MetaAware: Identifying Metamorphic Malware*

Qinghua Zhang, Douglas S. Reeves
Cyber Defense Laboratory, Computer Science Department
North Carolina State University, Raleigh, NC 27695-8207
{qzhang2, reeves}@ncsu.edu

Abstract

Detection of malicious software (malware) by the use of static signatures is often criticized for being overly simplistic. Available methods of obfuscating code (so-called metamorphic malware) will invalidate the use of a fixed signature, without changing the harmful effects of the software. This paper presents a new approach for recognizing metamorphic malware. The method uses fully automated static analysis of executables to summarize and compare program semantics, based primarily on the pattern of library or system functions which are called.

The proposed method has been prototyped and evaluated using randomized benchmark programs, instances of known malware program variants, and utility software available in multiple releases. The results demonstrate three important capabilities of the proposed method: (a) it does well at identifying metamorphic variants of common malware; (b) it distinguishes easily between programs that are not related; and, (c) it can identify and detect program variations, or code reuse. Such variations can be due to insertion of malware (such as viruses) into the executable of a host program. We argue that this method of metamorphic code detection will be difficult for malware writers to bypass.

1 Introduction

Personal computers and other devices attached to the Internet must be protected from the enormous amount of malicious software which attempts to infiltrate such systems via the network. Examples of such malicious software, termed *malware*, include viruses, worms, spyware, and trojans. Malware instances have a variety of malicious purposes, effects, and penetration methods [21].

Signature based malware detectors have been popular and successful. An example of a signature would be a sequence of bytes that is completely characteristic of a specific malware instance. If a program upon inspection is found to contain such a byte sequence, it is suspected of being infected by malware. Deriving signatures of new attacks is a major function of the virus-checking and intrusion-detection industries.

However, signature based approaches are essentially syntactic and lack insight into the programs' semantics. Christodorescu and Jha [10] pointed out that such detection methods can be easily defeated by *metamorphism*, which uses code obfuscation techniques to transform the representation of programs. Metamorphic malware can obfuscate their entire code in a variety of ways, such as control flow transposition, substitution of equivalent instructions, variable renaming, etc. [9, 26]. This creates an arms race between the metamorphic malware writers (or obfuscation engines such as Mistfall, Win32/Simile, and RPME as pointed out in [26]), the signature writers, and the owners/administrators of the threatened computers or devices, which must be protected.

It is also quite common that new malware variants (or *mutants*) rapidly evolve from old malware, to which new functions have been added, or existing functionalities have been tweaked [15]. For example, the VX Heavens website [7] provides access to thousands of malware variants in a variety of different categories. For each malware variant, a signature must be identified, packaged, and downloaded to the base of users expecting protection from the new attack. The huge range of possible variants, and the speed with which they appear, makes this a less and less practical approach. Zmist is an advanced metamorphic virus that demonstrates a set of polymorphic and metamorphic code writing skills which include entry-point obscuring, randomly using an additional polymorphic decryptor, code permutation and code integration [26].

This paper offers a semantic characterization of programs, and a method of matching such characterizations, as a basis for malware detection that is resilient to many commonly-used obfuscation techniques. Generally the problem of determining whether a program will exhibit

*This work is supported by the National Science Foundation (NSF) under grant CNS-0627505.

a certain behavior is undecidable. Therefore, this paper presents an approximation approach to address the problem.

Essentially, this paper proposes a code pattern generation method which characterizes a code fragment's semantics, or functionality, based on system calls executed by the malware. It is based on static analysis of control and data flow of call traces, which include the calling instruction, as well as the instructions that prepare parameters for the system call. This paper also proposes a pattern matching method, which tells whether two patterns semantically resemble each other. Let the characterization of a code fragment be represented as `f: code -> pattern`, and the matching function as $\widehat{\mathtt{M}}$: `(pattern1, pattern2)` $->$ `r`, where `r` $\in [0, 1]$. When two code fragments `k` and `l` have similar or identical functionality, it is desired that `M(f(k), f(l))` be as close to 1 as possible; otherwise, it is desired that `M(f(k), f(l))` be close to 0. A weighted matching is used to compute the degree of similarity between two code fragments, based on their patterns.

The proposed approach differs from a previous research contribution [12] with similar assumptions and goals. That work proposes to use semantic templates of certain malicious behavior (such as the decryption loop in polymorphic malware) to detect malware. The templates are generated by studying the common behavior of a set of collected malware instances, and are generated manually. The method in this paper, in contrast, automatically generates a pattern that characterize a program's semantics, and uses this pattern to detect either obfuscated, or mutated, malware.

The proposed method has been implemented and evaluated on actual malware variants, widely-used benchmark programs that have been randomized, and different releases of of the GNU binutils programs [5]. The evaluation results demonstrate the proposed method can make a clear distinction between semantically equivalent or related programs, and those that are not. The measured similarity score of an original benchmark program and its randomized version in most cases achieves a value of .95 or greater. The measured similarity scores of different releases of the GNU binutil programs can achieve .75 or better. The measured similarity scores for different malware variants vary, but there is a very clear distinction between malware variants and non-variants. To apply the proposed approach in practice, a reasonable threshold can be set by the user to determine the sensitivity of malware detection.

The rest of this paper is organized as follows. Section 2 briefly reviews related work. Section 3 explains the proposed method. Section 4 presents the results of evaluating this method experimentally. Section 5 compares the proposed method with previous approaches, and points out the limitations of the proposed method. Section 6 concludes the paper.

2 Related Work

Malware detection has been an important research topic for quite some time [25, 20]. Some recent work has focused on the problem of metamorphic and polymorphic malware that uses code obfuscation techniques to bypass static signature based approaches. Christodorescu *et al*. [9] presented a unique view of malicious code detection as an obfuscation-deobfuscation game, and used control flow graph comparison to detect some simple obfuscation techniques often used by virus writers. The same authors [12] later proposed to use instruction semantics to formalize templates of certain malicious behavior, such as the occurrence of decryption loops in polymorphic (self-decrypting) malware. They then applied a template matching algorithm to detect malware. PolyUnpack [24] can identify unpack-executing malware based on a combination of static and dynamic analysis. This approach is based on the observation that sequences of packed or hidden code in a malware instance can be identified when its execution is checked against its static code model. Chouchane and Lakhotia [8] proposed using "engine signatures" to assist in detecting metamorphic malware. Basically, this technique evaluates collected forensic evidence from x86 code segments through a code scoring function. This score is a measure of how likely it is that the code has been generated by a known instruction-substituting metamorphic engine.

There has also been substantial work on the detection of obfuscated malicious code in network traffic. Zhang *et al*. [30] proposed a novel approach that can detect in network packets highly obfuscated polymorphic exploit code, based on static program analysis, and emulated instruction execution. This work is limited to the detection of malware that uses polymorphism, or self-decryption. Newsome *et al*. [23] proposed a signature generation system, which generates signatures that consist of multiple disjoint content substrings to match polymorphic or metamorphic worms. This approach is based on the assumption that there are invariant substrings that are present in all variants of a worm payload, and thus form a signature. Support for this assumption comes from exploits which contain common substrings that are crucial to successfully exploiting vulnerable servers. This approach needs to collect enough training flows to confidently produce valid, invariant signatures.

Kruegel *et al*. [18] proposed the use of structural analysis and comparison of binary code, based on its control flow. They used graph theory (coloring, isomorphism) to improve the results of their comparison. The difficulties of disassembling obfuscated code located at arbitrary locations inside network traffic are not fully addressed, however, and the method is computationally expensive. We also have found that metamorphism and obfuscation change the control flow in such a way that a straightforward comparison

on structural characteristics is likely to fail.

Other recent work [17, 11] has analyzed unique malware execution behavior to deal with the problems of polymorphism and metamorphism.

We now present a new method of detecting obfuscated variants, or mutants, of malware. Following the description, the method is evaluated experimentally, and compared with some of these recent approaches.

3 The New Method

In this section we present a new method of static analysis of executables. This method disassembles two executables, and then computes the degree of similarity between them. The essential characteristics used for this comparison are the system or library function calls made by the two programs. The method is intended to be used for recognition (and subsequent isolation) of metamorphic malware and malware variants.

3.1 Overview

Static program analysis is used for many purposes, such as security vulnerabilities checking [28, 14], and program behavior modeling for intrusion detection [25, 27]. Static program analysis needs to be done only once, and does not require run-time monitoring of program execution, which has substantial overhead. Proving that two programs (for instance, an instance of a virus and a suspected metamorphic variant) are functionally equivalent is an undecidable problem, unfortunately. The goal for static analysis of this paper is thus less than a full proof of functional equivalence.

Instead, we propose to to characterize a program in a way that can combine both structure and function. This characterization is referred to as the *pattern* of a code fragment. Ideally, the pattern should be markedly different for distinct malware, and the obfuscation used by metamorphic engines would not drastically change the pattern derived by static analysis. The challenge is to compute such patterns quickly, and to find a way to compare patterns that yields insight into the similarity between program functions. These patterns then can be used in a way that is similar to the way that signatures are used by conventional virus checkers. The use of patterns must be substantially more resistant to obfuscation than the use of fixed signatures, however.

When two programs are analyzed to produce patterns representing their function, these patterns can be compared to determine how similar the programs are. The process of comparing patterns is termed *pattern matching* in this paper. The output of pattern matching is a *similarity score* between 0 and 1, where a value of 0 is interpreted to mean the program functions are very different, and a value of 1 is interpreted to mean the program functions are extremely close or identical. To make a decision whether an unknown program is similar enough to a known malware to require that it be quarantined, this score must be greater than a user-defined threshold.

We propose that patterns are based primarily on the system calls or library executed by the malware. We propose to statically analyze the control and data flow of *call traces*, which are the instructions that prepare the parameters used by a system call, plus the corresponding call instruction. System call based modeling has been frequently used to characterize a program's behavior for intrusion detection purpose [25, 27]. It is a reasonable assumption that a compromised application cannot cause much harm unless it interacts with the underlying operating system [27].

As an illustration of this behavior, the Sapphire worm executes the following set of system calls [3]: `LoadLibrary`, `GetProcAddress`, `GetTickCount`, `socket`, `sendto`. Malware which did not make use of such system functions would likely be harder to write, and result in a much larger code size. The use of existing code obfuscation techniques or metamorphic program transformations does not in general remove such system calls from the malware.

The proposed pattern generation and matching methods are now described in detail.

3.2 Pattern Generation Based on Static Analysis

As mentioned above, system calls and library function calls are proposed to be used as the basis for generating a pattern that characterizes a target malware. Control flow and data flow analysis are used for this purpose.

To generate a pattern for code fragment p, p is disassembled first. There are a number of methods and tools designed to disassemble obfuscated binary code. The method of Kruegel *et al* [19] was adapted for this purpose.

Once the code is disassembled, the control flow is easily obtained through static analysis. The result of such an analysis is a set of basic blocks, and the transfers of control between those blocks. The call instructions that branch to system or library functions are then identified. Let I_i denote such a library call instruction in block i, and let the total number of library call instructions in the program be denoted as N.

The next task is to identify instructions that affect the parameters (values in memory or registers) used by the system functions when they are called by the program. While there can be many such parameters, the only such parameter used at present by the proposed method is the *target address* of the `call` instruction. Finding the instructions that affect the target address can be accomplished by data flow analysis.

The data flow analysis is initially given a single block i,

and includes the system call or library function call I_i contained in that block. In block i, the instructions affecting the parameters of I_i are determined. Essentially, they refer to the instructions with definitions reaching [1] this call. For each of these instructions, the blocks affecting their input operands are determined by data flow analysis. This process of backwards data flow analysis continues until either (a) the target block i is again reached (in which case a cycle has been discovered), or (b) there are no more instructions remaining for which backwards data flow analysis must be performed. The dependency or data flow relationship between instructions can then be represented as a graph, with a vertex for each instruction in the program, a directed edge from u to v if the instruction corresponding to vertex u affects the operand(s) of the instruction corresponding to vertex v, and the vertex representing I_i as the sink of the graph. A *maximal instruction trace* B in this graph is a path from a vertex having no predecessor in the graph, to the vertex representing I_i. The above process is performed for each system or library function call encountered in the disassembled code.

Following this data flow analysis, the instructions of each maximal instruction trace are processed to generate a subpattern for the trace. For this purpose, each instruction is first converted into an intermediate representation, based on the semantics of the instruction. This intermediate representation is convenient for processing, and allows functionally equivalent instructions to be represented in the same way. It also allows the method to be applied regardless of the instruction set architecture, although at present only the Intel x86 architecture [6] is targeted, due to its popularity.

The intermediate representation consists of the *operation type*, the *operands*, and the *operand addressing modes* (i.e. immediate data, register, or memory addressing). The operation types for the x86 architecture are classified into seven major categories (e.g., data transfer, arithmetic, logical, control transfer) and within each category multiple subcategories may be defined. For instance, the `loop` and `jcc` instructions both transfer control and therefore are assigned to the same operation type. Operands are classified as being of type source (read only) or destination (write only or read/write), and the addressing mode and associated register, if any, are recorded. The conversion to intermediate form allows many instructions that are functionally equivalent to be identified to a limited degree. For instance, using intermediate representations, the instructions `sub ecx, ecx` and `xor ecx, ecx` are identified as functionally equivalent to `mov ecx, 0`, and the instruction `push eax` is identified as being equivalent to `dec esp, 4` followed by `mov [esp], eax`.

After conversion to the intermediate representation, the instructions in each maximal trace are symbolically executed in a very limited way. Currently, this symbolic execution is simply the propagation of constant values. Suppose the first (earliest) instruction in a trace assigns a constant value `c` to a register or memory location, and this constant value can be propagated to the target system call I_i as the address to which flow of control will occur. This `call` instruction and the (constant) target address then form an *element* of a subpattern for a single maximal trace ending at I_i. Note that this symbolic execution is not sophisticated enough to recognize all target addresses unambiguously. For instance, some addresses may be computed from information that is not available until runtime. Therefore, when an instruction cannot be symbolically executed, the propagated constants are bound to it and recorded. All such instructions in their intermediate representation are recorded in order and form an element of the subpattern for a single maximal trace ending at I_i. An executable instruction in a maximal trace is not included in the element.

The *subpattern* for I_i, denoted U_i, is the set of all such elements for all maximal traces ending at I_i. The set of all such subpatterns U_i for all the system or function calls in code fragment p is called the pattern of p, and is denoted as P^p. The intuition behind this definition of a program pattern is that a malware program will normally make use of some well-defined system services, whose addresses must be found (so that they can be accessed) in a well defined way. Attempts to obfuscate the program function, without changing the set of system or library function calls, can still leave this behavior visible to inspection. Even obfuscation of the target of a system call may leave the true target exposed as one possibility. Since the proposed method uses all possible traces, this true target will remain part of the pattern of the code fragment. The use of both symbolic execution and control flow analysis for disassembly will also overcome many known methods of obfuscation, as will be shown in section 4.

Figure 1 shows an example of the the patterns generated for code fragments of the Sapphire worm, and for a metamorphic variant of this worm. For purposes of illustration, each sub-pattern is presented in Intel x86 assembly language form, and is a result of data flow analysis and symbolic execution. For instance, subpattern 1 of pattern PA results from symbolic execution of the instruction trace `mov esi, [0x42AE1018]` || `call [esi]`, in which the second instruction operand depends on the first instruction. Subpattern 3 of pattern PA has two elements, which result from two traces whose target is the same library function call. In this example, there happens to be multiple subpatterns which are identical. This is because some of the library functions are called in multiple places, with different parameters.

The next section explains a method of pattern matching

[1] Please refer to a textbook on compiler theory [22] for the explanation of reaching definition in data flow analysis.

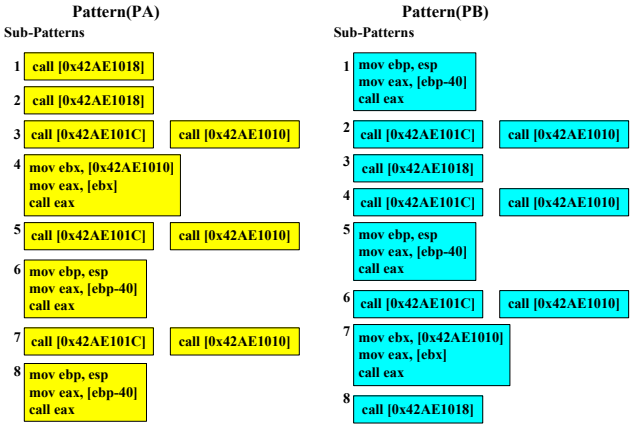

Figure 1. The patterns of code fragments of Sapphire worm and its metamorphic version.

to compute the similarity between two binaries. The input to this process is the patterns derived from the binaries in the way just described. The pattern matching algorithm is intended to overcome the differences between two variants of the same malware.

3.3 Pattern Matching

The purpose of pattern matching is to determine if two code fragments are similar enough to exhibit functional equivalency. The proposed method does not produce a formal proof of equivalence. Not only is that undecidable, but malware variants may in fact compute somewhat different results. Rather, we consider similarity in system or function call behavior to be strong evidence that programs have a similar purpose. The two requirements for defining patterns and the resulting pattern matching algorithm are:

1. The pattern derived from one malware program should be very different from patterns derived from other programs, whether benign, or malware of another type.

2. Patterns derived from metamorphic variants of a single malware program should be very similar.

The matching algorithm is defined as follows. Two code fragments k and l are given, where k may be, for instance, a known instance of malware. The pattern for k has been computed and is represented as $P^k = \{U_1^k, ..., U_{N_k}^k\}$. The pattern for l has been computed and is represented as $P^l = \{U_1^l, ..., U_{N_l}^l\}$.

Let similarity scores be real values between 0 (minimum similarity) and 1 (maximum similarity). Suppose similarity scores between all pairs of subpatterns, where one subpattern is taken from P^k and one subpattern is taken from P^l, have been computed.

A *pattern matching* of k and l is a one-to-one assignment from the set of subpatterns of k to the set of subpatterns of l. A maximum matching is one that includes all of the subpatterns of k, and/or all of the subpatterns of l. A maximum weighted matching is one that maximizes the sum of the similarity scores of the pairs of subpatterns that are matched. The value or score produced by a maximum weighted matching W is equal to the mean of the similarity scores of pairs of subpatterns that are present in that matching:

$$M(P^k, P^l) = \frac{\sum_{\langle U_i^k, U_j^l \rangle \in W} score(U_i^k, U_j^l)}{max(N_k, N_l)} \quad (1)$$

A maximum weighted matching is an optimistic approach to computing the similarity between two code fragments. The process of deriving and matching patterns should not be greatly affected by small errors in disassembly and data flow analysis, or by current program obfuscation techniques. These claims are evaluated in section 4.

Pattern matching is performed after similarity scores are computed for all pairs of sub-patterns. For each such pair of sub-patterns, the similarity score of all pairs of elements is computed, where one element is taken from the first subpattern, and the other element is taken from the second subpattern. From this, a maximum weighted matching of the elements of the two sub-patterns is computed, in the same way as mentioned before. The similarity score of this pair of sub-patterns is then the mean of the similarity scores of pairs of elements that are matched.

Finally, computing the similarity of two elements involves comparison of the instructions or instruction sequences (still in their intermediate form) in the two elements. This step finds a maximum weighted mapping between the instructions in the two elements. To do this, it is required to compute the similarity between any two instructions, using as input their intermediate forms. The computation is only an estimate of the similarity between instructions. Therefore, a heuristic method is used. This method first computes the similarity between operation types. As an example, `add` and `subtract` operations are deemed to be similar, while `add` and `call` are not. The comparison of operands checks for each operand pair whether the addressing mode, and register or memory addresses or immediate operands (when they can be determined) are the same, and scores them based on closeness. Closeness of operands is weighted more heavily than closeness of operation types when computing a final similarity score for two instructions. This computation is designed to be accurate enough to capture most obfuscations used in practice.

Figure 2 shows an example of the maximum weighted matching process for the two patterns shown in Figure 1.

	PB							
	0.1	0.2	**1**	0.2	0.1	0.2	0.1	1
	0.1	0.2	1	0.2	0.1	0.2	0.1	**1**
	0.05	**1**	0.2	1	0.05	1	0.05	0.2
PA	0.73	0.05	0.1	0.05	0.73	0.05	**1**	0.1
	0.05	1	0.2	**1**	0.05	1	0.05	0.2
	1	0.05	0.1	0.05	1	0.05	0.73	0.1
	0.05	1	0.2	1	0.05	**1**	0.05	0.2
	1	0.05	0.1	0.05	**1**	0.05	0.73	0.1

Figure 2. The maximum weighted pattern matching of code fragments of the Sapphire Worm and a metamorphic variant, whose patterns are shown in Figure 1. Each Cell is a similarity score of two subpatterns, one from pattern PA, and one from pattern PB. The marked cells show the maximum weighted matching. The score of pattern matching $M(PA, PB)$=8/8=1.

A software prototype of the proposed method has been implemented, based on the ideas described above. The Hungarian algorithm [29] is a well known method for solving weighted matching problems and was used in the implementation. The complexity of this algorithm is approximately $O(\max^2(N_k,N_l))$. Although the Hungarian algorithm has a polynomial running time, this could still be undesirably slow. For instance, a large program whose code size is measured in MB can easily produce thousands of subpatterns. Therefore, when the number of subpatterns exceeds a threshold, an approximate version of maximum weighted maching is used. In the next section, the preliminary results from testing of this software are described.

4 Evaluation

The proposed method computes the similarity between two binary executables, based on the characteristics described above. If one executable is derived from another (i.e., is a variant or version of another), the computed similarity should be very high. Otherwise, the computed similarity should be low, with a large gap allowing these two cases to be easily distinguished.

The proposed method has been fully implemented. This implementation can analyze executables for both the Linux and Windows operating systems, compiled for the Intel x86 instruction set architecture.

Three sets of inputs were used to test this hypothesis experimentally. The first set of inputs consisted of benchmark programs (compiled for Linux) that were processed using a tool for fine-grained randomization of commodity software [16]. The second set of inputs consisted of variants of known Windows viruses, downloaded from the VX Heavens [7] website. The third set of inputs consisted of various releases of the GNU binutils programs, compiled for the Linux platform. For each set, the similarities of known variants or versions were computed, and when it made sense to do so, the similarities of unrelated programs (neither derived from the other) were computed as well. The experiments and the results are described in more detail below.

		Matching	Code	Running Time(s)		
		Score	Size (K)	Disass.	Pattern Gen.	Matching
SPEC	twolf	98.48%	164.70	9.24	4.44	0.34
CPU2000	mcf	97.81%	7.86	0.015	0	0.01
	gcc	99.39%	1158.36	415.6	210.41	56.52
	bzip2	99.23%	29.18	0.09	0.05	0.01
	vortex	99.77%	399.82	44.75	23.17	4.05
	crafty	99.90%	173.75	7.85	3.57	5.28
	perlbmk	95.74%	483.12	123.8	66.31	2.6
	parser	99.07%	104.29	5.29	2.71	0.22
	gzip	76.87%	31.43	0.11	0.06	0.01
	vpr	79.94%	100.54	1.79	0.98	0.26
Apache	httpd	99.22%	300.69	42.46	20.48	323.06
Misc	ghttpd	99.42%	7.60	0.02	0.01	0

Figure 3. Pattern matching between randomized and original programs. Programs are from SPEC CPU2000 benchmark, the Apache web server httpd, and the GazTek web server ghttpd. Code size refers only to the code segment size, not the size of the entire executable program.

We were not able to use any existing tools designed to produce an obfuscated version of an arbitrary executable program that contain all common obfuscation techniques. Previous work [12] manually generated the obfuscated test cases with simple obfuscation techniques. The work [10] generated obfuscated test cases on programs written in visual basic language to test the resilience of commercial antivirus software to metamorphic malware.

4.1 Randomized Executables

To test resilience to obfuscation, a set of programs was randomized using the ASLP tool [16]. This tool uses binary rewriting to rearrange the static code and data segments of an executable file in a random way. It performs fine-grained permutation of the procedure bodies in a code segment, and of data structures in the data segment. This randomization technique can invalidate the use of static signatures for recognition of malicious code. This experiment was performed on a machine running Fedora Core 1, with a Pentium 4 CPU of 2.26GHz, and 512M of RAM.

ASLP was applied to programs taken from two well-known benchmark suites: the SPEC CPU2000 programs [1] and two web server programs (the Apache web server httpd [2] and and the GazTek web server ghttpd [4]). The similarity between the randomized and original versions of each program was then computed using the proposed method. The results are shown in figure 3.

Of these 12 test cases, 10 have similarity scores above 95%. This demonstrates that program changes due to randomization do not affect the ability of the proposed method to recognize the similarity in function between normal and randomized versions. The proposed method of analysis, using system calls as a point of reference, is robust to such changes in program structure.

Therefore, similarity scores for the two programs' randomized and normal versions are still high enough to conclude that they are the same program.

4.2 Variant Detection Evaluation

Virus and malware writers have manually created many variations of common exploits, in an attempt to evade virus-checking tools. Hundreds of such examples can be found at popular hacking web sites. Such examples make use of a wide range of obfuscation techniques.

One such website is VX Heavens [7], which identifies programs that are variants of the same malware. More than 200 pairs of malware mutants were downloaded from this website. These program instances were selected from multiple malware categories. Among these instances, 36.6% were worms, 18.3% were viruses, 20.8% were backdoor programs, and 16.3% were trojan programs. The remainder of the programs included flooders and exploits. The tested malware programs had sizes ranging from 8K to 1M bytes. These malware programs and their mutants employ multiple commonly used obfuscation techniques. For instance, the simplest obfuscation technique used is register renaming. A more complicated obfuscation technique is using functionally equivalent instruction substitution. The code addresses of the mutants can be very different. The relative offset in corresponding function `call` instructions are frequently adjusted. Sometimes, even within a single program, the same library functions may be imported multiple times at different addresses, although at runtime, these may be reloaded to the same address. Recognizing that two `call` instructions refer to the same target address requires data flow analysis techniques. A much more common case is that new functions or behaviors are added or revised in mutants. There is another obfuscation that is not considered here but is often encountered when downloading the malware programs. A none trivial number of malware programs use encryption. The limitation of the proposed approach will be discussed in the next section.

In the first test, similarity scores were computed for pairs of executables that represented the same attack. A histogram of the resulting similarity scores is shown in Figure 4. The great majority of pairs are easily recognized as variants of the same function. For example, over 90% of the pairs have a similarity score of .7 or greater. These mutants represent the state of the art in mutation and obfuscation of malware, and thus are a worthwhile test case for any program attempting to recognize metamorphic variants in an automated way. The results indicate that comparison with a previous version of malware will, with high probability, identify a new version of the malware.

It is also important to measure the similarities computed between programs which are *not* variants of the same malware. In the second test, the malware programs were randomly paired with each other, excluding all instances that were identified on VX Heavens as being variants of the same malware. Similarities for the resulting 200 pairs were then computed using the proposed method. Figure 5 shows the results. The computed similarities are very low, with over 90% having a similarity score of .1 or less. Approximately 1% have a similarity score of .7 or greater. It may be that malware even in different families are derived from a common code base, explaining these results. Further investigation is required.

For these programs, a similarity score of .7 would be an optimal threshold for concluding whether two programs, one of which is known to be malicious, are functionally equivalent. While this threshold does not perfectly distinguish malware variants from non-variants, keep in mind that this is a tough test case: identifying hand-crafted metamorphic malware, and distinguishing it from other malware, rather than distinguishing it from non-malicious code.

4.3 Version Difference Evaluation

The third experiment tested how well the proposed method recognized variations between different versions or releases of a program. Such versions are not intentionally obfuscated, but represent another case of software that is derived from a previous version of a program. They are therefore a useful test of the proposed method.

Releases 2.10 through 2.17 of the GNU project "binutils" binary tools [5] were used for this purpose. These tools are used for compiling, linking, and debugging programs. They make use of several common libraries for low level, shared functions. Different releases will represent varying degrees of modification to the original program code.

Figure 6 shows the result of computing the similarity between consecutive releases of each program, using the proposed method. For the great majority of cases, the computed simularity was greater than .7. The cases where this was *not* true are instructive to examine (see Figure 7. Between release 2.10 and 2.11, code sizes of the utilities increased by approximately 50%, indicating a major revision, and the computed similarity scores were correspondingly lower (around .5). Figures 8 and 9 compare both release 2.10 and release 2.11 to release 2.17. It is clear that release 2.11 is much closer (in size and similarity) to 2.17 than release 2.10 is to 2.17. Also, between release 2.13 and 2.14, the size of `c++filt` grew 10-fold, indicating essentially a replacement by a new program; the computed similarity in this case was close to 0.

These results indicate that the proposed method is effective at computing the degree of similarity between programs in a way that is meaningful, and that is not sensitive to modifications that preserve a program's function.

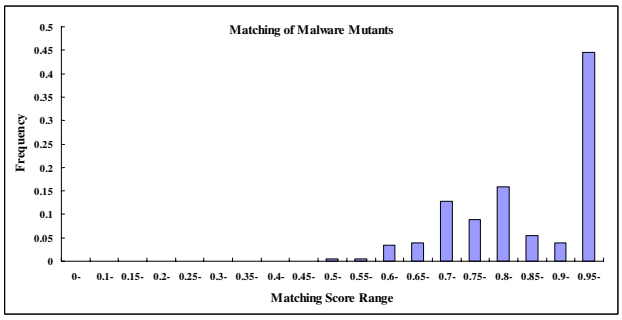

Figure 4. Malware Variants Pattern Matching. Each x-axis value stands for a range of matching scores

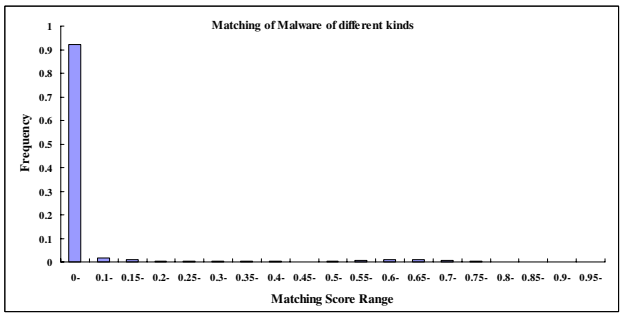

Figure 5. Malware Variants Pattern Matching. Each x-axis value stands for a range of matching scores

5 Discussion

5.1 Comparison with Previous Work

This section compares with three previous methods [9, 12, 8] which statically analyze program semantics to detect metamorphic malware.

As pointed out in [12], the control flow graph comparison method of [9] can only handle very simple program obfuscations. For example, the detection algorithm only allows `noop` instructions to appear between matching instructions. By comparison, the method proposed in this paper can handle a much wider range of obfuscations than this. It can also detect program mutants that have similar but not identical behaviors.

The method of [12] uses semantic templates to detect malware that has certain common behaviors. A template is manually generated by studying the common behavior of a set of collected malware instances; how to generate a general semantic template is not addressed. Our method, by contrast, proposes an automatic pattern generation method that characterizes a program's semantics. Furthermore, we argue that the proposed method is harder to be bypassed. The proposed method uses maximum weighted matching to be tolerant to inaccurate program disassembly and static analysis.

Chouchane and Lakhotia [8] use "engine signatures" to assist in detecting metamorphic malware. That work, however, can only deal with known instruction-substituting metamorphic engines. There are many ways to create metamorphic engines, by no means limited to instruction substitution. Moreover, their technique can be defeated by shrinking substitution methods. Our proposed method does not rely on specific engines. It characterizes and compares a program's semantics more generally. It uses control flow and data flow analysis, and is more robust against complex metamorphism.

5.2 Limitations

There are two main limitations that can cause the failure of the proposed method. The major limitation is due to the use of static analysis. Since static analysis does not execute the program, run-time information is not available to derive a more accurate pattern. A case in point is static disassembly, which is not guaranteed to be 100% accurate [19]. Various techniques, such as indirect addressing, self-modifying code, and dynamic code loading can lower the accuracy of static disassembly. A number of malware instances downloaded from VX Heavens could not be disassembled properly, even though they were executable. One such technique used by such malware is manipulation of the section table in the program header. Examples include manipulating the program entry pointer to start in a non-code section, and changing the size of the code section to a false value. Such techniques can cause disassembly to conclude incorrectly that the file is not executable. Polymorphic malware is a special case of self-modifying code for obfuscation. The solution to this problem would be to combine dynamic analysis with static analysis to improve the disassembly accuracy. Many techniques to improve disassembly accuracy have also been proposed [13], but are not currently implemented in our prototype.

The second major limitation results from code evolution. Similarity of a mutant to an original code version largely depend on how the new instance evolves. If the majority of functionalities (i.e. the malware payload) of a mutant is replaced, only a small part will be matched, resulting in a low similarity score. In addition, the malware writer can purposely insert random "junk" functionalities *in terms of actual system calls made* to lower the matching score. Theocratically, if the number of "junk" functionalities goes unbounded or infinite, the proposed method will likely fail. To address this issue, it may be necessary for the program pattern to be focused on specific functions of the malware, rather than on the entire program's function. For instance, a whitelist of system calls or library calls can be built to fil-

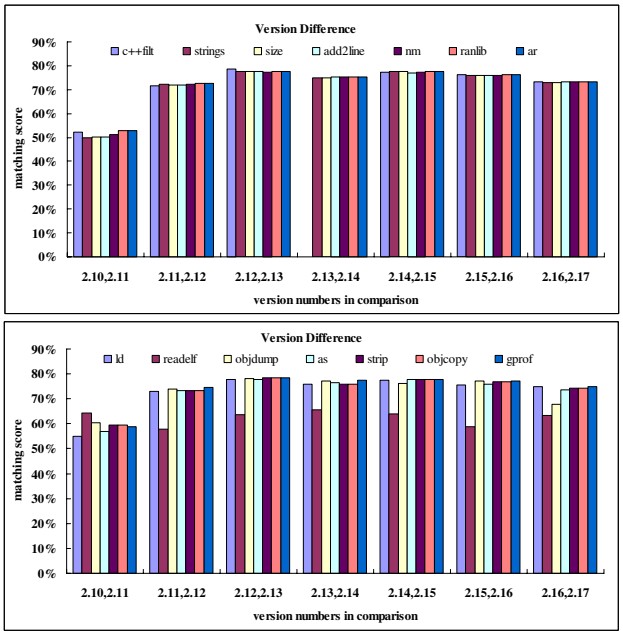

Figure 6. Pattern Matching of GNU Binutils programs. Each pattern matching is performed between two consecutive versions of a GNU Binutils program.

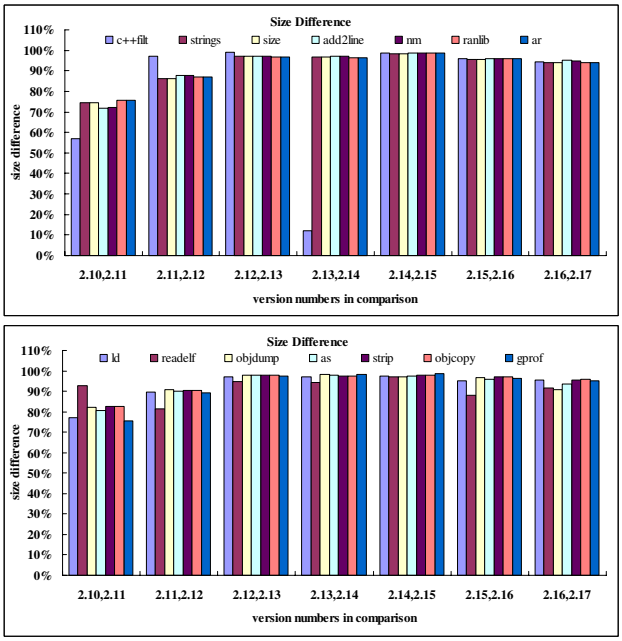

Figure 7. Size comparison of GNU Binutils programs. Each size comparison is performed between two consecutive versions of a GNU Binutils program.

ter out common functionalities that are usually unimportant such as `printf`.

6 Conclusion

This paper presented a new approach to characterize and compare program semantics. A direct application of the proposed method is to recognize metamorphic malware programs, which conventional signature-based detection methods are less successful at detecting. The proposed method has been prototyped and evaluated using randomized benchmark programs, various types of real malware programs, and multiple releases of the GNU binutils programs. The evaluation results demonstrate three important capabilities of the proposed method: (a) it has great promises in identifying metamorphic variants of common malware; (b) it distinguishes easily between programs that are not related; and, (c) it can identify and detect program variations, or code reuse. Such variations can be due to insertion of malware (such as viruses) into the executable of a host program, or programs revision. Thus an indirect application of the proposed work is to help localize an occurrence of one fragment of code inside another program using the maximum matching.

Future work will consider more accurate analysis of the parameters passed to library or system functions. We also believe the method's ability to identify similarities between binary executables will be useful for code attribution and other reverse engineering purposes.

Acknowledgements. The authors would like to thank Chongkyung Kil for randomizing the benchmarks using the ASLP tool, and providing them with the results.

References

[1] SPEC CPU2000. http://www.spec.org/cpu/.

[2] Apache Web Server. http://httpd.apache.org/.

[3] eEye Digital Security company. http://www.eeye.com.

[4] GazTek Web Server. http://gaztek.sourceforge.net/ghttpd/.

[5] GNU Binutils. http://www.gnu.org/software/binutils/.

[6] Intel Architecture Software Developers Manual. Volume 2: Instruction Set Reference.

[7] VX heavens. http://vx.netlux.org.

[8] M. R. Chouchane and A. Lakhotia. Using Engine Signature to Detect Metamorphic Malware. In *Proceedings of the 4^{th} ACM Workshop on Rapid Malcode*, November 2006.

[9] M. Christodorescu and S. Jha. Static Analysis of Executables to Detect Malicious Patterns. In *Proceedings of the 12^{th} USENIX Security Symposium*, pages 169–186, Auguest 2003.

[10] M. Christodorescu and S. Jha. Testing malware detectors. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 34–44, 2004.

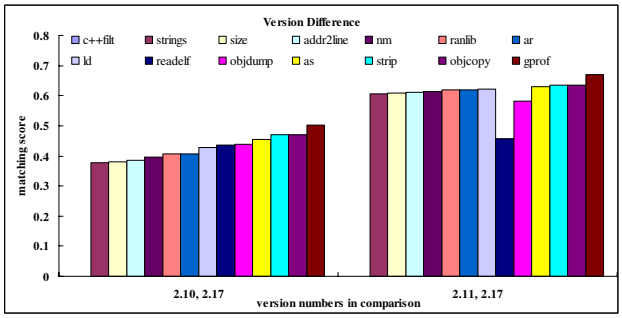

Figure 8. Pattern Matching of GNU Binutils programs. Each pattern matching is performed between version 2.10 and 2.17, or version 2.11 and 2.17 of a GNU Binutils program.

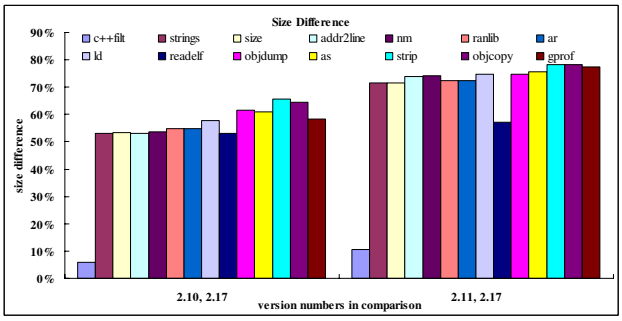

Figure 9. Size comparison of GNU Binutils programs. Each size comparison is performed between version 2.10 and 2.17, or version 2.11 and 2.17 of of a GNU Binutils program.

[11] M. Christodorescu, S. Jha, and C. Kruegel. Mining Specifications of Malicious Behavior. In *Proceedings of the 6th ESEC/FSE*, September 2007.

[12] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-Aware Malware Detection. In *Proceedings of 2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 32–46, May 2005.

[13] C. Cifuentes, M. Van Emmerik, D. Simon D.Ung, and T. Waddington. Preliminary Experiences with the Use of the UQBT Binary Translation Framework. In *Proceedings of the Workshop on Binary Translation*, pages 12–22, October 1999.

[14] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer Overrun Detection using Linear Programming and Static Analysis. In *Proceedings of the 10th ACM Conference on Computer and Communications Security(CCS)*, pages 345–354, October 2003.

[15] J. Gordon. Lessons from Virus Developers: The Beagle Worm History Through April 24, 2004. May 2004.

[16] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *Proceedings of the 22th Annual Computer Security Applications Conference (ACSAC'06)*, December 2006.

[17] E. Kirda and C. Kruegel. Behavior-based Spyware Detection. In *Proceedings of the 15th USENIX Security Symposium*, pages 273–288, Auguest 2006.

[18] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic Worm Detection Using Structural Information of Executables. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID05)*, pages 53–64, September 2005.

[19] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static Disassembly of Obfuscated Binaries. In *Proceedings of the 13th USENIX Security Symposium*, pages 255–270, Auguest 2004.

[20] W. Lee and S. Stolfo. Data Mining Approaches for Intrusion Detection. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

[21] G. McGraw and G. Morrisett. Attacking Malicious Code: A Report to the Infosec Research Council. *IEEE Software*, 17(5):33–41, Sept./Oct. 2000.

[22] S. S. Muchnick. *Advanced Comiler Design Implementation*. Morgan Kaufmann Publisher, CA, USA, 1997.

[23] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *Proceedings of 2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 226–241, May 2005.

[24] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Proceedings of the 22th Annual Computer Security Applications Conference (ACSAC'06)*, December 2006.

[25] A. Somayaji S. Forrest, S. Hofmeyr and T. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of 1996 IEEE Symposium on Security and Privacy*, pages 120–128, May 1996.

[26] P. Szor. *The Art of Computer: Virus Research and Defense*. Symantec Press, NJ, USA, first edition, 2005.

[27] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of 2001 IEEE Symposium on Security and Privacy*, pages 156–169, May 2001.

[28] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the 7th Network and Distributed System Security (NDSS'00) Symposium*, February 2000.

[29] D. B. West. *Introduction to Graph Theory*. Prentice-Hall, NJ, USA, second edition, 2001.

[30] Q. Zhang, D. S. Reeves, P. Ning, and P. Iyer. Analyzing Network Traffic To Detect Self-Decrypting Exploit Code. In *Proceedings of 2007 ACM Symposium on InformAtion, Computer and Communications Security (ASIACCS'07)*, March 2007.