

7.2 Solutions to the Assignment Problem

The assignment problem can be solved optimally using the Munkres or Hungarian algorithm [43] 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.

7.3 Similarity Search of Malware

The similarity between two objects is given by:

\[ s(p, q) = 1 - \frac{d(p, q)}{|q|} \]

We scale the distance relative to our query so that we can perform a range search relative to only the query using an efficient metric access method. Because we scale to the query and not max(1, |p|, |q|) we have cases where d(p, q) > |q|. In this case our similarity function would give us a negative result. To simplify interpretation of this result, we say that it is not at all similar and discard it.

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 neighbor \( p \) to the query \( q \) in the set of malware, \( E \), returned by our pre-filtering process.

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

8 Nearest Neighbour Similarity Searches

8.1 Metric Distance Functions

A metric on a set \( X \) is a function (known as the distance function or distance)

\[ d : X \times D \rightarrow \mathbb{R} \]

For all \( x, y, z \) in \( X \), this function is required to satisfy the following conditions:

1. \( d(x, y) \geq 0 \)
2. \( d(x, y) = 0 \) if and only if \( x = y \)
3. \( d(x, y) = d(y, x) \)
4. \( d(x, z) \leq d(x, y) + d(y, z) \) (triangle inequality)

If the distance function has the properties of a distance metric then indexing and searching a database can be performed more efficiently. Therefore it is beneficial to compare objects or birthmarks (software fingerprints) using distance functions that are metric. Examples of metric access methods are in [33, 34, 44].

8.2 Similarity Search Using Metric Access Methods

To search for malware that are similar to our query in both the pre-filtering and classification stages, a metric access method is employed. Metric trees encapsulate data structures including BK Trees [45], VP Trees [33] or dynamic indexing structures such as M-Trees [44] and Slim-Trees [46]. Our implementation uses the GBDI metric access method library [47].

In our prototype, we use a Vantage Point Tree [33] for indexing the feature vectors used in the pre-filtering stage. Then, the final classification process uses DBM-Tree [34] to perform a similarity search. Note that our classification system uses two levels of indexing and has different metric access methods for each.

In our prototype we can configure the similarity search to return either any similar objects, or all similar objects. We use the any range search for classification, and the all range search for pre-filtering. By performing a similarity search to find any similar object, the performance is significantly improved when there are many near duplicate malware stored in the database. The any range search was implemented by us into the GDBI Arboretum library [47].

9 Implementation and Evaluation

9.1 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++. Emulation is used to perform unpacking. However, the classification process uses only static analysis and that is the focus of our current work.

9.2 Effectiveness of String Signatures

The first evaluation we performed was to examine the similarity matrices for our string based signature classification algorithms on a known family of related malware. We also compared these methods to the q-gram approach. The system ideally identifies high similarity between malware that belongs to the same family. The malware chosen was the Rorol family of malware to replicate previous research [10, 28, 29]. 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 [48]. Identified malware variants have similarities exceeding or equal to 0.6. Identified variants additionally have their table cells highlighted. The more cells highlighted the more effective each approach is. We also evaluate using only the byte-level content for similarity comparisons. The Rorol malware family is not strongly polymorphic, so byte-level content is still somewhat effective. The most important observation is that comparing string based signature approaches in Table 1 to the q-gram approach in Table 2 shows that the q-gram vector based signature detects more malware variants. It is also noted that q-grams are theoretically more efficient. It is for these reasons we decided to focus on vector based signatures.

9.3 Evaluation Setup

To perform more evaluations 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 [49] 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 [50] executable directories. The system we used...
TABLE 1

SIMILARITY MATRICES FOR RORON 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>o</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>0.60</td>
<td>0.35</td>
<td>0.38</td>
<td>0.45</td>
<td>0.74</td>
<td>0.60</td>
<td>0.60</td>
<td>0.73</td>
</tr>
<tr>
<td>0.70</td>
<td>1.00</td>
<td>0.46</td>
<td>0.50</td>
<td>0.37</td>
<td>0.73</td>
<td>0.95</td>
<td>0.96</td>
<td>0.73</td>
</tr>
<tr>
<td>0.35</td>
<td>0.46</td>
<td>1.00</td>
<td>0.64</td>
<td>0.59</td>
<td>0.36</td>
<td>0.46</td>
<td>0.46</td>
<td>0.35</td>
</tr>
<tr>
<td>0.38</td>
<td>0.50</td>
<td>0.64</td>
<td>1.00</td>
<td>0.61</td>
<td>0.42</td>
<td>0.49</td>
<td>0.50</td>
<td>0.40</td>
</tr>
<tr>
<td>0.45</td>
<td>0.37</td>
<td>0.59</td>
<td>0.61</td>
<td>1.00</td>
<td>0.47</td>
<td>0.37</td>
<td>0.37</td>
<td>0.46</td>
</tr>
<tr>
<td>0.74</td>
<td>0.73</td>
<td>0.36</td>
<td>0.42</td>
<td>0.47</td>
<td>1.00</td>
<td>0.73</td>
<td>0.72</td>
<td>0.86</td>
</tr>
<tr>
<td>0.60</td>
<td>0.95</td>
<td>0.46</td>
<td>0.49</td>
<td>0.37</td>
<td>0.73</td>
<td>0.96</td>
<td>1.00</td>
<td>0.72</td>
</tr>
<tr>
<td>q</td>
<td>0.60</td>
<td>0.96</td>
<td>0.46</td>
<td>0.50</td>
<td>0.37</td>
<td>0.73</td>
<td>0.96</td>
<td>1.00</td>
</tr>
<tr>
<td>a</td>
<td>0.73</td>
<td>0.73</td>
<td>0.35</td>
<td>0.40</td>
<td>0.46</td>
<td>0.86</td>
<td>0.72</td>
<td>0.72</td>
</tr>
</tbody>
</table>

Levenshtein String Metric on Byte-Level Content

<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>o</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>0.70</td>
<td>0.42</td>
<td>0.42</td>
<td>0.44</td>
<td>0.72</td>
<td>0.70</td>
<td>0.70</td>
<td>0.70</td>
</tr>
<tr>
<td>0.70</td>
<td>1.00</td>
<td>0.47</td>
<td>0.47</td>
<td>0.48</td>
<td>0.94</td>
<td>1.00</td>
<td>1.00</td>
<td>0.93</td>
</tr>
<tr>
<td>0.42</td>
<td>0.47</td>
<td>1.00</td>
<td>0.71</td>
<td>0.80</td>
<td>0.48</td>
<td>0.47</td>
<td>0.47</td>
<td>0.48</td>
</tr>
<tr>
<td>0.42</td>
<td>0.47</td>
<td>0.71</td>
<td>1.00</td>
<td>0.72</td>
<td>0.47</td>
<td>0.47</td>
<td>0.47</td>
<td>0.47</td>
</tr>
<tr>
<td>0.44</td>
<td>0.48</td>
<td>0.80</td>
<td>0.72</td>
<td>1.00</td>
<td>0.49</td>
<td>0.48</td>
<td>0.48</td>
<td>0.50</td>
</tr>
<tr>
<td>0.72</td>
<td>0.72</td>
<td>0.94</td>
<td>0.48</td>
<td>0.47</td>
<td>0.49</td>
<td>1.00</td>
<td>0.94</td>
<td>0.94</td>
</tr>
<tr>
<td>0.70</td>
<td>1.00</td>
<td>0.47</td>
<td>0.47</td>
<td>0.48</td>
<td>0.94</td>
<td>1.00</td>
<td>1.00</td>
<td>0.93</td>
</tr>
<tr>
<td>q</td>
<td>0.70</td>
<td>1.00</td>
<td>0.47</td>
<td>0.47</td>
<td>0.48</td>
<td>0.94</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>a</td>
<td>0.70</td>
<td>0.93</td>
<td>0.48</td>
<td>0.47</td>
<td>0.50</td>
<td>0.96</td>
<td>0.93</td>
<td>0.93</td>
</tr>
</tbody>
</table>

Normalized Compression Distance (NCD) Metric

<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>o</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.94</td>
<td>0.80</td>
<td>0.50</td>
<td>0.52</td>
<td>0.52</td>
<td>0.82</td>
<td>0.80</td>
<td>0.80</td>
<td>0.82</td>
</tr>
<tr>
<td>0.80</td>
<td>0.92</td>
<td>0.51</td>
<td>0.54</td>
<td>0.53</td>
<td>0.88</td>
<td>0.93</td>
<td>0.93</td>
<td>0.89</td>
</tr>
<tr>
<td>0.50</td>
<td>0.51</td>
<td>0.93</td>
<td>0.77</td>
<td>0.83</td>
<td>0.52</td>
<td>0.51</td>
<td>0.51</td>
<td>0.52</td>
</tr>
<tr>
<td>e</td>
<td>0.52</td>
<td>0.54</td>
<td>0.77</td>
<td>0.94</td>
<td>0.85</td>
<td>0.54</td>
<td>0.54</td>
<td>0.54</td>
</tr>
<tr>
<td>0.52</td>
<td>0.53</td>
<td>0.83</td>
<td>0.85</td>
<td>0.93</td>
<td>0.53</td>
<td>0.53</td>
<td>0.53</td>
<td>0.53</td>
</tr>
<tr>
<td>k</td>
<td>0.82</td>
<td>0.88</td>
<td>0.52</td>
<td>0.54</td>
<td>0.53</td>
<td>0.94</td>
<td>0.88</td>
<td>0.88</td>
</tr>
<tr>
<td>m</td>
<td>0.80</td>
<td>0.93</td>
<td>0.51</td>
<td>0.54</td>
<td>0.53</td>
<td>0.88</td>
<td>0.93</td>
<td>0.93</td>
</tr>
<tr>
<td>q</td>
<td>0.80</td>
<td>0.93</td>
<td>0.51</td>
<td>0.54</td>
<td>0.53</td>
<td>0.88</td>
<td>0.93</td>
<td>0.93</td>
</tr>
<tr>
<td>a</td>
<td>0.82</td>
<td>0.89</td>
<td>0.52</td>
<td>0.54</td>
<td>0.53</td>
<td>0.92</td>
<td>0.89</td>
<td>0.89</td>
</tr>
</tbody>
</table>

9.4 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 3 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.

9.5 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. 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 [28].
Our system did not report any samples as belonging to the family by the system. The number of false positives gives indication of how the distance functions perform using non similar programs. In our first test we simply aggregated the families of malware from our true positive testing. Our system did not report any samples as belonging to incorrect families. We then implemented a more thorough test of our system. We performed an evaluation using a much larger malware database size of 10,000. We classified the set of 1601 benign programs and expected that any identified malware would be a false positive. The evaluation demonstrates false positives when using the q-gram as is also demonstrated in Table 5. 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 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 [29]. 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 4. The results show that our prototype detects more malware variants in this family of malware than existing systems.

The Netsky, Ronor, 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 results of our algorithm and existing algorithms [28, 29]. Table 4 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.

### 9.6 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. In our first test we simply aggregated the families of malware from our true positive testing. Our system did not report any samples as belonging to incorrect families. We then implemented a more thorough test of our system. We performed an evaluation using a much larger malware database size of 10,000. We classified the set of 1601 benign programs and expected that any identified malware would be a false positive. The evaluation demonstrates false positives when using the q-gram as is also demonstrated in Table 5. 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 expect 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.

#### 9.7 Algorithmic Complexity Analysis

The algorithmic complexity of comparing malware signatures is shown in Table 6. We examine our string based signature based on using the optimal edit distance, the vector-based signature using the Manhattan distance, and the set of strings-based signature using the optimal and greedy approach to solving the assignment problem. We also compare our approach with previous work in SMIT, exact control flow graph matching in [28], and traditional graph algorithms. Our vector-based signature is the most efficient and a distance between signatures can be performed in O(1) relative to the size of the programs in

---

### Table 3: Num. False positives using k-subgraphs and Q-grams.

<table>
<thead>
<tr>
<th>Similarity</th>
<th>K-Subgraphs</th>
<th>QGrams</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>356845</td>
<td>40055</td>
</tr>
<tr>
<td>0.3</td>
<td>285202</td>
<td>7899</td>
</tr>
<tr>
<td>0.4</td>
<td>200526</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 4: Number of Malware Detected

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Klez</th>
<th>Netsky</th>
<th>Ronor</th>
<th>Frethem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum</td>
<td>36</td>
<td>49</td>
<td>81</td>
<td>269</td>
</tr>
<tr>
<td>Exact</td>
<td>20</td>
<td>29</td>
<td>17</td>
<td>139</td>
</tr>
<tr>
<td>Heuristic Approx.</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</td>
<td>20</td>
<td>43</td>
<td>73</td>
<td>217</td>
</tr>
</tbody>
</table>

### Table 5: False positives.

<table>
<thead>
<tr>
<th>Classification</th>
<th>Num. 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>

### Table 6: Algorithmic Complexity Comparisons

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMIT</td>
<td>O(N^3)</td>
</tr>
<tr>
<td>Exact Matching</td>
<td>O(NlogN)</td>
</tr>
<tr>
<td>Graph Edit Distance</td>
<td>NP</td>
</tr>
<tr>
<td>Graph Isomorphism</td>
<td>NP</td>
</tr>
<tr>
<td>String</td>
<td>O(ken[N]^2)</td>
</tr>
<tr>
<td>Vector</td>
<td>O(1)</td>
</tr>
<tr>
<td>Set of Strings-Optimal</td>
<td>O(N^2)</td>
</tr>
<tr>
<td>Set of Strings-Approximate</td>
<td>O(N(N+1)/2)</td>
</tr>
</tbody>
</table>

---

Fig. 5. Malware (top) and benign (bottom) sample processing times.
terms of number of procedures. This is why our system performs so efficiently. The string-based signature performs quite slowly because each procedure incurs a cost, k, relative to the size of the procedures’ decompiled control flow graphs. Our set of strings-based distance can be performed in \( O(N^2) \) which is comparable to the previous research in SMIT [25] which uses an approximation to the graph edit distance on the programs’ call graphs. For large graphs we can use the approximate algorithm in our approach which performs in \( O(N(N+1)/2) \) and is more efficient than SMIT. The exact matching algorithm was proposed in our previous research but does not perform approximate matching of control flow graphs which our current work does. The classical single graph based distance and equality algorithms are either in \( \text{NP} \) (graph edit distance) or believed to be in \( \text{NP} \) (graph isomorphism) making control flow intractable when used for a signature.

9.8 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 Fig. 5. The malware processing times are higher in general. The median time for processing malicious samples is 0.84 seconds. 90% of the samples could be processed in under 1.31 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. 90% of samples could be processed in under 0.56 seconds. 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 [28] 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 an Email gateway system.

10 LIMITATIONS AND DISCUSSION

10.1 Code Packing

A malware obfuscation technique commonly employed to resist static analysis is packing. Malware packing that encrypts, compresses, or obfuscates the code contents and then later regenerates the original program needs to be removed. The majority of packed samples can be automatically unpacked, but there exist binaries which evade this analysis. Instruction virtualization [51, 52] is resistant to an entirely automated static analysis. Instruction virtualization implements an emulator which interprets bytecode representing the hidden code. Therefore, the hidden code in its original unpacked form is never revealed. If unpacking cannot be achieved by a malware classification system, then the packing tool may be classified instead of the packed contents. It is probably advantageous for Antivirus to blacklist programs that cannot be unpacked. Manually written static unpackers can be developed on a case by case basis and this is what is traditionally employed by commercial Antivirus. A better approach is to detect packed programs and flag them as suspicious. Benign programs that are packed can be whitelisted. The scope of our system is limited to malware that can be unpacked using the approach of application level emulation. Application level emulation is fast but because of its limited use of a faithful emulator, malware can detect its presence and therefore change its behaviour. Unpackers such as Renovo [53] employ whole system emulation and are more resistant to detection. The current problem is that such systems have poor performance in terms of real-time constraints. Another approach is to unpack on the fly during program run-time by monitoring memory access, as is done by OmniUnpack [54]. This system claims real-time performance suitable for Antivirus. Such a system could be combined with our work to make a real-time malware classification capable of unpacking most or all non instruction virtualization based malware.

10.2 Obfuscation

For the most part, code packing is the obfuscating process employed by malware authors. Therefore, once a sample has been unpacked, analysts have access to the original unobfuscated image. This is becoming prevalent as malware becomes more like traditional software development and malware authors employ high level languages to implement their works.

Control flow can be obfuscated but this is typically not present in most malware today. Code insertion, deletion, substitution, and reordering within a basic block does not affect the structured control flow that our system uses. This makes control flow a more invariant program representation than traditional byte-level signatures. If control flow is modified, then our system can perform an approximate match. The changes to the decompiled strings should show the changes locally. The global view of the strings should still retain similarity. Through the normal process of software development and evolution, decompiled strings of control flow graphs can identify those
changes while still identifying them as variants.

Obfuscations such as opaque predicates which add conditional branches which always evaluate to the same path but are hard to determine statically present a bigger problem. Unless opaque predicates account for the majority of the control flow, our system should still detect the malware as a variant. Other obfuscations including negating conditions and swapping the branches resulting in different decompiled strings. A solution to this could involve using an unordered Abstract Syntax Tree (AST) instead of a string. If malware in the future obfuscates control flow like this, we may consider using k-subtrees of the AST instead of q-grams of the decomposed strings.

11 Conclusion

Malware can effectively be characterized by its control flow. We proposed a malware classification system using approximate matching of control flow graphs. We first tried using string signatures to describe malware. We then 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


Silvio Cesare is a PhD candidate at Deakin University receiving a DUPRA scholarship. His research interests include analysing software for theft, vulnerabilities, bugs, or malicious intent. He has previously been employed as a scanner architect in the network vulnerability assessment and management industry. He has a Bachelor in Information Technology and a Master of Informatics from CQUniversity. He is a member of the IEEE and lives in Melbourne, Australia.

Yang Xiang received his PhD in Computer Science from Deakin University, Australia. He is currently a full professor at School of Information Technology, Deakin University. He is the Director of the Network Security and Computing Lab (NSCLab). His research interests include network and system security, distributed systems, and networking. He has published more than 130 research papers in international journals and conferences. He has served as the Program/General Chair for many international conferences such as ICA3PP 12/11, IEEE/IFIP EUC 11, IEEE TrustCom 13/11, IEEE HPCC 10/09, IEEE ICPADS 08, NSS 11/10/09/08/07. He serves as the Associate Editor of IEEE Transactions on Computers, IEEE Transactions on Parallel and Distributed Systems, Security and Communication Networks (Wiley), and the Editor of Journal of Network and Computer Applications. He is the Coordinator, Asia for IEEE Computer Society Technical Committee on Distributed Processing (TCDP). He is a Senior Member of the IEEE.

Wanlei Zhou received his PhD degree in 1991 from the Australian National University, Canberra, Australia, and the DSc degree from Deakin University, Victoria, Australia, in 2002. He is currently the chair professor of Information Technology and the Head of School of Information Technology, Deakin University, Melbourne. His research interests include distributed and parallel systems, network security, mobile computing, bioinformatics, and e-learning. He has published more than 200 papers in refereed international journals and refereed international conference proceedings. Since 1997, he has been involved in more than 50 international conferences as the general chair, a steering chair, a PC chair, a session chair, a publication chair, and a PC member. He is a senior member of the IEEE.