Using Engine Signature to Detect Metamorphic Malware

Mohamed R. Chouchane
Software Research Laboratory
The Center for Advanced Computer Studies
University of Louisiana at Lafayette
+1 337 482-5082
mohamed@louisiana.edu

Arun Lakhotia
Software Research Laboratory
The Center for Advanced Computer Studies
University of Louisiana at Lafayette
+1 337 482-6766
arun@louisiana.edu

ABSTRACT
This paper introduces the "engine signature" approach to assist in detecting metamorphic malware by tracking it to its engine. More specifically, it presents and evaluates a code scoring technique for collecting forensic evidence from x86 code segments in order to get some measure of how likely they are to have been generated by some known instruction-substituting metamorphic engine. A prototype simulator that mimics real instruction-substituting metamorphic engines was implemented and used to conduct several experiments that evaluate the goodness of the scoring technique for given engine parameters. The technique was also used to successfully help track variants of `W32.Evol` to their engine.

Categories and Subject Descriptors
D.4.6 [**Operating Systems**]: Security and Protection – *invasive software;* K.6.5 [**Management of Computing and Information Systems**]: Security and Protection – *invasive software*

General Terms
Measurement, Experimentation, Security.

Keywords
Virus Scanner, Metamorphic Engine.

1. INTRODUCTION
Metamorphic malware is that which propagates by creating transformed copies of its code. The generated code, often called a *variant* of the malware, is typically a working program which has the same functionalities as the original. Metamorphism has been used by malware authors to thwart detection by static-signature-based anti-malware scanners and has the potential to lead to a breed of malicious programs that is virtually undetectable statically and could seriously damage the target computing systems and potentially cause billions of dollars in financial losses over a very short period of time [9]. Most commercial static anti-malware scanners look for malware signatures (typically, a sequence of bytes within the malware code) to declare, with some measure of confidence, that the program being scanned is malicious.

WORM'06, November 3, 2006, Alexandria, Virginia, USA.

The main goal of metamorphism is to change the appearance of a malicious program while keeping its functionality and many metamorphic transformations can be used (or combined) to achieve this goal. A number of these transformations came to be understood thanks to analyses done by people in the anti-malware research community; others were simply advertised by metamorphic malware authors. These transformations include register renaming, code permutation, code expansion, code shrinking and dead code insertions and vary in the amount of effort needed to apply them and the performance/size overhead they impose on the new variant. Detailed discussions of these transformations, and others, can be found in [9]. Most metamorphic malware typically consist of a malicious payload attached to a metamorphic engine. When the engine is called, its algorithm mutates the payload (and sometimes the engine code as well) by applying a number of these transformations to it.

This paper illustrates the *engine signature* approach through the use of an engine-specific scoring procedure that scans a piece of code to determine the likelihood that it is (part of) a program that has been generated by a known instruction-substituting metamorphic engine. This way, all we need to store for detection purposes is information about the engine rather than information about each possible malicious variant it can produce.

The scoring technique is designed for metamorphic engines that transform their input variant of the malware using a finite set of transformation rules mapping instructions to code segments that implement their operational semantics. When such an engine is given as input a (typically malicious) code segment to be transformed, it scans the segment for occurrences of instructions that can be transformed. The engine sometimes needs to perform simple context analyses, or make assumptions about the code, to decide whether or not the instruction can be transformed. The third rule in the transformation system of Figure 3, for example, is semantics preserving only provided that register `eax` be dead at that point. When the engine determines that it is safe to transform the instruction, it probabilistically decides whether or not to replace it with an equivalent code segment. `Evol(ve)`, the engine of `W32.Evol`, is an example of such an engine where each rule has its own application probability.

The scoring technique was inspired by our observation that much (typically, over 50%) of the original variant of instruction-substituting metamorphic malware is transformable; that is, most of its instructions are transformable by the engine. We use the phrase *engine friendliness* to refer to this level of transformability of the variant. Such engines also tend to be written such that their transformation rules preserve this level of transformability across generations; that is, high transformability (i.e., high engine

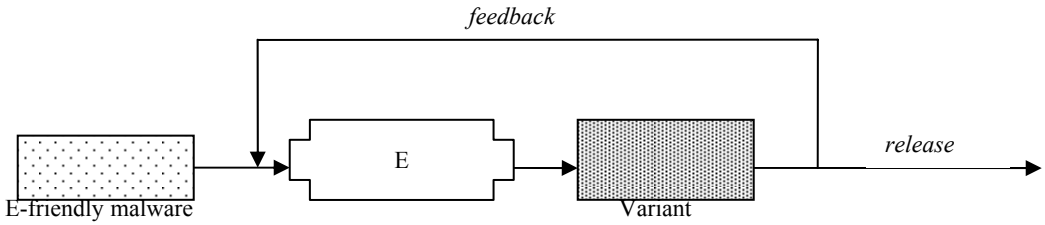

Figure 1. The metamorphic engine E uses its transformation system to probabilistically replace instructions in its input with equivalent code segments. As a result, an intractably high number of new variants could be produced by the engine. The scoring function captures the "density" on codes segments introduced in the variants by the engine to track the variants to their producer: the engine.

friendliness) usually applies to all variants of the malware. This is typically done, in the case of instruction-substituting engines, by ensuring that even the code segments that are used to replace other code segments contain at least one transformable region.

Since metamorphic engines which operate on binaries are notoriously hard to write, not many reliable metamorphic engines have so far been written. But of those that were, some were made available as object files for malware authors to write their payloads and link them to these engines thus adding metamorphism to the new payload. The engine signature approach presented in this paper may then be taken in this case to gather enough forensic evidence to link segments of this new malicious payload to the engine being reused.

Section 2 describes related work. Section 3 applies the engine signature approach for detecting metamorphic malware by presenting a scoring technique for tracking code segments to known instruction-substituting metamorphic engines. Section 4 describes the experiment we conducted to evaluate the scoring technique. Section 5 discusses the evaluation results. Section 6 concludes the work and lists a number of future research problems.

2. RELATED WORK

A number of definitions of malware that may be interpreted as describing metamorphic malware have been given and results about the computability and complexity of their respective detection problems have been established[2][3][8]. These results, while perfectly sound, do not apply in the context of our work: An approximate detection scheme of the output of some well-known metamorphic engine.

Industry-adopted approaches for detecting metamorphic malware still rely heavily on the concept of static signature scanning to detect the malware. Static signature scanning typically scans a program for the presence of sequences of bytes known to belong to known malware (malware which has been received and analyzed by anti-malware vendors). Other schemes run the suspect program and monitor its behaviors hoping to catch any unexpected or known malicious behavior (such as a sequence of call instructions known to be routinely used by some malware.) Such an approach is very impractical given that metamorphic malware authors tend to design it such that intractably many variants can be generated on each run of the engine on some variant. This alone represents a major impediment to static signature scanning. Dynamic analysis can also be thwarted by testing the patience of the emulator or by taking the malicious control flow path only rarely.

Recent work by Karim et al. [5] views variants of malware that permutate and transform some or all of their code when propagating as related to previous variants through evolutionary relationships. Actually, their method also takes advantage of the common practice of code reuse employed by most malware writers. By extracting these evolutionary relationships, they built a phylogeny and, in a way similar to that used in biology to decide whether an organism is likely to be an evolved strain of another, they were able to classify the programs being scanned as being a potentially modified version of some malware. Chritodorescu et al. [4] built a "semantic rather than purely syntactic" algorithm for detecting malware. Their algorithm uses its knowledge of a specification of malicious behavior, described using a template, to detect if a given program satisfies the behavior. They show that their implementation, approximating a solution to this undecidable template-matching problem, outperformed McAfee® VirusScan® by detecting a larger number of variants of a known malware. Walenstein et al. [10] take a code normalization approach to reduce variants of known metamorphic malware to a common normal form. They use this form as a signature for that malware and declare a program to be a variant of the malware if it eventually reduces to that form. A similar normalization approach was proposed by Brushi et al. [1]. Kruegel approached the issue by extracting patterns from executables to track them to some known malicious malware [7].

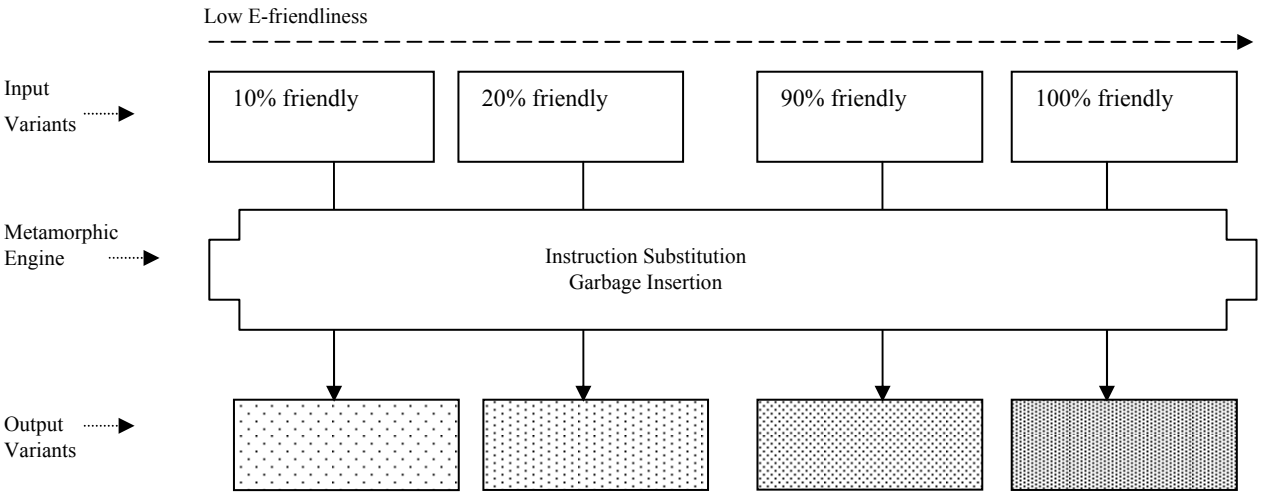

Figure 2. An increase in the engine-friendliness of the input variant induces an increase in the density of the output variants with clues that have been introduced by the engine.

```
mov [esi+4], 9      →    mov [esi+4], 6
                         add [esi+4], 3
mov [ebp+8], ecx    →    push eax
                         mov eax, ecx
                         mov [ebp+8], eax
                         pop eax
push 4              →    mov eax, 4
                         push eax
push eax            →    push eax
                         mov eax, 2Bh
```

```
Clue Set ={mov add,
           push mov mov pop,
           mov push,
           push mov}
```

Figure 3. At left is a subset of the transformation system of `W32.Evol`. When any of the left hand sides is met in the variant to be transformed, the engine non-deterministically replaces it with the corresponding code segment. The clue set is constructed from the transformation system by extracting the opcodes of the right hand sides.

These approaches either track variants of metamorphic malware to other variants or to some template capturing some abstraction of some malicious behavior. Our scoring function, however, not only assists in matching variants of malware to the malware, but also in tracking programs, or even code segments, to known code-substituting metamorphic engines of the kind described in the introduction.

Our approach is analogous to authorship analysis when applied to determine the likely author of some typically high-level language program. A considerable amount of work on authorship analysis has been done [6]. It is somewhat relevant to our approach in that our attempt to track metamorphic malware to its engine is analogous to tracking code to its authors. Most of these forensic analyses of code assume the availability of source code, which is generally not available when analyzing potentially malicious binaries.

3. THE SCORING TECHNIQUE

We henceforth let *E* denote some instruction-substituting metamorphic engine as described in the introduction. We will use the term *clue* to refer to any fraction, typically a sequence of instructions, of any information extracted from a code segment and suggesting that the segment may have been generated by *E*. We denote by *T* the rule set carried by *E*. Each rule in *T* maps an instruction (the *left hand side*) to a sequence of one or more instructions. This sequence is referred to as the *right hand side* of the rule. Right hand sides are thus examples of clues. Such clues are chosen and assigned *weight*s equal to their instruction count. This choice of weight assignment is arbitrary and may not be the best in all cases, but it suits our purposes.

We say that a code segment is *E-friendly* if it was written "with *E* in mind"; that is, the segment has a non-zero frequency of the left hand sides of *T* (i.e., transformable instructions). The high *E*-friendliness of a given variant must hold across subsequent generations of the malware. This is usually achieved by requiring

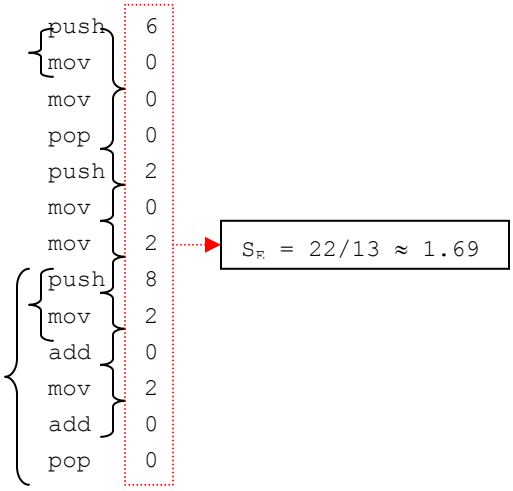

Figure 4. An illustration of the scoring approach on a code segment suspected of having been generated by Evol.

at least one occurrence of a left hand side in most, if not all, right hand sides.

Each rule in T is accompanied with its *rule application probability* (the probability that the rule will be applied when its left hand side is encountered in the segment to be transformed.) The set T must be explicitly carried by the engine and is assumed to be extractable manually, or interactively from the engine in a laboratory environment. This approach will of course work only when the static disassembly of the executable to be scanned is successful. Given a code segment V, suspected of being (part of) the output of E, we reduce each instruction in V to its opcode mnemonic. This abstraction of the actual instruction is actually needed to represent the (typically intractably large) set of possible right hand sides that a transformation involving variables taking on scalar values might generate.

We define a *scoring function* that takes as input a code segment (a sequence of x86 opcode mnemonics) and returns a score for that sequence. For a given code segment V, the score of V with respect to E, which we denote by $S_E(V)$, is a measure of how likely V is to be (part of) a program generated by E. $S_E(V)$ is proportional to the forensic evidence linking V to E and inversely proportional to the instruction count of V. It can be viewed as the density of V with clues from the transformation rules of E. The expression of S_E takes into account the fact that some of the clues are more informative than others. The scoring function is given by the expression

$$S_E(V)=\sum_c \sum_s w_c\, e_{cs} / |V|,$$

where $|V|$ is the instruction count of V, w_c is the weight of clue c, and $e_{cs} = 1$ if clue c is at site s and 0 otherwise. A naïve algorithm computing this function would simply do a linear scan of V. For each instruction i visited, it would determine whether i is the beginning of an occurrence of one or more clues. If it is, it accumulates the sum of the weights of these clues in some variable. It would finally divide the accumulated sum by the instruction count of V then return the result. Figure 4 gives an example using the scoring function.

4. EVALUATIONS

We implemented a prototype simulator of an instruction-substituting metamorphic engine and used it to run two experiments to evaluate the scoring function. A code segment is simulated by abstracting it to a list of opcode mnemonics in the same order in which they appear in the segment. The rule set of the engine is simulated using a list of pairs of such tuples. The simulator is restricted to mapping single-element tuples to larger ones, thus capturing the metamorphic transformation which substitutes an instruction with possibly larger code segments. The rule set we chose to use was that used by `Evol(ve)`, the metamorphic engine of the `W32.Evol` malware.

Evaluation 1 (Tracking typical engine output to the engine): The goal of this first experiment was to determine how well, and for what parameter choices, the scoring function can assist in discriminating the engine's output from arbitrary code segments. The experiment actually consisted of a number of simulations. Each simulation generated a "first generation" code segment with fixed engine-friendliness and used the engine to produce a thousand each of 2nd, 3rd, 4th, 5th, 6th, and 7th generation descendants of the segment. The choice to go that deep in generating variants of the code was arbitrary but nevertheless sufficient to make preliminary claims about the scoring function. We computed the scores of the variants thus simulated and analyzed their frequency distributions. Figures 5 and 6 give the frequency distributions of the second through seventh generations of variants produced on input programs of engine-friendliness 5% and 50% respectively.

Evaluation 2 (Tracking variants of fixed metamorphic malware to the engine): The goal of this second experiment was to determine how well, and for what parameter choices, the scoring function can assist in discriminating variants of known malware from arbitrary code segments. For this experiment we took a variant of the `W32.Evol` malware, extracted its opcode list, and using the transformation rules that this malware's engine uses, generated a thousand each of 2nd, 3rd, and 4th generation variants of the opcode string. We computed the score of each of these variants and analyzed the frequency distribution of these scores. The simulation results are shown in Figure 7. We have also computed the scores of real `W32.Evol` variants we have collected from infected programs. The 2nd generation, 3rd generation, and 4th generation variants scored 1.62, 1.95, and 2.13 respectively.

5. DISCUSSION

The first evaluation shows that the scoring method successfully managed in several cases to score the engine's output considerably higher than engine-independent segments. It can be seen from the simulation results that the function performed particularly well on all variants when the original variant friendliness was over 50%. For lower engine-friendliness values, say 5%, the function was only successful in telling later generation variants from engine-independent segments. For example, some engine-independent segments scored over 0.013 while minimum scores of 0.009 were observed for second generation variants of 5% friendly segments. This implies that writing malware with low engine-friendliness would be one way to evade our approach (at least for earlier generations). But again, low friendliness implies that a good section of any variant of the malware will not be transformable hence defeating the very

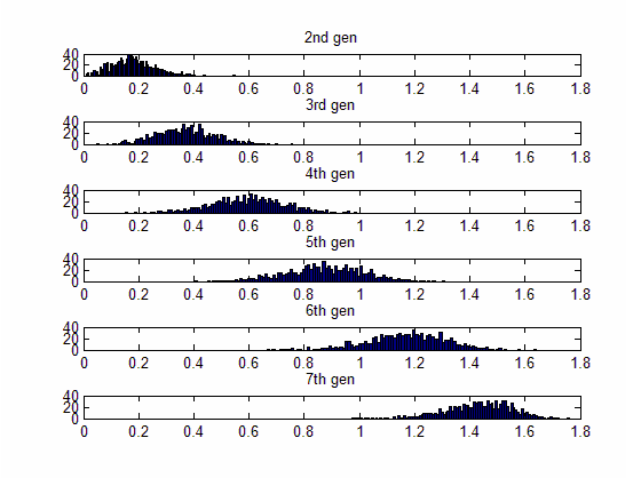

Figure 5. The frequency distribution of the scores of 2nd to 7th generations with initial engine-friendliness 5%.

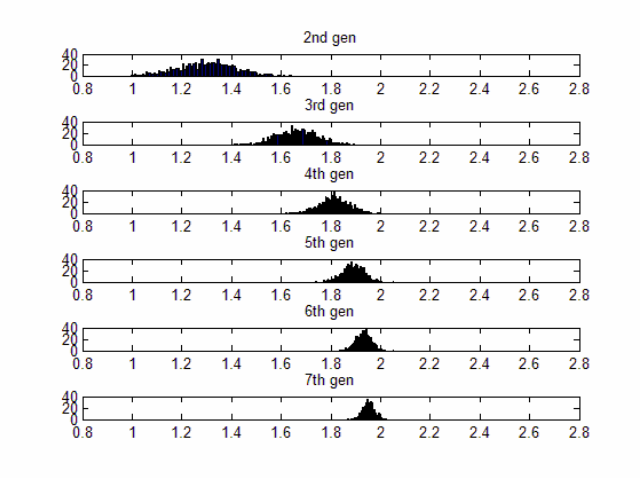

Figure 6. The frequency distribution of the scores of 2nd to 7th generations with initial engine-friendliness 50%.

purpose of metamorphism and leaving itself open to traditional static signature scanning.

Both evaluations produced a Gaussian-like score distribution. For the second evaluation, this suggests that segments from `W32.Evol` could be tracked to `W32.Evol`'s engine by measuring their closeness to the means of the distributions corresponding to each generation. An inspection of the simulation results reveals that segment scores seem to somehow converge towards some small range of values as the segment mutates. We have observed this trend as we were experimenting with other friendliness, rule application probability combinations and it can also be clearly observed in Figure 6. This small range of values could then be used to give some measure of confidence of how likely a code segment is to be part of a later variant of some metamorphic malware using `Evol(ve)` as its engine.

6. CONCLUSIONS AND FURTHER WORK

We introduced a novel approach for dealing with metamorphic malware. This approach capitalizes on the engine's power to track its output to it. More specifically, it takes advantage of the fact that metamorphic engines, in order to generate an output program that is as different as possible from the input, typically require their input to be highly transformable. Intuitively, an engine signature is forensic evidence extracted from some code segment and giving some measure of confidence that the segment is part of a program (usually malware) that was generated by the engine.

We used a scoring function to illustrate this approach on an instruction-substituting metamorphic engine, that of `W32.Evol`. More analysis would perhaps be needed should the scanner wish to gather more evidence that the code being scanned is in fact from a variant of `W32.Evol`. As for the authorship-tracking dimension, the function did particularly well on all generations except for those variants whose original engine-friendliness is below 10% for the case study transformation system.

Other flavors of weight assignment can also be explored. For example, a garbage segment should be given more weight than a right hand side segment; intuitively, odds that some random code segment contains a do-nothing segment, especially a large one, known to be routinely inserted by some engine at more than one

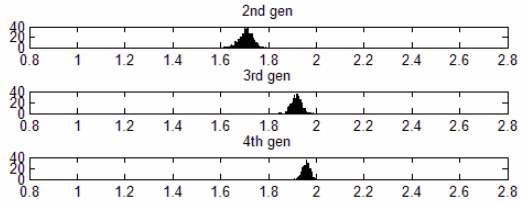

Figure 7. The frequency distribution of the scores of 2nd to 4th generations of simulated `W32.Evol` variants.

location are considerably lower than those for programs generated by the engine on input a friendly malware. Clues that cannot be altered across generations should weigh more than those which can; for example, some clue may contain a section that is not transformable by the engine and hence remains in the code across generations.

Some engines, such as W32.Simile (a.k.a. MetaPHOR), shrink code by applying transformations mapping relatively large code segments to smaller ones. The shrinking part (or application of expanding rules both ways), should adversely affect the current scoring function if the engine takes the shrinking direction of the rules (considerably) more often than it does the expanding direction. And, even so, in order to thoroughly defeat the function, most of the smaller segments must be of minimal size; that is, in the order of one instruction per segment, leaving the malware authors with fewer transformation options.

We will also investigate the possibility of adapting this research to determine toolkit authorship to actually track recently released malware to known malware generation tookits.

Acknowledgment. This work was supported in part by funds from the Louisiana Governor's Information Technology Initiative. The authors thank Rachit Mathur for extracting the samples and rule set `W32.Evol`.

7. REFERENCES

[1] Brushi, D., Martignoni, L., and Monga, M. Using Code Normalization for Fighting Self-Mutating Malware. In *Proceedings of the International Symposium of Secure Software Engineering* (Arlington, VA, 2006).

[2] Chess, D. M., and White, S. R. An Undetectable Computer Virus. In *Proceedings of Virus Bulletin Conference* (2000).

[3] Cohen, F. Computational Aspects of Computer Viruses. *Computers & Security, 8* (1989), 325-344.

[4] Christodorescu, M., Jha, S., Seshia, S. A., Song, D., and Bryant, R. E. Semantics-Aware Malware Detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P'05)* (Oakland, CA, USA, May 8-11, 2005)

[5] Karim, M. E., Walenstein, A., Lakhotia, A. ,and Parida, L. Malware Phylogeny Generation using Permutations of Code. *European Research Journal of Computer Virology 1, 1-2* (Nov. 2005) 13-23.

[6] Krsul, I., Spafford, E. H. Authorship Analysis: Identifying The Author of a Program. Technical Report 96-052, 1996.

[7] Kruegel, C., Kirda, E., Mutz, D., Robertson, W., and Vigna, G. Polymorphic Worm Detection Using Structural Information of Executables. In *Proceedings of the 8th Symposium on Recent Advances in Intrusion Detection (RAID)* (Seattle, WA, USA, September 7-9, 2005).

[8] Spinellis, D. Reliable Identification of Bounded-Length Viruses is NP-Complete. *IEEE Transactions on Information Theory*, 49, 1 (2003), 280-284.

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

[10] Walenstein, A., Mathur, R., Chouchane, M. R., and Lakhotia, A. Normalizing Metamorphic Malware Using Term Rewriting. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006)* (Sheraton Society Hill, Philadelphia, PA, USA, 2006).