Exploiting Similarity Between Variants to Defeat Malware

“Vilo” Method for Comparing and Searching Binary Programs

Andrew Walenstein, Michael Venable,
Matthew Hayes, Christopher Thompson, and Arun Lakhotia
Software Research Laboratory
Center for Advanced Computer Studies
University of Louisiana at Lafayette
walenste@ieee.org

White Paper for BlackHat DC 2007

Abstract

Many malicious programs are just previously-seen programs that have had some minor changes made to them. A slightly different variant hardly qualifies as a stealth attack: being 99% the same as a known piece of malware should be a dead giveaway. This white paper describes a method for searching database of programs for a match. The methods are adapted from ordinary text search and analysis; the key to making them work is in selecting the right aspects of the programs to compare. The aspects compared are features called “n-perms” which are constructed from abstracted, disassembled code. Two studies show that these methods can be applied successfully to the problems of matching malware.

Contents

1 Introduction

Every day, scores of "new" malicious programs flood into the offices of anti-malware researchers worldwide. Most of these "new" ones, however, are really just recycled old code with a few changes made to them. What is needed is a way of matching the new, unknown programs against a database of old programs in such a way that the differences do not confound the search. What is needed is a kind of Google for code, but where the search terms are whole programs themselves.

This white paper shows how feature-matching techniques from text-based search can be adapted and used to create such a search interface. We had originally used these techniques to create *phylogenetic trees* of malicious program families, that is, trees that show derivation relationships between different related variants [1]. A core requirement for generating the phylogenetic trees in this method was a similarity measure between programs. The similarity score is based on just the operations from a disassembly of the programs in question. Once you have a similarity measure there is a straightforward procedure to create a search engine for programs: iterate through a database of programs and calculate the similarity between each program and the query program. Similarity becomes the match score. The question is: is such a search reasonable for program matching?

The data we have so far from case studies suggest it can be accurate, and should scale to practical sizes. More background and motivation is provided in Section 2, the feature-matching approach to program search is described in Section 3, and an evaluation based on artificially constructed data is described in Section 4.

2 Problem and Motivation

Approximate program matching can play an important role in malware defense because of (a) the huge numbers of minor variations of existing malware, and (b) the impact that good match techniques can have on malware analysis or detection.

2.1 Few Families, Many Variants

The majority of malicious software floating around are variations of already well-known malware. Consider the data from Microsoft's "Microsoft Security Intelligence Report: January – June 2006" [2]. According to their data, there were 97,924 variants of malware found within the first half of 2006. That is over 22 different varieties per hour. The report does not say precisely what Microsoft considers a variant, but clearly simple byte-level differences are not enough for them to label two files as distinct variants. For instance, of the 5,706 unique files that were classified as Win32/Rbot, only 3,320 distinct variants were recognized. The variants Microsoft list are not just trivially different files.

Clearly, these 97,924 distinct bundles of malevolence cannot *all* be completely different malware built lovingly from scratch. Instead, the vast majority are merely modifications of previous software—new versions of programs. According to Symantec [3] and Microsoft [2] typically only a few hundred *families* are typical in any half-year period.

And the Microsoft numbers paint a striking picture. Figure 2.1 shows a pie chart of the numbers of variants found. It breaks down the distribution into the 7 most common families, the 8-25 ranked families, and then the rest. The top 7 families account for more than 50% of all variants found. The top 25 families account for over 75%. Thus it is a solid bet that any new malicious program found in the wild is a variation of some previous program. The lion's share of work in handling the flood of new programs would be done if one could recognize even

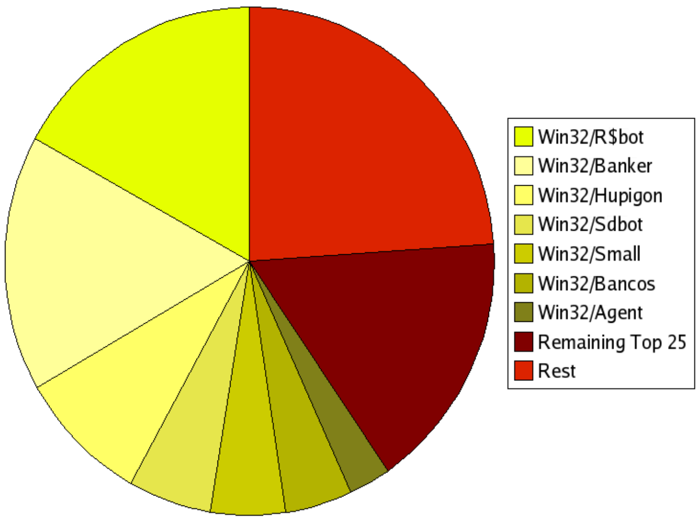


Figure 1: Relative numbers of distinct variants found by Microsoft, by family

only the topmost 25 families automatically.

There are two main factors driving this explosion of variety. The primary driver is that malware authors want to reuse their old programs. Obviously the main reason nowadays is that the malware author seeks to keep making money. An effective, mostly bug-free worm, for example, is costly to develop from scratch. Costly in time, costly in effort, maybe even costly in terms of dollars spent to buy an undisclosed vulnerability. That is an investment she would want to keep. However the problem they encounter is that malware detectors keep (eventually) catching their creations. Signatures are created and propagated. They do not seem to get the hint and give up. Instead, the solution they pursue, now more commonly, is to unleash scores of subtly different versions of their programs. Some of these—like any other software product that undergoes change—are different simply because they have been updated, upgraded, or fixed. However in other cases the changes are made to evade the new signatures.

A second cause for the sheer numbers of variants is the increasing rate of signature updates. In earlier times the rate was more leisurely for malware and defense evolution [4]. Anti-malware companies are now pressured and able to respond with new signatures with increasing frequency: in some cases several times per day.

2.2 The Role of Program Binary Comparison

A comprehensive way of counteracting the flood is required. Surely it must be considered a weakness in attack to have each variant so similar to a previously known variant. The situation seems ripe for a counter-attack. The anti-malware community has plenty of past samples. Often only minor changes are made between variants. Just how hard can it be to find matching samples and use these in defense? There would be at least two main classes of use for such search services: automated detection and in assisting malware analysts.

Automated Detection

First, there may be a possibility for automated defense. If a program comes in which is 99.5% the same as `Win32.Bagle.AG`, say, how likely is it to be a completely benign, non-malicious program? Currently a variety of heuristic means are used to detect variants, including matching suspicious or blacklisted behaviors (possibly through emulation), and so-called anomaly detection based on monitoring behavior. An alternative would be to perform whole-program matching and then filter any program matching too closely to a known malicious program. Such a detector would provide a preemptive defense against close family members. While this is a noted possibility, to limit scope the focus of this white paper is only on the search problem for triage and malware analysis.

Malware Triage And Analysis

Malicious variants fill the work queues of anti-malware vendors and other security analysts. Consider a perhaps prototypical anti-virus analyst in charge of making new signatures to catch the new variants. She would have to

1. Establish which family of malware the sample comes from. Google searches on extracted strings only helps so much.
2. Find the differences affecting the signature matching. The old signatures are not matching.
3. Find unique features to use for a new signature (but see the next point).
4. Look for similarities to other members of the family. Obviously one does not want a new signature for each and every variant.

There may be few organization-wide resources to consult in performing any of these actions. History is not working on the analyst's side if it cannot be brought to action at the moment when it is needed. The failure to recall related programs magnifies the degree to which the whole process relies on the deep knowledge of the analyst.

An effective search method would help. If one had the appropriate search that can find approximate matches, then a new approach may be possible:

1. A new, unknown sample arrives. Closely related programs—known already to be malicious—are retrieved automatically. The analyst need not have seen the family before.
2. Signatures associated with all of the related samples are retrieved. Past documentation and reports, if these are maintained, can be recalled in conjunction.
3. The similarity analysis is necessarily based on commonalities and differences. With the right analysis, these differences and commonalities can be reported to the analysts. Differences permit the analyst to quickly focus on how to fix existing signatures. Commonalities help the analyst discover more generic signatures.

Such a search system could help the defense infrastructure: it can reduce time to signature release, and frees the team to focus on the most important problems. These should improve detection rates.

2.3 Key Challenges for Malware Search

Attacking the variant flood problem in a principled matter requires being able to compare executables, i.e., to create a *similarity function* for executables. There are a whole host of issues to contend with in comparing binaries. For example, packing and encrypting is a problem that must be dealt with in some manner. Such things are, however, problems to contend with for essentially any malware analysis, and are beyond the scope of searching. Instead, the issues raised here are restricted to issues on creating a useful, scalable search system. Specifically, three key concerns are considered:

Change sensitivity.

A good program matcher would ensure that variations between programs do not throw off the search. What this means to us is that the similarity function used to compare executables must be such that it is insensitive to the types of variations that one wishes counteract. An example of such variation is the differences in branch targets due to reordering or due size changes or reordering. For instance, if a malware author adds a few statements to the beginning of her program, then all of the subsequent branch targets will move. Any suitable similarity function will not be sensitive to such changes: it has to somehow account for or ignore branch target changes. Simple reoderings of modules, procedures, or even statements should similarly be accounted for.

Many of the common differencing or similarity functions for strings are not suitable in this regard because they focus specifically on such orderings. Sequencing, after all, is the defining characteristic of a string. Examples of these techniques include longest common subsequence, Levenshtein or edit distances, and Hamming distances. Various such techniques have been used before in finding matching source code (e.g., Kim *et. al* [5]), but are not considered because of the insensitivity we are desiring.

Handling common code.

Many programs—malicious or not—will link in common code such as initialization sequences and standard startup functions. Moreover at the low level even unrelated code shares some commonalities such as standard function prologues and epilogues. The match method should account for these so that emphasis is not unduly placed on features that are relatively insignificant with respect to identifying programs.

Simplicity.

Efficiency is always an issue. It is desirable to avoid costly analysis on the programs, such as extracting control flow graphs, program slices, or semantic-level representations. Certainly various similarity measures can be defined over such higher-level representations, such as control flow [6, 7, 8, 9, 10]. Besides being potentially more costly and difficult to scale to thousands of program comparisons, the more complicated methods are that much more vulnerable to obfuscations [11]. It is important to keep exploring what could be done with the near minimum of extracted information.

3 Vilo: Program Search Methods

In this section we outline a method for search a database of programs for a match to a given program. For ease of reference we are calling it the "Vilo" approach. The approach is an adaptation of text retrieval matching using the so-called $tf \times idf$ term-vector query matching methods. These have been used for matching text documents to queries (see e.g., Moffat *et. al* [12]), and for the related task of detecting duplicate documents [13].

This section overviews the fundamental ideas of the approach with particular emphasis on how they can be applied to executable program comparison. It also introduces the specific feature extraction techniques that are employed to try to ensure the approach meets our key challenges identified above (sensitivity to change, handling common code, simplicity). These fundamental ideas can be summarized as follows:

1. Whole programs are compared, meaning the set of features that are compared are relatively comprehensive. This can be contrasted to traditional signature-based techniques which look for a highly focused but diagnostic feature, such as a unique byte sequence. The features we use, in particular, are so-called "n-grams" and "n-perms" extracted from abstracted disassemblies.
2. A vector model is used for comparison. Feature counts are converted into vectors that can be compared by measuring their cosine angle. This is fast, and is simple to calculate.

3. Feature weights are automatically calculated from a corpus of programs or program fragments. Typically this is the database of programs to match against. The weighting scheme addresses the identified concern of handling common code effectively.

3.1 Feature Comparison Approach

How can two objects be compared? One way is to examine their structure. Another is to define a set of features that they may share and count the commonalities and differences. So, for example, chairs can be compared on the basis of how many legs they have, and the degree to which they fold or swivel. This type of comparison inherently does not depend on semantics, structure, or order *apart* from what is required by the definition of the features. However if the comparison is to work well, one must have the right set of features to compare.

n-grams

Text-based methods frequently use what are called "n-grams". An n-gram is simply a sequence of n *characters* found in succession within some document. n is typically 2 or 3, but in malware analysis instances of 5 and 20 have been tried [14, 15]. Various different definitions of "characters" can be used: it could be letters, words, sentences, or paragraphs, and they could modified (e.g., made lower case) or filtered entirely (e.g., filtering spaces). Figure 2 shows an example using 2-grams on text, while mapping to lower case and filtering out spaces and punctuation. From the list of 2-grams the differences and commonalities can be tallied. The differences are $\{si, st\}$ and commonalities are $\{at, ca, ec, he, in, is, th, ti\}$. These can be counted to arrive at a similarity score. n-grams are simple to extract (use a sliding n-sized window), and easy to compare. However the number of features can be extremely high as n grows, and by their nature (character sequences) create some sensitivity to sequencing.

There are many ways to apply n-grams to programs. The most commonly reported is probably to use raw bytes (e.g., [14, 16, 17]) as characters. However extracted embedded strings, disassembly, and some combination of all of these could be used. We have tried all of these, and each appears to do different things well. The focus here is on applications to the disassembled code specifically, which is not as well studied. In particular, we apply it to abstracted assembly: only the operations are used as characters. An example is shown in Figure 3.

Using the disassembly is strategically important because (a) it is relatively difficult to change the code substantially, and (b) the operations pertain to the essential characteristics of the program's behavior. In addition, filtering out all but the operations reduces the sensitivity to multiple relatively unimportant differences, particularly those due to "noise" within the arguments to the operations (branch targets, etc.).

Once the counts of features are tallied they can be compared. This is done by representing the feature counts in a vector and then using vector cosine.

STRING	2-GRAMS
The cat is in.	`th he ec ca at ti is si in`
Is the cat in?	`is st th he ec ca at ti in`

Figure 2: 2-grams on ASCII text

```
55                     push  ebp
b8 11 00 00 00         mov   $0x11,eax
89 e5                  mov   esp,ebp
57                     push  edi
99                     cltd
56                     push  esi
c7 45 e4 11 00 00 00   mov   $0x11,0xffffffe4(ebp)
```

(a) disassembly

FEATURE TYPE	FEATURES EXTRACTED
2-grams	`push_mov, mov_mov, mov_push, push_cltd, ...`
2-perms	`push_mov, mov_mov, push_cltd`

(b) features extracted from operations (boxed column)

Figure 3: Feature extraction from abstracted disassembly

n-perms

It is possible to reduce the importance of sequence information a step further. This can be done by introducing what are called "n-perms" [1]. n-perms are exactly like n-grams except that ordering of the characters is not considered during matching. Therefore, under n-perms, the string the matches the string hte and matches the string eth. Statement reorderings tend not to have a noticeable effect when using n-perms. A potential disadvantage as compared to n-grams is that false matches are more likely for any given n. In practice, we have found that using larger values of n reduces false matches while still being able to match permuted variations [1]. They, along with n-grams, are suitable feature types for this type of comparison.

3.2 Weighting

Not all n-gram or n-perm features are going to be equally interesting. For example, certain operation sequences—such as function epilogues or startup code—are common to many programs, even unrelated ones. What is needed is an automatic way of adjusting the weights of these features. It is not desirable to manually specify these weights by, for example, requiring an expert to create a list of common operation sequences to filter out (i.e., a "stoplist").

A well-known solution is the so-called "$tf \times idf$ weighting scheme. The "idf" stands for "inverse document frequency," and indicates that feature is weighted by the inverse of how frequently it appears in documents.[1] In the case of text retrieval, this weighting encodes the heuristic logic that a match on the relatively rare word "constitution", say, will tend to be more meaningful than matches on the word "the." For example, given the feature sets for two viruses $v_1 = [3\ 4\ 2\ 1]$ and $v_2 = [4\ 5\ 1\ 0]$, using standard cosine similarity

$$sim(v_1, v_2) = \frac{v_1 \bullet v_2}{|v_1|\,|v_2|} = \frac{3 \times 4 + 1 \times 0 + 2 \times 5}{\sqrt{3^2 + 1^2 + 2^2}\,\sqrt{4^2 + 0^2 + 5^2}} = 0.918$$

If there are 10 programs in the archive and the number of programs the features appear in are [9 8 3 2], then weighted versions of the vectors are $w_1 = [3/9\ \ 4/8\ \ 2/3\ \ 1/2] = [.33\ .25\ .66\ .50]$ and $w_2 = [4/9\ \ 5/8\ \ 1/3\ \ 0/2] = [.44\ .63\ .33\ .00]$, and the similarity becomes $sim(w_1, w_2) = .795$. The

[1]See http://www.soi.city.ac.uk/ ser/idf.html.

lowered similarity score reflects the reduced importance of the features that do match. While this is a straightforward scaling, other functions of the document frequency are also used, such as the logarithm of the document frequency [12].

3.3 Search

Given a program similarity function there is a simple naive algorithm to search a database of existing programs for a match to a given unknown program U: iteratively calculate the pairwise similarity between U and every program in the database, and then sort the result in descending order of similarity. The result is a ranked result list. The naive algorithm is is in the order of $O(NM)$ where N is the number of programs in the database and M is the vector length.

4 Evaluation

Two studies were conducted to evaluate the effectiveness of the Vilo approach to searching malicious program databases—specifically, the use of n-perms over operation sequences while using $tf \times idf$ + cosine formulation of similarity.

4.1 Performance Evaluation

The first study examined whether the naive scheme is efficient enough for realistic use.

A search server was implemented in `C`. It reads a database of feature count vectors and calculates the feature weightings. It then listens for queries, which are other feature count vectors. It returns a ranked list of all matches found. There is an internal minimum threshold of .002: any match with less than that similarity score is not returned. The hardware used was a Dell Precision 650 with a single Intel Xeon (dual core) processor running at 2.80 GHz, with 512k L2 cache running Fedora Core 5 Linux (smp). It had 4GB dual-channel ECC DDR RAM.

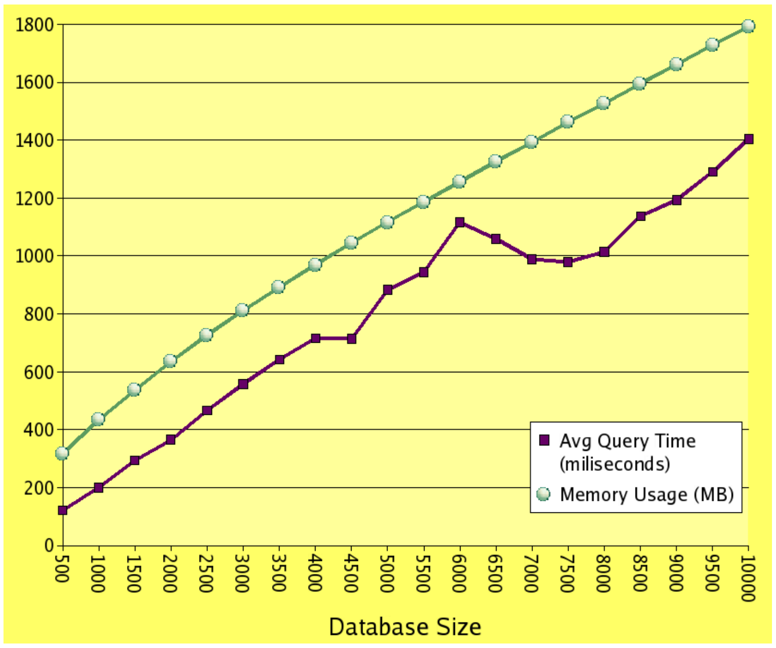
The disassembler used was `objdump`. `objdump` is a simple linear sweep disassembler. Like any disassembler, it can be fooled by obfuscations. Our apparatus allowed us to use `IDA Pro` also, but for performance and accuracy reasons we used `objdump` for these studies. 10-perms, over the operations in the disassembly, were used. The $tf \times idf$ scaling used, in particular, was $ln(\frac{N+1}{DF_t})$, where DF_t is the document frequency for a given feature t, N is the size of the database, and ln is the natural logarithm.

A set of sample databases was constructed. Each is loaded into the search server, and then searches were performed. The database size was varied from 500 to 10,000 samples. The amount of memory used by the search server (in MB) was collected, along with the average query time in terms of CPU milliseconds during the actual search process alone (communication overhead was not counted). Memory use was collected by using a Linux version of the search server and reading all the anonymous memory blocks reported by `pmap`. CPU time was averaged over 200 queries randomly selected from the database (the queried program was compared to all other programs in the database).

We did not have access to thousands of authentic, non-packed malicious samples. We generated feature count vectors by creating random seed feature sets to represent major malware families, and then programmatically mutating their feature sets to create variants. Program sizes of the generated feature count vectors were extrapolated from a convenience sample of 542 authentic worms collected from our departmental mail server, and the web. Mutation rate was fixed at .05, meaning 1 feature in 20 would be randomly modified to create a new variant.

For any given program size used, two entirely new random seeds were constructed and used to generate two equal-sized families.

Results and Discussion



In the figure, the x-axis indicates database size, i.e., the number of generated feature count vectors in the database. The red line with squares indicates average query times in milliseconds, and the green line with the circles indicates memory usage in megabytes.

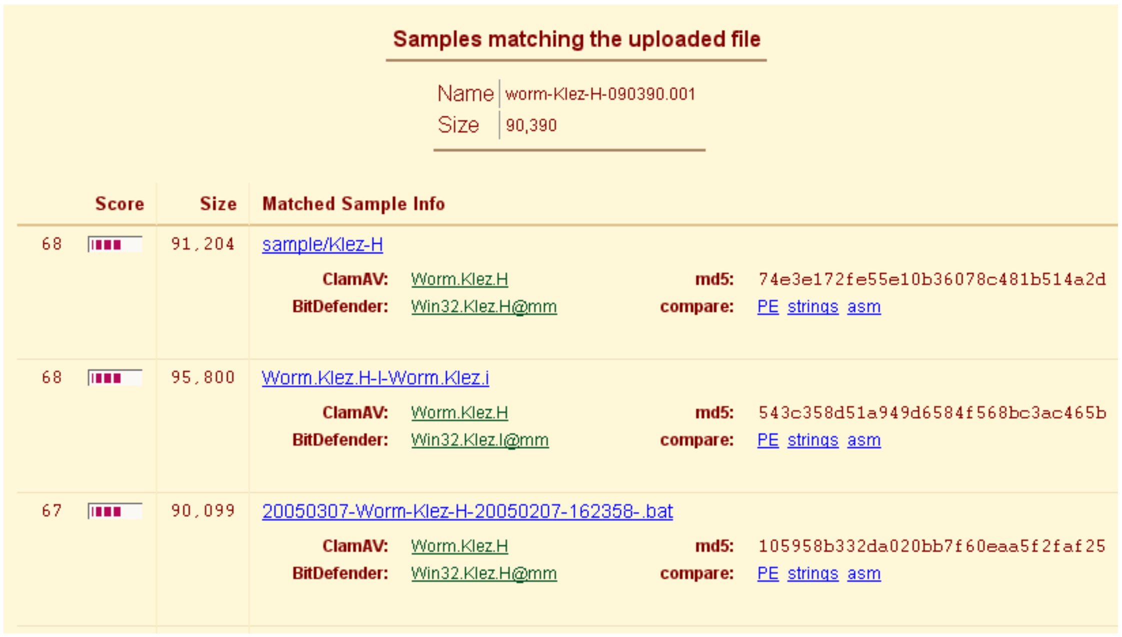
The expected use of the server is in cases where variants outnumber families substantially. Our constructed data set mirrored this situation. With relatively modest hardware the response time was suitable for interactive querying into the tens of thousands: less than 1.5 seconds per query, on average. Both memory use and CPU use are approximately linear. At this point we have no explanation for the noted dip in average CPU use at about the 6,000 mark.

4.2 Accuracy Evaluation

The second study examined more closely at whether the approach can yield accurate results. The search server was queried to return a ranked list of match results. A sample result list is shown in Figure 4.2. A threshold-based filter was added such that, given a threshold η, no result was returned to the user whose similarity score was less than η.

Two error classes are traditionally defined for search interfaces: *false positives* and *false negatives*. These are both defined in relation to the "goodness" of a match in a search result, where "goodness" is usually expressed in terms of "relevance" to the user's query. In our case, the user's query was a program and the relevant programs would be all those that are related by some type of derivation or commonality in code. By this definition, some samples from different families may be said to be related. For example, if some variants of `Win32.Sdbot` and `Win32.Gaobot` happen to use a common base code then they can be said to be related, and thus the `Win32.Sdbot` variant would be *relevant* if `Win32.Gaobot` is the query program.

The database was loaded with a set of feature count vectors from a collection of programs with *a priori* known relationships. The relationships are used to define relevance of the match.



Samples matching the uploaded file

Name	worm-Klez-H-090390.001
Size	90,390

Score		Size	Matched Sample Info			
68	▮▮▮▮	91,204	sample/Klez-H			
			ClamAV:	Worm.Klez.H	md5:	74e3e172fe55e10b36078c481b514a2d
			BitDefender:	Win32.Klez.H@mm	compare:	PE strings asm
68	▮▮▮▮	95,800	Worm.Klez.H-I-Worm.Klez.i			
			ClamAV:	Worm.Klez.H	md5:	543c358d51a949d6584f568bc3ac465b
			BitDefender:	Win32.Klez.I@mm	compare:	PE strings asm
67	▮▮▮▮	90,099	20050307-Worm-Klez-H-20050207-162358-.bat			
			ClamAV:	Worm.Klez.H	md5:	105958b332da020bb7f60eaa5f2faf25
			BitDefender:	Win32.Klez.H@mm	compare:	PE strings asm

Figure 4: Example ranked results list after uploading a KLez sample

Multiple queries are then run on randomly selected entries from the database. Classic measures of average *precision* and *recall* were collected. Precision measures the fraction of false positives within the result set; recall measures the fraction of false negatives. In most search systems there is a tradeoff between recall in precision. There is an expectation in typical threshold-parameterized systems that when the match threshold is raised the precision rises and recall falls. We tested two similarity thresholds: .100 and .002. The .100 was selected as an interesting point after some time playing around with different parameter values.

264 samples of 32-bit Windows malware was selected from the convenience sample of the first study. Using Bitdefender and ClamAV we tried to ensure that all were from the top-25 families identified by Microsoft for the first half of 2006 [2]. We used the familial names to determine relevancy; families known or suspected to be related were filtered out first, so as not to confuse the results. This filtering was appropriate since it still allows testing of the false positive error class while not affecting the false negative.

36 of the 264 were known to have been constructed by a construction kit. We obtained access to the code and created 202 variants by mutating the code and recompiling in a controlled environment. This generated a realistic set of family members related by a simulated evolution tree. In combination with the other authentic samples the database held 466 samples in total.

Results and Discussion

Threshold	Mean Precision	Mean Recall
.002	0.79	1.00
.100	1.00	1.00

The ideal is 100% recall and 100% precision. This was achieved for this small database using the .100 threshold: all the members of the same family were returned and no members of some other family were returned. The approach appears to cleanly partition the different families. When the threshold was lowered to .002 some non-family members were returned.

5 Conclusions

The feature vector approach to program similarity comparison yields a method for searching for previous malware that matches a new variant. The method meets several important criteria. It is simple and lightweight, requiring no analyses more complicated than disassembly. It has a method for automatically scaling the weights placed on features such that the less important features are given appropriately less emphasis. This addresses concerns of spurious matches due to common but unrelated code. The nature of the feature matching ignores order, and the use of n-perms can reduce the ordering sensitivity even further without necessarily compromising match accuracy.

The approach has potential for important applications in malware analysis. While the white paper focused primarily on traditional search applications, the tests suggest it also has potential for automated detection. It must be emphasized that while the results of the studies are encouraging, they are preliminary. We are presently performing more comprehensive evaluations on larger and more complicated data sets.[2]

Acknowledgments

Funding for this research was supplied by the Louisiana Governor's Information Technology Initiative.

[2] If someone has a data set they are able and willing to share, please contact the author.

References

[1] M. Karim, A. Walenstein, A. Lakhotia, and L. Parida, “Malware phylogeny generation using permutations of code,” *Journal in Computer Virology*, vol. 1, pp. 13–23, Nov. 2005.

[2] M. Braverman, J. Williams, and Z. Mador, “Microsoft security intelligence report: January–June 2006,” 2006. `http://microsoft.com/downloads/details.aspx?FamilyId=1C443104-5B3F-4C3A-868E-36A553FE2A02`.

[3] Symantec, “Symantec internet security threat report: Trends for January 06 – June 06.” `http://www.symantec.com/enterprise/threatreport/index.jsp`.

[4] P. Ször, *The Art of Computer Virus Research and Defense*. Symantec Press, 2005.

[5] M. Kim and D. Notkin, “Program element matching for multi-version program analyses,” in *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, (New York, NY, USA), pp. 58–64, ACM Press, 2006.

[6] H. Flake, “Structural comparison of executable objects,” in *Proceedings of DIMVA 2004: Detection of Intrusions and Malware and Vulnerability Assessment*, pp. 161–173, 2004.

[7] H. Flake, “More fun with graphs,” in *Proceedings of BlackHat Federal 2003*, 2003. `http://www.blackhat.com/presentations/bh-federal-03/bh-fed-03-halvar.pdf`.

[8] E. Carrera and G. Erdélyi, “Digital genome mapping – advanced binary malware analysis,” in *Proceedings of the 2004 Virus Bulletin Conference*, pp. 187–197, 2004.

[9] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, “Polymorphic worm detection using structural information of executables,” in *Proceedings of the 8th Symposium on Recent Advances in Intrusion Detection (RAID'2005)*, Lecture Notes in Computer Science, Springer-Verlag, 2005.

[10] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, “Semantics-aware malware detection,” in *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P'05)*, pp. 32–46, 2005. `http://doi.ieeecomputersociety.org/10.1109/SP.2005.20`.

[11] A. Lakhotia and P. K. Singh, “Challenges in getting formal with viruses,” *Virus Bulletin*, vol. 9, Sept. 2003.

[12] J. Zobel and A. Moffat, “Exploring the similarity space,” *SIGIR Forum*, vol. 32, no. 1, pp. 18–34, 1998. `http://doi.acm.org/10.1145/281250.281256`.

[13] T. C. Hoad and J. Zobel, “Methods for identifying versioned and plagiarized documents,” *Journal of the American Society for Information Science and Technology*, vol. 54, pp. 203–215, January 2003.

[14] J. O. Kephart and B. Arnold, “Automatic extraction of computer virus signatures,” in *4th Virus Bulletin International Conference* (R. Ford, ed.), (Abingdon, England, 1994), pp. 178–184, Virus Bulletin Ltd., 1994.

[15] L. A. Goldberg, P. W. Goldberg, C. A. Phillips, and G. B. Sorkin, “Constructing computer virus phylogenies,” *Journal of Algorithms*, vol. 26, no. 1, pp. 188–208, 1998. `http://www.cs.sandia.gov/~caphill/abstracts.html`.

[16] J. Z. Kolter and M. A. Maloof, “Learning to detect malicious executables in the wild,” in *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 470–478, 2004.

[17] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan, “N-gram-based detection of new malicious code,” in *Proceedings of the 28th Annual International Computer Software and Applications Conference*, IEEE CSP, 2003. `http://doi.ieeecomputersociety.org/10.1109/CMPSAC.2004.1342667`.