

AN AUTOMATED VIRUS CLASSIFICATION SYSTEM

Marius Gheorghescu

Microsoft Corp, Security Business and Technology
Unit, One Microsoft Way, Redmond, 98052,
Washington, USA

Tel +1 425 707 3972 • Email
mariusg@microsoft.com

ABSTRACT

In recent years, significant interest has developed around automated malware classification methods and an industry-wide naming convention. However, in the anti-virus industry, virus naming is not a uniformly standardized process and only worsens with each new malware sample. Virus naming cannot be reliable unless the virus analyst can tell if a new sample is part of an existing family in a reasonable amount of time. Previous research in automatic classification has produced several interesting classification methods; however, to our knowledge, none of the methods can deal with an entire virus collection or produce meaningful results in a reasonable amount of time.

In this paper, we introduce an innovative classification system that uses an average desktop machine. The classification system compares new and unknown samples with all existing malware, and within a few minutes, returns matches for that sample based on evolutionary behaviour of existing malware. Compared to previous methods, our method is independent of the malware class and language. We describe three approximate matching algorithms and evaluate their run time and storage space requirements. We also discuss how these methods are applied in several malware-handling tasks including sample clustering, outbreak detection, automatic virus naming, and phylogeny tree.

1. INTRODUCTION

Until recently, automatic virus classification has largely been ignored because manual classification has been acceptable. However, the increasing number of viruses and virus variants and the need to reduce time-to-market necessitate anti-virus (AV) vendors to reduce the time spent in investigating the origins of samples. This situation leads AV vendors to resort to unsynchronized or randomly chosen virus names [1]. Most vendors have problems classifying samples internally and some have trouble keeping to a single naming standard in their product. Virus naming problems cause confusion not only in the research community, but also amongst users and product support groups. Attempts have been made to standardize the names of *in-the-wild* viruses but AV vendors generally cannot comply for various reasons (e.g. names are already propagated in internal structures, immediate gratification in changing the name is not obvious, influence of customer advisories and the media). However, the majority of viruses are *zoo* viruses and for these viruses, the classification accuracy remains at the discretion of each vendor.

The CARO naming convention [2, 3] initially proposed in 1991 established a set of syntactical rules and conventions for

virus names, but the naming process relies entirely on the etiquette and experience of the virus analysts. Unfortunately, only a subset of AV products has adopted the CARO convention and some have modified the convention for their own use. Another important and successful initiative is the VGrep project [4]. This project strives to build a cross-reference database for virus names given by various products. As of April 2005, the VGrep databases comprise roughly 250,000 name classes (rows) across 20 of the most common AV products (columns). With 250,000 name classes, one might initially conclude that 250,000 unique viruses have been determined to date. In actuality, the average number of different viruses per product is 55,000 (the sum of unique names in each row divided by the number of rows) and the remaining records are the result of detection and identification problems. The most astonishing proof that naming issues persist is the disagreement between different AV products in VGrep about virus naming.

To illustrate this discrepancy, we implemented a utility to count the number of entries where at least n products used the same name, ignoring case sensitivity. Table 1 contains the results from our utility. Not a single entry was reported as sharing the same name by all the products in the database, with only two old virus names ('Brasil' and 'Chaos.1241') common between 15 AV products. With our utility, we only measured n products in each row, but when considering n products in the database, the agreement rate is lower. The agreement rate is higher if the vendor-specific prefixes and suffixes are ignored, but we did not exclude suffixes because we did not believe it would produce significantly different results.

n	Equivalent names	Percentage
3	61,020	25.55%
5	29,300	12.12%
10	1,866	0.77%
15	2	0.0%

Table 1: Number of VGrep entries where at least n products used the same name.

We argue that classification and naming problems arise from the lack of a well-defined process for discovering malware relationships. In this paper, we investigate three similarity methods and demonstrate that building a real-time automated classification system is possible.

The classification methods we introduce in this paper rely on the observation that malware samples in the same family or derived from the same source code share a large number of basic blocks. The first classification method is based on the ubiquitous edit distance [5, 6] applied on basic block sequences and the use of a filtering algorithm. The second method improves the speed by taking a quantitative approach to matching and using an inverted index extended by a B-Tree structure. The third method, based on fixed size feature maps represented as Bloom filters [7], reduces the storage space significantly, but it also reduces accuracy.

2. PREVIOUS WORK

There are several initiatives in virus classification; however, to our knowledge, none of the techniques proposed meets the actual classification need – an autonomous and automated system that takes a new and unknown file and finds related samples in a reasonable amount of time. For arbitrary binaries,

[8] and [9] provide methods to find differences between two similar binaries known in advance. In the area of malware programs, Goldberg *et al.* [10] investigated the construction phylogeny tree for closely related samples, but no practical results were found. Wehner [11] provides a method to classify static malware using the normalized compression distance (NCD). The authors present interesting results, but the method is prone to false positives when the compression exploits the similarity between unrelated languages or files. It is also unclear whether the classification speed is suitable for a real-time system. Carrera and Erdélyi [12] introduced a function-level comparison method that relies on IDA [13] to extract the control flow graph. The classification results are promising, but the authors do not mention the time complexity of the system. Karim *et al.* [14] introduce a method to construct the phylogeny tree using maximal p-patterns; however, the results are not adequate to assess the performance and the accuracy of the algorithm.

3. PRELIMINARIES

In this section we describe the preprocessing stage, which is identical for the three methods we use. We intend to classify both static and parasitic malware and we do not make a distinction between malware that spreads in binary or source code form (i.e. script viruses). We do not focus on malware code that evolves from infection to infection (i.e. whole body metamorphic viruses) because these samples occur infrequently, and thus can be classified manually.

In general, malware coders will change small parts of the original source code or patch the binary to produce a new variant. Source code modifications in high-level languages are usually small and localized, but during compilation minor changes in the source code usually produce significant changes in the binary, and often can be a challenge to detect.

When the malware is static and self-contained (e.g. a Backdoor or Trojan), the malware author may leave the source code unchanged, but compress or obfuscate the binary to avoid detection. Even though reconstructing an exact copy of the original may not be possible in all cases, the reconstructed binary usually remains similar to the original in structure and functionality. In our tests, we will assume that samples are processed and reduced to a plain, unencrypted and unpacked form. The unpacking functionality is already available in most anti-virus engines.

For parasitic infectors, the isolation of the virus code is trivial, assuming the samples are replicated in the lab using easily identifiable host programs. In the case of encrypted or polymorphic viruses, the static area code needs to be extracted. This extraction can be achieved by emulating or tracing the sample in a controlled environment and using heuristic methods to detect decryption loops and encrypted areas. The number of polymorphic viruses in existence is significant, but the methods and layout employed are limited; therefore, the success rate of this process is very high.

Even though various abstractions are possible, we use the control flow graph (CFG) abstraction for classification. In this graph, vertices represent basic blocks and arcs represent execution flow. A basic block is a continuous sequence of instructions that contains no jumps or jump targets. The CFG is applicable to binaries but also to source code or script languages. Figure 1 shows a sequence of x86 assembly instructions with basic block markers. Figure 2 shows the

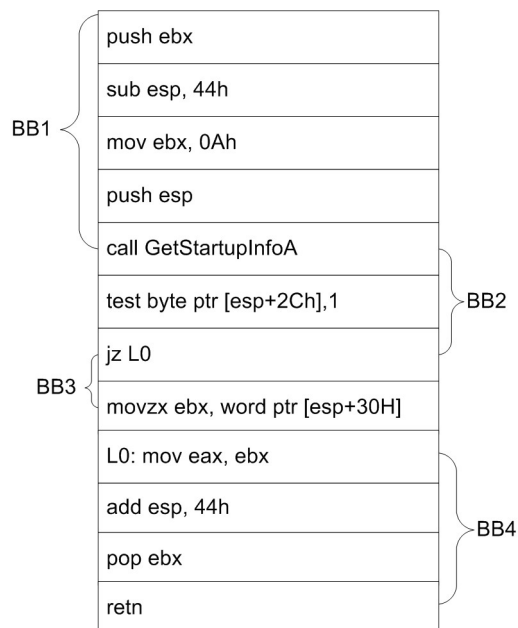


Figure 1: x86 assembly instructions with basic block markers.

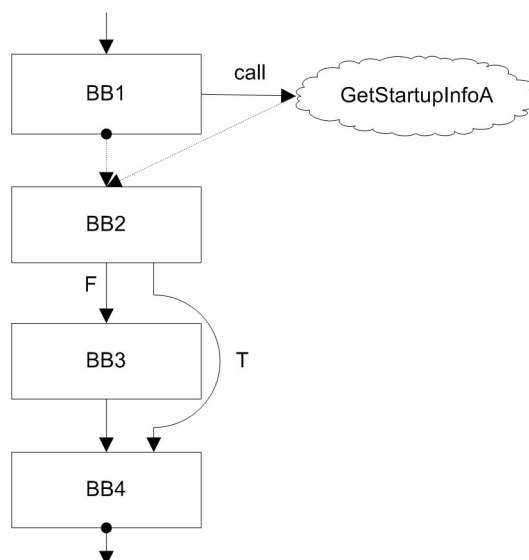


Figure 2: Control flow graph for the example in Figure 1.

corresponding control flow graph. GetStartupInfoA in this example becomes an external library function.

Basic blocks cannot be detected accurately in all cases because execution flow can change dynamically at runtime due to access restrictions, exceptions and external interrupts. Some viruses rely on specific processor features to trick emulation or debugging by changing the execution flow using non-flow instructions. However, in general, the CFG representation can be reconstructed easily from plain code [13].

By keeping a queue of unparsed targets, the CFG can be built in linear time in a single pass through the code. Initially the queue contains the program entry point(s). The main loop retrieves an unparsed target from the queue and does a fetch-decode loop until a stop, branching instruction, or another basic block is encountered. The exit targets are

attached as forward cross-references (arcs) to the current basic block, and are added to the queue if they do not point to parsed basic blocks. If one of the exit targets points inside an existing basic block, the target basic block needs to be split. The process continues until the queue of unparsed targets is empty.

In general, basic blocks extracted from compiled C code are small. In our tests, the average block size was around 12–14 bytes for 32-bit system executables or virus samples. This means that the unparsed target queue needs to be represented as a 1:1 index in order for the algorithm to execute in $O(n)$ time. In the case of source code samples, the fetch and decode loop needs to include the overhead of the lexical parser which is usually higher than the overhead of an instruction decoder.

To simplify the distance calculations we will ignore the arcs in the CFG and use only the nodes in the implicit file order. Although we believe arcs contain useful information, in general compilers group basic blocks together in the form of functions therefore the information loss is minimal.

In the case of low-level languages (i.e. assembly), the instructions are addressing data and code using fixed or relative offsets. This requires us to normalize such offsets so that they become location-insensitive. For the purpose of our experiments, we just replaced these offsets with a value zero.

Another preprocessing step is the static library code filtering. Malware programs compiled from high-level languages usually contain large portions of library code. The presence of the library code can decrease the classification accuracy significantly. The FLIRT technology [13] is very efficient at identifying library code. However, we used the same methods we used for classification to identify library code.

The similarity methods described in this paper require a comparison function between two basic blocks, or the ability to tell if two basic blocks are identical in functionality. One of the main advantages when using the basic block representation is the ability to change the content of a block without affecting other parts of the program. For example,

compilers rely on this property to perform local optimizations. This property also allows us to change the order of instructions inside a basic block during comparison. Figure 3 shows the system diagram, including the preprocessing stage.

4. DISTANCE ALGORITHMS

4.1. Edit distance

The string edit distance, also known as the Levenshtein distance, represents the minimum number of symbol insertions, deletions and replacements that transform one string into another. This distance is frequently used in genetics, spell checking, and file comparisons or patching (i.e. *diff*). The main drawback of the edit distance is that the classic *dynamic programming* implementation requires $O(mn)$ time and $m*n$ memory [15].

If distance is the only interest and the edit script (the sequence of operations) is not a concern, the memory requirement is $\min(m,n)$. Various improvements have been proposed to reduce the time complexity, but for long strings most improvements are insignificant [5]. Myers [16] introduced an algorithm that relies on bit-level parallelism to compute w cells in the matrix simultaneously (where w is the machine word size). A few modifications to the algorithm are required to compute the edit distance and to work with strings longer than the machine word. With desktop machines migrating to 64-bit architectures, the bit parallel implementation becomes an attractive choice for a small number of symbols.

A naïve approach to approximate matching on programs would be to compute the edit distance directly on the binary output of the preprocessor. We observed that by breaking the files into blocks of symbols of size k , we could reduce the total number of operations by k^2 and the memory requirements by k . For example, using $k=8$ would have a similar improvement on the speed as using the bit parallel implementation on a 64-bit desktop machine. Using a fixed size k would produce poor results because a single byte insertion or deletion would shift the content of all the subsequent blocks. Nevertheless, basic blocks are not globally affected by insertions or deletions, thus, we can use the list of basic blocks in the implicit file order as a string of symbols. Assuming the average basic block size for x86 binaries has an average of 12 bytes, we can reduce operations by 150 times compared to the naïve approach.

The number of operations required to search an entire collection is $O(mnk)$, where k is the number of samples. In this form, the search would take a few days when comparing a sample with 250,000 other samples, where each sample has 15,000 basic blocks. While the search operation can easily be performed in parallel on multiple machines, other optimizations can make classification on a single desktop machine possible.

The first optimization is to use a filter that reduces the number of samples for which the edit distance algorithm needs to be computed. The filtering process is a commonly used optimization step when computing the edit distance [5]. As a filter, we can use one of the algorithms discussed in section 4.2 and 4.3. Another optimization, suggested by Ukkonen [6], is to use a cut-off algorithm by only computing the cells of the *D.P.* matrix that are below the threshold. A fourth

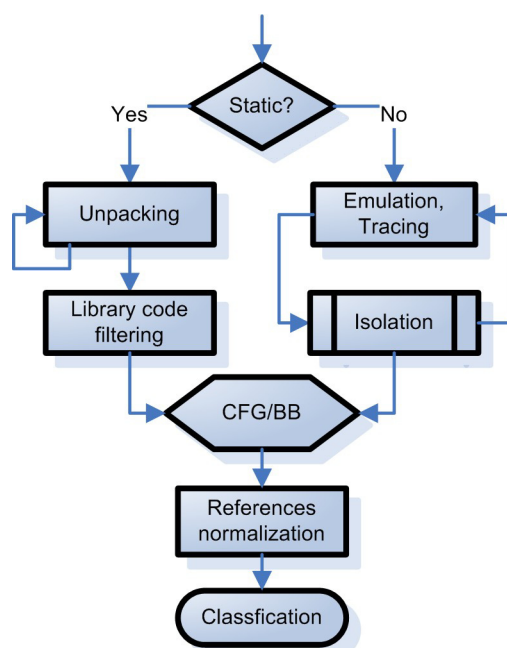


Figure 3: System diagram for the method.

optimization is to exploit the metric properties of the edit distance and index collection for range searching [17].

To simplify the edit distance computation, we assign an equal cost of one to the insert, delete and replace operations. When comparing two arbitrary samples, the base sample and modification may be identified by the number of insertions. However, this case generally does not occur and our intention is to obtain a symmetrical weight. We define the similarity ratio between two programs of length x and y respectively as:

$$w_{ed} = 1 - \frac{ed(x, y)}{\max(x, y)}$$

4.2. Inverted index

An alternative method for answering approximate queries is the inverted index, commonly used in word search engines. The inverted index consists of a set of symbols or words, where each symbol has an associated list of pointers to the origin of the symbol (i.e. document, paragraph). The inverted index is efficient in ranking word queries, easy to implement, and usually smaller than the total size of documents that are being indexed. The query process consists of reading the pointer list for each word in the query and updating the hit-count for each document.

A common extension of the query is to verify the order of the words. However, we will not use this extension because we want to make the query process as fast as possible. A proper structure for this number of keys is the B-Tree. B-Trees are commonly used in database engines because they can index a large number of keys and searching requires only $\log_m n$ disk operations where n is the number of keys in the tree and m is the order of the tree. The only drawback of this kind of tree is storage; depending on the order of the tree, the pointers and unused key slots can consume more than half of the total storage space.

The search algorithm for this layout is simple. First, each basic block in the source graph is searched in the B-Tree in $O(\log_m n)$ time. If the basic block is found, the inverted index is read and the sample hit-count is updated. The B-Tree insertion natively generates a small number of disk write operations in the form of block replacements, but the inverted index requires insertions in the list associated with each basic block. Because storing the list of pointers as a simple linked list would generate excessive disk read operations during search, we need to choose a proper allocation unit so that the wasted space is minimized. Empirically, the number of pointers per allocation unit needs to be fairly small – typically 2^5 to 2^7 entries – given that on average a new malware program shares about 50–60% of the code with the samples in the collection.

4.3. Bloom filters

Each of the two described methods has a drawback: edit distance is CPU-intensive and the inverted index is I/O bound. We now introduce an alternative method with efficient storage and a search time that is linearly dependent on the collection size. Bloom filters were introduced as a method for efficient membership queries with allowable errors [7]. Bloom filters are efficient not only in query time but also in storage space since they are fixed in size. Therefore, we can reduce the storage space by representing the basic blocks in a Bloom filter, ignoring the file order and the CFG topology. To

build the Bloom filter, we define a hash function $h(B)$ that has a basic block input that produces a value in the interval $[1, m]$ and a set of binary features $f_1, \dots, f_m$ representing the Bloom filter and defined as follows:

$$f_i = \begin{cases} 1, \exists h(b) = i, b \in CFG \\ 0, otherwise \end{cases}$$

Similarity queries can be answered by computing the hash function for each basic block in the source sample and verifying if the bit at the corresponding position in the target filter is set. When comparing multiple samples, we can simply build the Bloom filter for the source sample and compare it to the filters in the collection using bitwise operators. If, during comparison, we take into account the zero bits (representing basic blocks that are missing from the sample) we obtain the Hamming distance.

In practice, however, the Hamming distance makes a poor similarity measure for our purpose, and most samples are reported as being very close because results are skewed by the items that are missing from the sets. The large number of samples requires us to use a large filter; therefore, the Bloom filter of a sample can have most of the bits set to zero. A distance that considers only the items that are in the two sets is the symmetrical set difference, which is $(X \cap Y) \setminus (X \cap Y)$. We define the similarity ratio as follows:

$$d(x, y) = \frac{\sum x_i \wedge y_i}{\sum x_i \vee y_i}$$

The major benefit of using Bloom filters is that the database size is exactly $n \cdot m / 8$ bytes, where n is the number of samples in the collection and m is the Bloom filter size in bits. Because we used $m = 2^{16}$ in our experiments, we can expect the database size to be approximately 2^{31} bytes, which means that the entire database can be stored in RAM.

The distance between two samples is computed in $O(m)$ time and can only be further optimized by using bit parallelism and bit-counting optimizations. However, the collection lookup can be optimized by using a range search index. The set of filters is divided recursively into subsets that are closely related, and for each subset we compute the resulting Bloom filter using a bitwise OR operation between the members. The child nodes only need to store the bits that are set to '1' in the parent node. This representation reduces the space and computation time significantly, but the insertion becomes difficult because some nodes will need resizing.

5. APPLICATIONS

In addition to virus classification, these methods can be applied to various related processes such as sample clustering, submission queue filtering, and outbreak detection. This section overviews these applications.

In the case of large virus analysis teams, a virus analyst is more efficient in dealing with variants in the same family at once rather than variants spread across families, but variants of the same virus are infrequently assigned to the same analysts. Grouping similar samples in the submission queue or assigning virus analysts to a specific family rather than across families can decrease significantly the time required to analyse them. The methods we described allow arbitrary samples to be clustered together.

In general, new outbreaks are slight modifications of past outbreaks, but they can go unnoticed in the submission queue until the overall statistics change because of the large numbers of samples being submitted by customers. Having the ability to find duplicate submissions can increase the reaction time. In the past few years, a new type of attack has become common – machines compromised manually by rootkits or backdoors. This malware class does not generate outbreaks, and as a result, the number of submissions is very low and the statistical methods fail in prioritizing such malware. In these cases, a clustering-based system can prioritize such samples much faster.

Even if vendors succeed in using the same family name and synchronizing the variant markers for the first few variants in the family, the synchronization is often lost over time. This loss of control happens because each vendor assigns the variant marker in increasing lexical order (i.e. a letter in the CARO convention, number or version in some other cases) based on the order in which new variants are found. Variants are not synchronized using other factors because the sources and means by which variants are received are different (e.g. customer submissions, harvested samples, sample sharing between companies, virus writers submitting samples to their favourite products, etc.). The only accurate order for variants

is the order in which the variants were created. Unfortunately, discovering these details for most viruses is impossible (without heuristic methods). The only information that is easy to detect is the change between new variants and the original variant in the family. Using automatic classification, the distance from the base variant can be encoded as a variant sequence in the same way that discovery time is encoded. In the results section we show that our methods can detect the order in which the variants of a family are created.

Phylogeny trees [18] are a visualization method for evolutionary relationships. Phylogeny trees are commonly used in genetics, but they have also been suggested for computer virus classification [10, 12, 14]. Using a phylogeny tree representation allows a better understanding of the origins of a virus sample, compared to a list of ranked results. However, phylogeny trees do not preserve the relationships during incremental updates and are not fit for representing entire malware collections.

6. RESULTS AND RECOMMENDATIONS

This section contains the results obtained from applying the discussed methods. For all of the tests, we used a notebook computer with an *Intel Pentium M* 1.86GHz CPU, a 533MHz bus, 1.5GB of RAM, and a 60GB/7,200rpm HDD. Prototype

v0.21	1.000									
v0.26	0.686	1.000								
v0.30	0.588	0.697	1.000							
v0.33	0.583	0.694	0.886	1.000						
v0.37	0.523	0.624	0.784	0.840	1.000					
v0.50	0.482	0.569	0.662	0.683	0.756	1.000				
v0.51	0.481	0.568	0.660	0.682	0.753	0.993	1.000			
v0.73	0.366	0.429	0.463	0.472	0.495	0.579	0.580	1.000		
v0.84	0.287	0.321	0.347	0.357	0.366	0.375	0.376	0.461	1.000	
v1.00	0.287	0.321	0.347	0.357	0.366	0.375	0.376	0.461	1.000	1.000
ED	v0.21	v0.26	v0.30	v0.33	v0.37	V0.50	v0.51	v0.73	v0.84	V1.00

Table 2: Edit distance results on 10 variants of the Hacker Defender family.

v0.21	1.000									
v0.26	0.518	1.000								
v0.30	0.458	0.537	1.000							
v0.33	0.453	0.534	0.755	1.000						
v0.37	0.418	0.493	0.667	0.773	1.000					
v0.50	0.391	0.458	0.531	0.556	0.605	1.000				
v0.51	0.389	0.455	0.528	0.553	0.601	0.985	1.000			
v0.73	0.302	0.352	0.395	0.406	0.418	0.525	0.525	1.000		
v0.84	0.252	0.283	0.312	0.320	0.321	0.336	0.336	0.402	1.000	
v1.00	0.252	0.283	0.312	0.320	0.321	0.336	0.336	0.402	1.000	1.000
BF	v0.21	v0.26	v0.30	v0.33	v0.37	v0.50	v0.51	v0.73	v0.84	V1.00

Table 3: Quantitative classification results on 10 variants of the Hacker Defender family. BII and BF produced identical results.

code was written in C++ and compiled using *Microsoft Optimizing Compiler* version 13.10. In this section, BF denotes the Bloom filter method, BII denotes the inverted index method and ED denotes the edit distance method. We used 32-bit values when representing pointers in the structures of the databases. In addition, we represented the basic blocks as 32-bit CRCs to increase the speed of comparisons and reduce storage use. We used 2^{16} bits for the Bloom filter representation (8Kb per file) and a B-Tree of order 15.

In the first test, we evaluated the classification accuracy on 10 variants of the *Hacker Defender* family. We selected this family because the rootkit is popular, version data is available in the binaries, and automatic classification is a challenge to implement because the code originates from Delphi. There are also significant changes between versions: the binaries grow from 37,888 bytes in version 0.21 to 70,144 bytes in version 1.00. Table 2 shows the edit distance results. Table 3 shows the quantitative classification results in which BII and BF produced identical results in this test. Both methods produced similar results, and in both cases we found that the closer the version numbers were to each other, the closer they were to matching one another (the match decreases from top to bottom and from right to left). Figure 4 offers a graphical representation of the data in Table 2 and confirms that the creation order is identical to the version order.

Figure 5 describes the database size for the three methods. For this test, we used 4,000 randomly selected Win32 malware samples, and measured the database size every time 500 samples were inserted. The sample collection comprises 28,282,530 basic blocks, of which only 6,300,246 were unique. In the edit distance database, the storage space is linearly dependent on the size of samples. In the Bloom filter database, the number of samples is linearly dependent on the number of samples. We evaluated different order values for the B-Tree, but we did not find a significant decrease in storage use. Based on this chart, we can estimate the database size for a collection of 250,000 samples to be approximately 15GB for the B-Tree format, 7GB for the basic block list, and 2GB for the Bloom filter implementation. This means that the databases can be stored on an average desktop machine.

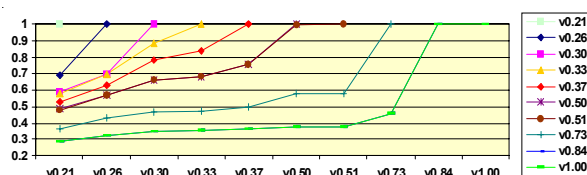


Figure 4: Evolutionary relationships on 10 variants of the *Hacker Defender* family, based on the edit distance classification.

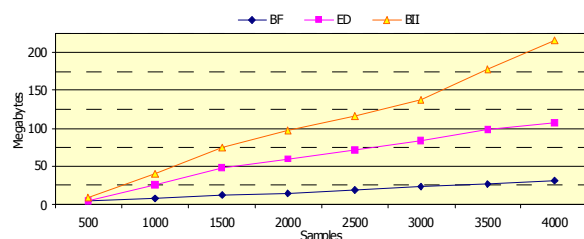


Figure 5: Database size for the three methods.

Database content		Query and insert time (seconds)			
# of samples	basic blocks	BF	ED	BII query	BII insert
1	5,500	0.000	0.135	0.009	0.005
5	36,000	0.001	1.568	0.010	0.011
10	21,000	0.001	2.373	0.011	0.012
20	97,000	0.002	4.345	0.012	0.012
50	310,000	0.004	13.962	0.017	0.012
100	792,000	0.008	39.206	0.043	0.012
500	3,215,000	0.045	–	0.073	0.013
1,000	6,785,000	0.090	–	0.109	0.014
4,000	28,287,000	0.364	–	1.201	0.032

Table 4: Query and insertion speed for the three methods.

The query and insertion speed for the three classification methods are given in Table 4. For this test, we populated the database with an increasing number of samples that were not in the database and measured the query time at each step, using *HackerDefender* v0.73. We also measured the insertion time for the inverted index, but not the ED and BF methods because the insertion time was constant. Furthermore, the results do not include the disk I/O and preprocessing overhead. The database size is expressed in number of samples and total number of basic blocks. As we expected, the edit distance computation quickly became unsuitable for comparing a large number of samples. Contrary to our initial expectations, the Bloom filter implementation was faster than the inverted index implementation.

In another test, we measured a query time of 100 seconds using a simulated BF database containing 250,000 samples. Using the BF method for the main query and applying the ED on the results is a practical implementation for the classification system that provides both speed and accuracy. Although the number of malware programs may exceed 250,000 samples in the future, in practice, the collection will be divided into several classes (i.e. DOS, Macro, Script, Win32, etc.). Consequently, using the appropriate class database instead of a single database for all classes can significantly reduce the effective search time.

7. CONCLUSIONS

In this paper we evaluated three malware classification methods from the perspective of run time, storage space, and classification accuracy. We found that it is possible to build an automated real-time system that can answer relationship queries and run on an average desktop machine. We found that the three distance algorithms return similar classification results. Compared to the previous research in this area, we have demonstrated that malware classification can be implemented in real time without affecting the overall reaction time. We hope that our contribution will lead to further exploration of implementing automated real-time malware classification that will lead eventually to an industry-wide standard in malware classification and naming.

ACKNOWLEDGEMENTS

I would like to thank Adrian Marinescu, Adrian E. Stepan,

Alexey Polyakov, June L. Park and Matthew Braverman for their helpful comments and suggestions.

REFERENCES

- [1] Raiu, C., 'A virus by any other name: virus naming practices', <http://securityfocus.com/>, June 2002.
- [2] *Virus Bulletin*, <http://www.virusbtn.com/old/archives/200301/caro.xml>, January 2003.
- [3] Bontchev, V., Skulason, F. and Solomon, A., 'CARO Virus Naming Convention', <http://www.caro.org/>, 1991.
- [4] Whalley, I., Gryaznov, D.O., VGreg, <http://www.virusbtn.com/resources/vgrep/index.xml>.
- [5] Navarro, G., 'A guided tour to approximate string matching', <http://www.dcc.uchile.cl/~gnavarro>, *ACM Computing Surveys*, 33(1):31–88, March 2001.
- [6] E. Ukkonen, 'Finding approximate patterns in strings', *Journal of Algorithms*, 6(1):132–137, 1985.
- [7] Bloom, B., 'Space/time trade-offs in hash coding with allowable errors', *Communications of the ACM*, 13(7):422–426, July 1970.
- [8] Sabin, T. 'Comparing binaries with graph isomorphisms', <http://razor.bindview.com/publish/papers/comparing-binaries.html>, April 2004.
- [9] Dullien, T., Rolles, R., 'Graph-based comparison of executable objects', <http://www.sabre-security.com/files/BinDiffSSTIC05.pdf>, 2005.
- [10] Goldberg, L.A., Goldberg, P.W., Phillips, C.A., Sorkin, G.B., 'Constructing computer virus phylogenies', *Journal of Algorithms*, 26(1):188–208, January 1998.
- [11] Wehner, S. 'Analyzing worms using compression', <http://homepages.cwi.nl/~wehner/worms/>, June 2004.
- [12] Carrera, E., Erdélyi, G., 'Digital genome mapping – advanced binary malware analysis', *Proc. Virus Bull. Int. Conf.*, 187–197, September 2004.
- [13] DataRescue, *IDA Pro 4.8*, <http://www.datarescue.com/>.
- [14] Karim, E., Walenstein, A., Lakhotia, A., Parida, L., 'Malware phylogeny using maximal p-patterns', *Proceedings of the EICAR 2005 Conference*, 167–174, April – May 2005.
- [15] Sellers, P.H., 'The theory and computations of evolutionary distances: Pattern recognition', *Journal of Algorithms*, 1:359–373, 1980.
- [16] Myers, G., 'A fast bit-vector algorithm for approximate string matching based on dynamic programming', *Journal of the ACM (JACM)*, 3:395–415, May 1998.
- [17] Navarro, G., 'Searching in metric spaces by spatial approximation', <http://www.dcc.uchile.cl/~gnavarro>, *SPIRE 1999*, 141–148, September 1999.
- [18] Baldauf, S., 'Phylogeny for the faint of heart: a tutorial', *TRENDS in Genetics*, 19:345–351, June 2003.