2010 24th IEEE International Conference on Advanced Information Networking and Applications

A Fast Flowgraph Based Classification System for Packed and Polymorphic Malware on the Endhost

Silvio Cesare and Yang Xiang
School of Management and Information Systems
Centre for Intelligent and Networked Systems
Central Queensland University
Rockhampton, Queensland 4702, Australia
silvio.cesare@gmail.com; y.xiang@cqu.edu.au

***Abstract*—Identifying malicious software provides great benefit for distributed and networked systems. Traditional real-time malware detection has relied on using signatures and string matching. However, string signatures ineffectively deal with polymorphic malware variants. Control flow has been proposed as an alternative signature that can be identified across such variants. This paper proposes a novel classification system to detect polymorphic variants using flowgraphs. We propose using an existing heuristic flowgraph matching algorithm to estimate graph isomorphisms. Moreover, we can determine similarity between programs by identifying the underlying isomorphic flowgraphs. A high similarity between the query program and known malware identifies a variant. To demonstrate the effectiveness and efficiency of our flowgraph based classification, we compare it to alternate algorithms, and evaluate the system using real and synthetic malware. The evaluation shows our system accurately detects real malware, performs efficiently, and is scalable. These performance characteristics enable real-time use on an intermediary node such as an Email gateway, or on the endhost.**

Keywords-network security; malware; structural classification; unpacking; emulation

I. INTRODUCTION

Malware, short for malicious software, means a variety of forms of hostile, intrusive, or annoying software and program code. Malware is a significant problem in distributed computer systems and in endhost security. To prevent malware from causing damage to systems, untrusted programs are examined to identify those that are malicious. Identifying malware before it is allowed to execute its malicious intent improves system security.

To identify a program as being malicious or benign, automated analysis is required. The analysis can employ either a static or dynamic approach. In the dynamic approach, the malware is executed, possibly in a sandbox, and its runtime behaviour examined. In the purely static approach, the malware is never executed. Traditional solutions to secure the endhost against malware have focused on static detection.

A secure endhost system must detect malicious content, while simultaneously responding to the user's productivity demands. This dual requirement mandates that the malware detection system must have high performance. Without adequate performance, the experience and productivity of the user can be severely degraded. Traditional solutions to static malware detection have employed the use of signatures. Signatures capture invariant characteristics, or fingerprints, in the malware that uniquely identify it. Because of performance constraints, the most predominantly used signature is a string containing patterns of the raw file content [1, 2]. This allows for a string search [3] to quickly identify patterns associated with known malware. However, these patterns can easily be invalidated because minor changes to the malware source code have significant effects on the malware's raw content. Thus, traditional signatures can prove ineffective when dealing with unknown variants.

Detecting malware variants provides significant reward in endhost security. In this paper we describe malware variants with the umbrella term of polymorphism. Polymorphism describes related malware sharing a common history of code. Code sharing among variants can have many sources, whether derived from autonomously self mutating malware, or manually copied by the malware creator to reuse previously authored code.

Static analysis to identify control flow has been proposed as an alternative characteristic to fingerprint polymorphic malware [4]. Control flow describes the possible execution paths of the program or program procedure and is represented as a directed graph known as a flowgraph. Graphing the internal control flow of a procedure generates a control flow graph, while graphing the control flow between procedures generates a call graph. These flowgraphs based characteristics are identified as more invariant in polymorphic instances of a strain of malware [5]. Approximate matchings of flowgraph based characteristics can be used in order to identify a greater number of malware variants. Detection of variants is possible even when more significant changes to the malware source code are introduced.

To hinder static analysis and extraction of characteristics from malware, a code packing transformation [6] is often used to hide, encrypt, compress, or obfuscate the malware's real content [7, 8]. Code packing is a post-processing transformation applied to malware after compilation. At runtime in a packed binary, the original content is revealed for

1550-445X/10 $26.00 © 2010 IEEE
DOI 10.1109/AINA.2010.121

subsequent execution. Unpacking is a requirement to reveal the hidden characteristics of malware allowing flowgraph based classification to perform with a high degree of accuracy.

In this paper we present a system that employs static analysis to automatically identify variants to existing and known malware based on identifying common or isomorphic control flow graphs. While our system is primarily static, a dynamic analysis that simulates malware execution is employed additionally to perform code unpacking as a preprocessing stage before classification.

This paper makes the following contributions. First, we propose a novel and fast algorithm to determine program similarity by estimating isomorphisms between control flow graphs. These flowgraphs are identified from a dictionary of known malware forming a pre-populated database. Second, we implement and evaluate our idea using a novel prototype system. Additionally, our system employs application level emulation to perform automated unpacking. Finally, we demonstrate our system is fast enough for endhost adoption.

The structure of this paper is as follows. Section II describes related work in static malware classification. Section III describes the problem of software similarity and instance based learning in malware classification. Section IV describes the design and implementation of our prototype system. Section V performs complexity analysis and comparisons to existing systems. Section VI evaluates the prototype system using real and synthetic malware samples, and compares it to previous approximate flowgraph matching systems. Section VII examines the limitations of the research. Finally, Section VIII summarizes and concludes the paper.

II. Related Work

A variety of static approaches to malware classification have been proposed [9-12]. Windows API call sequences have been proposed [10] as an invariant characteristic in malware, but correctly identifying the API calls can be problematic [13] when code packing obscures the result. A variation of n-grams, coined n-perms [11] has been proposed to describe byte level malware characteristics which are subsequently used in a classifier. An approach employing edit distances, inverted indexes and bloom filters using the basic blocks of malware [14] was proposed. This approach using larger synthetic databases was found to have unsuitable speed for endhost use. The problem with these byte level approaches is that the characteristics used are less resilient to changes than compared to using flowgraph based features.

Malware classification using approximate matching of call graphs has been proposed in the past [4, 15, 16]. Call graph based classification approximates the graph edit distance, which allows the construction of similarity between two call graphs. These research systems greedily identify matching nodes between graphs. The identification of matching nodes has been proposed using the small primes product, checksums and control flow graph signatures [4, 15, 16]. The commercial system Vxclass [17] implements a system to identify malware variants, but without real-time performance. Binhunt [18] proposes a sound approach for flowgraph similarity by identifying the maximum common subgraph, yet with significantly worse execution speed. SMIT [19] identifies variants using minimum cost bipartite graph matching and the Hungarian algorithm, improving upon the greedy approach to graph matching. SMIT additionally uses metric trees and a multiresolution database to improve performance on graph based database searches. Our system performs complete classification, including unpacking, and with higher performance than previous systems. Intuitively, higher performance can be achieved by sacrificing the comparison of edges in the call graphs.

Control flow graphs have been proposed as alternate characteristics to call graphs. Using decompiled flowgraphs as a signature amenable to the string edit distance was proposed and shown to be effective [20], but has poor runtime performance. An efficient approach was proposed in an offline network analysis to identify structurally similar worms by identifying common subgraphs of size k [5]. This approach while seemingly more efficient than many other research systems was not demonstrated to perform fast enough for endhost use and did not employ a malware database. Additionally, malware unpacking was not performed on the content that was classified.

An efficient approach using tree automata [21] was proposed to recognize isomorphisms and subgraph ismorphisms in whole program control flow graphs. This approach did not perform approximate matching of malware, thus reducing its detection accuracy. Additionally, unpacking was not performed by the system, also reducing its effectiveness. Our work is distinguished by being more efficient than existing systems of comparable effectiveness.

III. The Software Similarity Problem

The software similarity problem is to determine the similarity between programs. The pairwise similarity between two programs is represented as a real number between 0 and 1 – 0 indicating the programs are not at all similar, and 1 indicating identical programs. Invariant characteristics between programs are used to calculate similarity. In the detection of software theft, these characteristics are known as software birthmarks. In plagiarism detection, software metrics are employed. Dynamic or static approaches are used to identify characteristics. An extension to pairwise similarity is the searching of similar objects contained in a database. This search is known as a similarity search.

Our anti-malware system uses a static approach to determine software similarity and identifies variants using a pre-populated database of malware characteristics. This approach is equivalent to a range or similarity search on the malware database to identify neighbours to the query. Classification is a case of instanced based learning, and works by identifying similar training examples. In our system, a similarity equal to or exceeding 0.6 identifies a variant.

IV. System Design And Implementation

In this section, the design and implementation of the prototype system is examined.

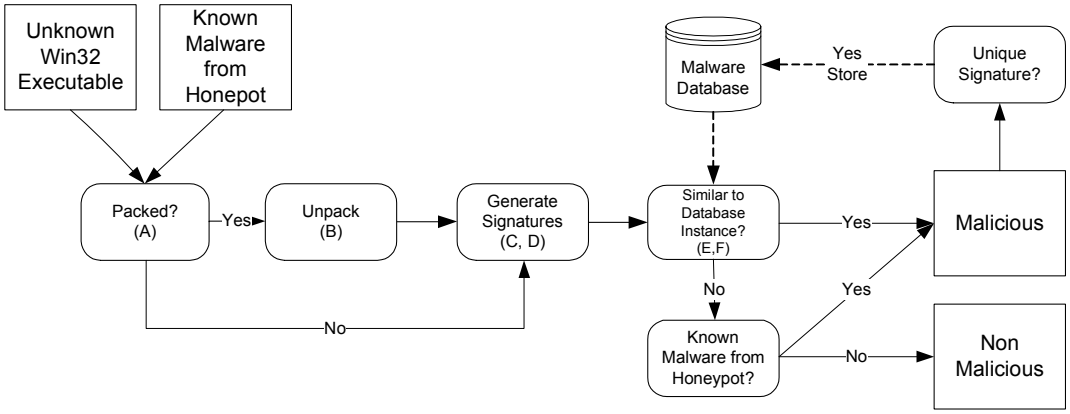

Figure 1. Block diagram of the malware classification system.

Binaries that have undergone a code packing transformation are statically identified. Dynamic analysis is employed for the malware unpacking. Static analysis then proceeds on the unpacked malware image to generate characteristic signatures and perform classification. If the binary is classified as malicious, or known to be malicious because it originates from a honeypot, and is less than 95% similar to existing malware, then the new signature is stored in the database.

The system design is shown in Fig. 1. Subsection A describes how identification of packed binaries occurs. Subsection B describes the automated unpacking system. Subsection C describes the control flow generation. Subsection D describes the signature generation. Subsection E and F describe the classification algorithm.

A. Identifying Packed Binaries

Packed binaries are identified to determine if static analysis should proceed immediately or if unpacking should begin. Entropy analysis [22] is used to detect binaries that have undergone the code packing transformation.

B. Automated Unpacking

The prototype system employs the use of an application level emulator [20] to unpack malware. The x86 instruction set architecture and the Windows API are emulated. Hidden code is naturally revealed during simulation in the malware's process image as dynamically generated code. This hidden code is extracted for use as described in the following sections.

C. Control Flow Graph Generation

Static analysis is performed on the unpacked malware image. The image is disassembled using speculative disassembly [23]. Data consisting of only zeros are ignored. Procedures are identified during disassembly as the target of a call instruction. An additional heuristic to eliminate incorrectly identified procedures is performed by culling procedures whose callers are not also identified as procedures.

The disassembly is translated to an architecture independent intermediate language. The intermediate language is then analysed to construct a control flow graph for each procedure. Control flow graphs are then simplified to eliminate redundant nodes.

D. Flowgraph Signatures

It is possible to generate a flowgraph signature using a fast and simple method because the flowgraph matching algorithm only identifies isomorphisms [4]. This approach takes note that if the signatures or graph invariants of two graphs are not the same, then the graphs are not isomorphic. The converse, while not strictly sound, is used as a good estimate to indicate an isomorphism. To generate a signature, the algorithm orders the nodes in the control flow graph using a depth first order, although other orderings are equally sufficient. A signature subsequently consists of a list of graph edges for the ordered nodes, using the node ordering as node labels. This signature can be represented as a string. An example flowgraph and signature is shown in Fig. 2.

To improve the performance, a hash of the string signature can be used instead. CRC64 is used in our prototype system. The advantage of this isomorphism or exact matching algorithm over an approximate matching algorithm is that classification using exact matches can perform very efficient signature searches using binary search trees. Potentially, other related data structures and algorithms such as perfect hashing could be employed.

The normalized weight of procedure x, is defined as:

$$weight_x = \frac{B_x}{\sum_i B_i}, \tag{1}$$

where B_i is the number of basic blocks of procedure i in the binary.

The similarity ratio between two flowgraphs with signatures x and y is:

$$w_{ed} = \begin{cases} 1, & x = y \\ 0, & x \neq y \end{cases}. \tag{2}$$

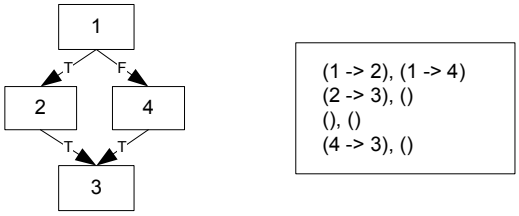

Figure 2. A depth first ordered flowgraph and its signature.

E. Malware Classification

To classify an input binary, the analysis makes use of a malware database. The database contains the sets of control flow graph signatures, represented as strings, and their associated weights for known malware. To classify the input binary, a similarity is constructed between the set of the binary's flowgraph strings and each set of flowgraphs associated with malware in the database.

The Dice coefficient [24] is a measure of similarity between two sets. For weighted sets A and B, the Dice coefficient is:

$$s(A,B) = \frac{2\sum_i w_i x \in A_i \cap B_i}{\sum_i w_i x \in A_i + \sum_i w_i x \in B_i}. \quad (3)$$

Set cardinality is equal to the sum of the weights of the elements, and when normalized is equal to 1, so the denominator in the Dice coefficient becomes 2, cancelling with the multiplier in the numerator. The Dice coefficient then becomes the sum of weights of the intersecting elements.

In our research, each flowgraph of the query binary which matches a signature of a malware flowgraph is processed. The malware flowgraph is ignored for subsequent matches. To perform the matching of a flowgraphs, searching of flowgraph signatures utilises an in memory associative mapping. Future work would see the use of persistent data structures such as B+Trees to avoid loading the database into memory.

Each matched flowgraph is associated with two weights. First, the weight associated with the malware's flowgraph, which is stored in the malware database. Second, the weight associated with the input binary's flowgraph, which is calculated during the static analysis.

For each matched procedure in the input binary, the similarity ratios are accumulated in accordance to the Dice coefficient. The asymmetric similarity is defined as:

$$S_x = \sum_i w_{ed_i} weight_{x_i}, \quad (4)$$

where i is the i^{th} matched flowgraph. The similarity ratio w_{ed} is always 1 for matched flowgraphs, but may be 0 if restrictions are relaxed to include non matched flowgraphs. As two weights are possible for each matched flowgraph i, two asymmetric similarities exist, S_a and S_b. These are evaluated in parallel, where S_a appropriates weights from the malware database, and S_b appropriates weights from the input binary. Intuitively S_a represents the size of the input binary as a subgraph of the malware, and S_b represents the size of the malware as a subgraph of the input binary. The asymmetric similarities are calculated for each particular malware in the database, so intuitively the calculations are repeated for each malware contained in the database.

The program similarity $S(a,b)$ is the final measure of similarity used for classification and is the product of the asymmetric similarities for a particular malware [20].

$$S(a,b) = S_a S_b \quad (5)$$

It may provide benefit to use alternate formula for calculating program similarity to reflect that viruses are identified as subsets of the set of signatures for the input binary. In this case, a single asymmetric similarity is the decisive factor. This technique has not been implemented by the prototype.

F. Improving Performance in Malware Classification

The above intuitive explanation is useful for understanding the process of classification, but for performance reasons the calculation of the asymmetric similarities does not occur individually for each malware in the database as described. To achieve a high level of performance, each procedure in the input program is processed and the asymmetric similarities incrementally built in each round for all malware matching flowgraphs. This is easily achieved piecewise, because the asymmetric similarity is merely the sum of the flowgraph weights. The two asymmetric similarities are maintained in a table associating them with each malware that has been partially matched. Each new round increments the asymmetric similarities by the appropriate weights when necessary.

Processing begins as explained previously by iterating over the control flow graphs and searching for matches in the malware database. Control flow graph signatures that are found numerously shared in multiple malware are evaluated last. The processing of these shared signatures is applied to each of the malware found using the earlier unique matches. This processing is performed by maintaining for each malware a smaller database of flowgraph signatures associated with only that particular malware. The processing is only applied if the evaluated weights may create a positive match in total similarity. Pseudo code to describe the algorithm is given in Fig. 3. The more control flow graphs present in a binary allow for more accurate classification. For programs with less than 10 flowgraphs, the binary is not eligible for classification by the prototype. In addition, for eligibility each flowgraph must have at least 5 basic blocks.

V. Analysis

The runtime complexity of malware classification is on average $O(N\log M)$ where M is the number of control flow graphs in the database, N is the number of control flow graphs in the input binary, and a dictionary search has a complexity of

```
S = 0.6
matches[name][Sa,Sb]   : output       : input initialized Sa=0, Sb=0
db                     : input        : malware database
in                     : input        : input binary
solutions              : global temporary

ProcessMatch(s: malware signature, similarityTogo)
{
        if (!seenBefore(s) && !solutions.seenBefore(s.malwareName)) {
                if (!matches[s.malwareName].find(s) and similarityTogo < S) {
                        // do nothing
                } else if (matches.find(s) &&
                        similarityTogo + matches[s.malwareName].Sa < S &&
                        similarityTogo + matches[s.malwareName].Sb < S)
                {
                        matches[s.malwareName].erase(s)
                } else {
                        matches[s.malwareName].Sa += weight_of_malware_cfg(s)
                        matches[s.malwareName].Sb += weight_of_input_cfg(s)
                }
        }
}

Classify(in: input binary, db: malware database)
{
        similarityTogo = 1.0
        foreach u in unique_cfg_matches(db, cfgs(in)) {
                solutions.reset()
                ProcessMatch(u, similarityTogo)
                similarityTogo -= weight_of_input_cfg(u)
        }
        dups = duplicate_cfg_matches(db, cfgs(in))
        foreach d in dups {
                if (1.0 - similarityTogo >= 1.0 - S)
                        break
                solutions.reset()
                foreach e in cfgs(d) {
                        ProcessMatch(malware_signature(d), similarityTogo)
                }
                similarityTogo -= weight_of_input_cfg(u)
                dups.erase(d)
        }
        foreach c in matches {
                tempSimilarityTogo = similarityTogo
                foreach d in dups {
                        solutions.reset()
                        foreach e in matching_cfgs_in_specific_db(db, d, c.malwareName)) {
                                ProcessMatch(malware_signature(e), tempSimilarityTogo)
                        }
                        tempSimilarityTogo -= weight_of_input_cfg(d)
                }
        }
        return matches
}
```

Figure 3. The malware classification algorithm.

$O(\log M)$. N is proportional to the input binary size and not more than several hundred in most cases. The worst case for classification is expected to have a runtime complexity of $O(N\log M + AN^2)$, where A is the number of highly similar malware to the input binary. In practice, this condition is unlikely to be significant as malware that is more than 95% similar to existing malware is not stored in the database, reducing the size of A. It is expected also that the average case is processing benign samples. Updates to the database can be made in $O(N\log M)$ time, using an $O(\log M)$ dictionary insertion time.

The runtime complexity investigated in a related and previously proposed system [20] employing approximate matching of control flow graphs is linear relative to the size of the malware database because it performs exhaustive calculations of pairwise similarity. This is worse than the logarithmic complexity of our system. Pairwise similarity in this system has a runtime complexity with an upper bound of

$O(ANB^2)$, where A and N are the number of control flow graphs in the compared pair, and B is the (maximum) size of a flowgraph signature. Exhaustive pairwise comparisons were found to be faster than searching a global flowgraph database which would have had an upper bound of $O(MNB^2)$. The use of metric trees reduces the lower bounds by factors of N and M respectively. The average case of searching using a metric tree is described below.

In the SMIT [19] malware classification system, the runtime complexity produces logarithmic pairwise comparisons relative to the size of the malware database. Pairwise similarity between call graphs using the Hungarian method is $O(N^3)$, where N is the sum of nodes in each graph. Evaluation demonstrated 20 pairwise comparisons per second on a modern desktop system. Metric trees and a multiresolution database reduce pairwise comparisons to a logarithmic growth. An average of 70% of the database size M=1000 was pruned when identifying the 10 nearest neighbours in a search utilising metric trees.

Our algorithm, has similar intentions and comparable results in identifying malware variants, but performs significantly more efficiently. Our system uses balanced binary search trees which have superior and predictable logarithmic growth compared to metric trees. VP-trees used in SMIT have a search complexity exponential in growth relative to the intrinsic dimensionality of its graph based objects [25]. Our system uses signatures with a dimensionality of 1. In a binary search using our system with M=1000, 99% of the database is pruned compared to the 70% using metric trees. Additionally, if perfect hashing is used for the dictionary search, complexity can be reduced to a constant time, but at the expense of more complex database updates. Our system also improves the performance relative to the query size by reducing the complexity from $O(N^3)$ to $O(N)$.

The runtime complexity of a typical multi-pattern string matching algorithm used in Antivirus systems, employing the Aho-Corasick algorithm [3] is independent of the size of the database and linear to the size of the input program and number of identified matches. The disadvantage of this approach, is that preprocessing of the database is required. Our system imposes more overhead by performing unpacking and static analysis, but detects significantly more malware, and is capable of real-time updates to the malware database. Additionally, in traditional Antivirus, false positives increase as the program sizes increase [21]. Our system is more resilient to false positives under these conditions because increased flowgraph complexity enables more precise signatures.

VI. Evaluation

A. Effectiveness

The prototype system identifies isomorphic control flow graphs. To evaluate the effectiveness of using isomorphisms or exact matching, we compared it to a previous system [20] employing approximate matching of control flow graphs. 40 malware variants from the Netsky, Klez, Roron and Frethem families of malware were classified. The Netsky, Klez and Roron malware samples were chosen to mimic a selection of the malware and evaluation metrics in previous research [4, 20].

TABLE I. Similarity matrices for malware families

	a	b	c	d	g	h
a		0.76	0.82	0.69	0.52	0.51
b	0.76		0.83	0.80	0.52	0.51
c	0.82	0.83		0.69	0.51	0.51
d	0.69	0.80	0.69		0.51	0.50
g	0.52	0.52	0.51	0.51		0.85
h	0.51	0.51	0.51	0.50	0.85	

klez

	aa	ac	f	j	p	t	x	y
aa		0.74	0.59	0.67	0.49	0.72	0.50	0.83
ac	0.74		0.69	0.78	0.40	0.55	0.37	0.63
f	0.59	0.69		0.88	0.44	0.61	0.41	0.70
j	0.67	0.78	0.88		0.49	0.69	0.46	0.79
p	0.49	0.40	0.44	0.49		0.68	0.85	0.58
t	0.72	0.55	0.61	0.69	0.68		0.63	0.86
x	0.50	0.37	0.41	0.46	0.85	0.63		0.54
y	0.83	0.63	0.70	0.79	0.58	0.86	0.54	

netsky

	ao	b	d	e	g	k	m	q	a
ao		0.44	0.28	0.27	0.28	0.55	0.44	0.44	0.47
b	0.44		0.27	0.27	0.27	0.51	1.00	1.00	0.58
d	0.28	0.27		0.48	0.56	0.27	0.27	0.27	0.27
e	0.27	0.27	0.48		0.59	0.27	0.27	0.27	0.27
g	0.28	0.27	0.56	0.59		0.27	0.27	0.27	0.27
k	0.55	0.51	0.27	0.27	0.27		0.51	0.51	0.75
m	0.44	1.00	0.27	0.27	0.27	0.51		1.00	0.58
q	0.44	1.00	0.27	0.27	0.27	0.51	1.00		0.58
a	0.47	0.58	0.27	0.27	0.27	0.75	0.58	0.58	

roron

The malware were obtained through a public database [26]. A number of the malware samples were packed. The prototype system automatically identifies and unpacks such malware as necessary. Each of the 40 malware sample was compared to every other sample. In approximate matching, 252 comparisons identified variants. The same evaluation was performed using our exact matching system, and 188 comparisons identified variants. No false positives were identified where a known malware had variants detected in another family. Approximate matching identifies more variants as expected while exact matching is demonstrated to be effective for a large selection of variants.

Table I evaluates the classification results in more detail. It is demonstrated that the classification systems finds high similarity in related malware families. Highlighted cells indicate a malware variant with a similarity equal to or exceeding 0.6. In normal operation, the system does not calculate the complete similarity between binaries which are not considered variants, however this performance feature was relaxed for this evaluation metric.

To evaluate the prototype system on a larger scale, 809 malware samples with unique MD5 hashes were collected between 29-04-2009 and 17-05-2009 from honeypots in the mwcollect Alliance network [27]. 754 samples were found to have at least one other sample in the set which was a variant. The strength and accuracy of this result is based on the evidence of our previous experiment where no false positives were identified. The summary of the results demonstrate the classification algorithm used by the prototype system is highly effective in detecting malware. The results also indicate that there is a high probability that new malware are variants of existing malware.

TABLE II. MALWARE PROCESSING TIME

Time(s)	Num. of Samples
0-1	299
1-2	401
2-3	46
3-4	30
4-5	32
5+	1

TABLE III. BENIGN SAMPLE PROCESSING TIME

Time(s)	Num. of Samples
0.0	0
0.1	139
0.2	80
0.3	42
0.4	28
0.5	10
0.6	10
0.7	3
0.8	6
0.9	5
1-2	17
2+	6

TABLE IV. SCALABILITY OF CLASSIFICATION

Database Size	1000	2000	4000	8000	16000	32000	64000
Time(ms)	< 1	< 1	< 1	< 1	< 1	< 1	< 1

TABLE V. SIMILARITY MATRIX FOR NONSIMILAR PROGRAMS

	cmd.exe	calc.exe	netsky.aa	klez.a	roron.ao
cmd.exe		0.00	0.00		0.00
calc.exe	0.00		0.00	0.00	0.00
netsky.aa	0.00	0.00		0.15	0.09
klez.a		0.00	0.15		0.13
roron.ao	0.00	0.00	0.09	0.13	

TABLE VI. FALSE POSITIVE EVALUATION

Similarity	Matches (approx.)	Matches (exact)
0.0	105497	97791
0.1	2268	1598
0.2	637	532
0.3	342	324
0.4	199	175
0.5	121	122
0.6	44	34
0.7	72	24
0.8	24	22
0.9	20	12
1.0	6	0

B. Efficiency

Table II evaluates the query response time of processing a sample in the set of 809 malware, including unpacking and classification time but excluding the loading time of the malware database. The evaluation was performed on a 2.4 GHz Quad Core Desktop PC with 4G of memory, running 32bit Windows Vista Home Premium. 86% of the malware were processed in under 1.3 seconds. The only malware that was not processed in under 5 seconds instead took nearly 14 seconds. This was because nearly 163 million instructions were emulated during unpacking. This is possibly the result of an anti-emulation loop. Manual inspection of the results also reveals some malware were not fully unpacked. The automated analysis is therefore likely generating signatures based on the packing tool, which becomes blacklisted by the system.

To evaluate the speed of classifying benign samples, 346 binaries in the Windows system directory were evaluated using the malware database created in the previous evaluation. The results are shown in Table III. The median time to perform classification was 0.25 seconds. The slowest sample classified required 5.12 seconds. Only 6 samples required more than 2 seconds. The results shown demonstrate efficient processing in the majority of benign and real malware samples, with speeds suitable for endhost adoption. It is much faster to process benign samples than malicious samples. Malicious samples are typically packed and the unpacking consumes the majority of processing time. The results clearly show this difference, and give more evidence that our system performs quickly in the average case.

To evaluate the scalability of the classification algorithm used in exact matching, a synthetic database was constructed. To simulate conditions likely in real samples, 10% of the control flow graphs were made common to all malware. The synthetic database contained up to a maximum of 64,000 malware, with each malware having 200 control flow graphs. The time to perform only classification, without unpacking, signature generation or any other processing is shown in Table IV. Less than a millisecond was required to complete classification for all evaluated database sizes. This shows an improvement to the Hungarian algorithm, multiresolution database, and metric trees used in SMIT [19], which classified 95% of samples in under 100 seconds, using a database of 100,000 real malware. Average time to classify in SMIT was 21 seconds, and the median classification time was 0.5 seconds. Our system in small scale testing unpacked, disassembled, and classified 99% of malware in under 5 seconds.

C. Accuracy

To evaluate the exact matching approach against false positives, the malware database created from the 809 samples was used when classifying the binaries in the windows system directory. No false positives were identified. The highest matching sample showed a similarity of 0.34. All other binaries had similarities below 0.25. This result clearly shows resilience against false positives.

Table V shows the similarity matrix for non similar programs and malware. As expected, these programs showed low similarity to each other.

To further evaluate and compare exact and approximate matching, Table VI shows a more thorough test for false positive generation by comparing each executable binary to every other binary in the Windows Vista system directory using both approximate and our exact flowgraph matching algorithm. The histogram groups binaries that share similarities in buckets grouped in intervals of 0.1. The results show there exists high similarities between some of the binaries, but for the majority of comparisons the similarity is less than 0.1. This seems a reasonable result as most binaries will be unrelated. Exact matching identifies fewer similarities than approximate matching, as expected. Exact matching also produces fewer comparisons due to the added requirement of each flowgraph

having at least 5 basic blocks, which resulted in some binaries being ineligible for analysis.

VII. LIMITATIONS

It is known that perfect disassembly and identification of control flow on the x86 instruction set architecture is undecidable in the general case of an obfuscated program [28]. In practice, an algorithmic solution exists because modern malware follows specific design constraints: 1) malware is written mostly in a high level language and compiled to form an executable 2) packing the executable uses known algorithms to obfuscate the program. By successfully unpacking malware, the unpacked images do not contain significant obfuscation. This makes automated analysis a practical endeavour. However, unpacking using application level emulation is liable to detection by malware and fails to fully emulate all novel samples [20]. Alternative and more effective real-time unpacking systems could be investigated, such as OmniUnpack [29]. More recently, packing using instruction set virtualization [30, 31] has shown to be resistant to automated unpacking, but this technique is not widely adopted by malware.

VIII. CONCLUSION

Malware variants can be detected by identifying similarity in control flow to existing malware. In this paper we proposed an algorithm to perform malware classification by estimating control flow graph isomorphisms. By accumulating isomorphic flowgraphs between the query program and known malware, similarities between programs can be constructed. These similarities allow malware variants to be detected. We implemented and evaluated this approach in a prototype system. The system employed an automated unpacking component to reveal the real contents of the malware. Our system was shown to detect a large proportion of malware when compared to an approximate flowgraph matching algorithm. The system was shown to be resilient to false positive identification. Additionally, the run-time complexity was shown to be logarithmic relative to the database size in the expected case. Finally, it was demonstrated that our system can effectively identify variants in larger samples of real malware, and the speed of classification warrants it suitable for use on the endhost.

REFERENCES

[1] K. Griffin, S. Schneider, X. Hu and T. Chiueh, "Automatic generation of string signatures for malware detection," Springer, 2009, pp. 101.

[2] J.O. Kephart and W.C. Arnold, "Automatic extraction of computer virus signatures," 1994, pp. 178-184.

[3] A.V. Aho and M.J. Corasick, "Efficient string matching: An aid to bibliographic search," Communications of the ACM, vol. 18, no. 6, 1975, pp. 340.

[4] E. Carrera and G. Erdélyi, "Digital genome mapping–advanced binary malware analysis," Virus Bulletin Conference, 2004, pp. 187-197.

[5] C. Kruegel, E. Kirda, D. Mutz, W. Robertson and G. Vigna, "Polymorphic worm detection using structural information of executables," Lecture notes in computer science, vol. 3858, 2006, pp. 207.

[6] P. Royal, M. Halpin, D. Dagon, R. Edmonds and W. Lee, "Polyunpack: Automating the hidden-code extraction of unpack-executing malware," Computer Security Applications Conference, 2006, pp. 289-300.

[7] Panda Research, "Mal(ware)formation statistics - panda research blog," 2007; http://research.pandasecurity.com/archive/Mal_2800_ware_2900_formation-statistics.aspx.

[8] A. Stepan, "Improving proactive detection of packed malware," Virus Bulletin Conference, 2006.

[9] J.Z. Kolter and M.A. Maloof, "Learning to detect malicious executables in the wild," International Conference on Knowledge Discovery and Data Mining, 2004, pp. 470-478.

[10] Y. Ye, D. Wang, T. Li and D. Ye, "Imds: Intelligent malware detection system," ACM, 2007, pp. 1047.

[11] M.E. Karim, A. Walenstein, A. Lakhotia and L. Parida, "Malware phylogeny generation using permutations of code," Journal in Computer Virology, vol. 1, no. 1, 2005, pp. 13-23.

[12] R. Perdisci, A. Lanzi and W. Lee, "Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables," Proceedings of the 2008 Annual Computer Security Applications Conference, IEEE Computer Society Washington, DC, USA, 2008, pp. 301-310.

[13] L. Boehne, "Pandora's bochs: Automatic unpacking of malware," University of Mannheim, 2008.

[14] M. Gheorghescu, "An automated virus classification system," Virus Bulletin Conference, 2005, pp. 294-300.

[15] T. Dullien and R. Rolles, "Graph-based comparison of executable objects (english version)," SSTIC, 2005.

[16] I. Briones and A. Gomez, "Graphs, entropy and grid computing: Automatic comparison of malware," Virus Bulletin Conference, 2008, pp. 1-12.

[17] Zynamics, "Vxclass," http://www.zynamics.com/vxclass.html.

[18] D. Gao, M.K. Reiter and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," Information and Communications Security, Springer, 2008, pp. 238–255.

[19] X. Hu, T. Chiueh and K.G. Shin, "Large-scale malware indexing using function-call graphs," Computer and Communications Security, ACM, pp. 611-620.

[20] S. Cesare and Y. Xiang, "Classification of malware using structured control flow," 8th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2010), 2010.

[21] G. Bonfante, M. Kaczmarek and J.Y. Marion, "Morphological detection of malware," International Conference on Malicious and Unwanted Software, IEEE, 2008, pp. 1-8.

[22] R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," IEEE Security and Privacy, vol. 5, no. 2, 2007, pp. 40.

[23] C. Kruegel, W. Robertson, F. Valeur and G. Vigna, "Static disassembly of obfuscated binaries," USENIX Security Symposium, 2004, pp. 18-18.

[24] G. Salton and M.J. McGill, Introduction to modern information retrieval, McGraw-Hill New York, 1983.

[25] N.Y. Peter, "Data structures and algorithms for nearest neighbor search in general metric spaces," Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, 1993, pp. 311-321.

[26] "Offensive computing," 2009; http://www.offensivecomputing.net.

[27] "Mwcollect alliance," 2009; http://alliance.mwcollect.org.

[28] R.N. Horspool and N. Marovac, "An approach to the problem of detranslation of computer programs," The Computer Journal, vol. 23, no. 3, 1979, pp. 223-229.

[29] L. Martignoni, M. Christodorescu and S. Jha, "Omniunpack: Fast, generic, and safe unpacking of malware," Proceedings of the Annual Computer Security Applications Conference (ACSAC), 2007, pp. 431-441.

[30] M. Sharif, A. Lanzi, J. Giffin and W. Lee, Rotalume: A tool for automatic reverse engineering of malware emulators, 2009.

[31] R. Rolles, "Unpacking virtualization obfuscators," USENIX Workshop on Offensive Technologies (WOOT), 2009.