Statistical Signatures for Fast Filtering of Instruction-substituting Metamorphic Malware

Mohamed R. Chouchane
CACS
UL Lafayette
Lafayette, Louisiana, USA
mohamed@louisiana.edu

Andrew Walenstein
CACS
UL Lafayette
Lafayette, Louisiana, USA
walenste@ieee.org

Arun Lakhotia
CACS
UL Lafayette
Lafayette, Louisiana, USA
arun@louisiana.edu

ABSTRACT

Introducing program variations via metamorphic transformations is one of the methods used by malware authors in order to help their programs slip past defenses. A method is presented for rapidly deciding whether or not an input program is likely to be a variant of a given metamorphic program. The method is defined for the prominent class of metamorphic engines that work by probabilistically selecting instruction-substituting program transformations. A model of the probabilistic engine is used to predict the expected distribution of instruction forms for different generations of variants. These predicted distributions form a type of "statistical signature" for the output of the metamorphic engines. A classifier is defined based on distance between the observed and the predicted instruction form distributions. A case study using the `W32.Evol` virus shows the classifier can distinguish between malicious samples from multiple generations. The classification method may be useful for practical malware detection by serving as an inexpensive filter to avoid more in-depth analyses where they are unnecessary.

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, Security

Keywords

Virus Scanner, Metamorphic Engine

1. INTRODUCTION

Malware authors frequently try to disguise their programs in order to avoid detection. One way they do this is to use a system that can automatically generate multiple different *variants* of a program. One way this can be done is by using a transformation system such as a *metamorphic engine* [14]. A metamorphic engine transforms the program during propagation; this alters the form of the descendant copies so as to reduce the number of constant patterns that can be used as a basis for detection.

Multiple approaches can be used to counteract the effects of such variation-inducing transformations. For example, there are program normalization approaches that could be used to remove the differences between variants and allow them to be detected more easily as a group [10]. Approaches in this vein include engine-specific normalization approaches [16], as well as more general normalization methods [2, 5]. Alternatives include more complex methods to detect unchanged program properties such as execution sequences. For instance, it is possible to use semantic-level program analyses to recognize a program's behavior [11, 8, 4] even if its form has been metamorphically transformed.

Regardless of whether such proposals work well enough, a problem that arises is that the cost of analysis may be so high that it is infeasible to use them for every possible executable that must be tested. This leads to the question: is there a fast method for testing an executable to determine whether it is worthwhile expending the cost of more intensive analysis?

This paper presents a method for constructing a decision procedure that may be used to try to efficiently determine which transformation engine is likely to have produced a program in question, if any. The approach is targeted to that class of metamorphic engines that probabilistically select instruction-substituting transformations, and possibly rename registers and permute instruction ordering. The approach leverages the fact that the expected distribution of various *instruction forms* within the descendants can be predicted from the probabilities of the various instruction substitution transformations. That is, the engines leave a type of *statistical signature*, i.e., a type of "engine signature" [3]. A decision procedure can be constructed by comparing the predicted instruction form distributions to the actual distributions found in a suspect program. A case study on the metamorphic virus `W32.Evol` suggests that the approach may be feasible in practice.

Section 2 presents the overall approach, as well as the class of metamorphic engines covered by the method. Section 3 describes a method for creating a predictor of the distribution of instructions for any given metamorphic generation; it also describes how to construct a procedure for deciding

WORM'07, November 2, 2007, Alexandria, Virginia, USA.

whether a given input sample is likely to have been generated by the modeled engine. Section 4 presents the case study. Conclusions and open issues are presented in Section 5.

2. PROBABILISTIC ENGINE MODELING

One of the fundamental constraints on closed-world metamorphic malware is that the engine must only produce program outputs that it is itself able to parse and manage; if it fails to do so, at least some of the offspring it creates will not be able to run and multiply correctly [9]. Furthermore, it is a principle of good metamorphic engine design that the engine is able to transform much of the program, else it will leave constant portions that may be used for recognition [3]. In addition, it seems plausible that many real metamorphic engines will have *limited repertoires* of transformations they are able to make. That is, they will be capable of generating output instructions selected from only a limited set of forms. Repeated application of these transformations (over generations) may only serve to emphasize this limited repertoire.

In combination, the above observations suggest it may be effective to identify variants of the metamorphic program by examining the programs for indications of being constructed by the limited repertoire specific to the metamorphic engine (as it applied to the initial program). A similar idea underlies attempts to detect authorship after noting that authors can be viewed as generators with idiosyncratic propensities for generating certain structures and phrases [7].

The present paper uses similar motivation to model a metamorphic engine as a probabilistic language generator, and then uses statistical properties of input programs as measures indicating the likelihood that they were "authored" by the modeled engine. It focuses on the specific class of metamorphic programs which operate by probabilistically selecting instruction-substituting transformations, perhaps in combination with register renaming and instruction permutation. Examples of these sorts of transformations are illustrated in Figure 1. Several examples of metamorphic engines with such combinations of transformations are found in the literature (e.g., Ször [14]). We define this class more formally using a transformation engine model, as follows, and then discuss properties of the class it identifies.

Let $\mathcal{M}$ denote a metamorphic engine with a finite set of n transformation rules t_i, of the form $T = \{t_1, t_2, ..., t_n\}$. Let an *instruction form* be an abstracted machine language statement such that specific registers and constants are not considered (but operation and indexing modes, etc., are). Let each t_i map one instruction pattern l_i to a non empty set $r_i = \{r_i^{i_1}, r_i^{i_2}, ..., r_i^{i_{max}}\}$ of possible code segments with distinct instruction forms. Thus, each application of any t_i alters the composition of instruction forms in one of a set of known ways. Each t_i is accompanied with a *rule application probability* $\mathcal{P}_i$ and each element $r_i^{i_j}$ in its right hand side is accompanied with a *probability of use* $\mathcal{P}_i^{i_j}$. The probabilities of use must add up to 1; that is, for each right hand side r_i, we have $\sum_{j=1}^{i_{max}} \mathcal{P}_i^{i_j} = 1$. Two different rules must not have left hand sides with identical instruction forms. T can be considered $\mathcal{M}$'s repertoire of instruction form transformations.

The rule application probabilities and the probabilities of use are assumed to be either (1) *fixed* for each rule and extractable interactively from the metamorphic engine or (2) implemented (as in `W32.Evol` and `W32.Simile`) using a random or pseudo-random number generating procedure that is part of the engine. If the latter is the case then we assume that the choices are uniform for each rule; that is, the rule application probability $\mathcal{P}_i = 1/k_i$ for each rule t_i, where k_i is the number of choices uniformly made to determine whether to apply rule t_i. The probability of use is $\mathcal{P}_i^{i_j} = 1/i_{max}$ for each $r_i^{i_j}$.

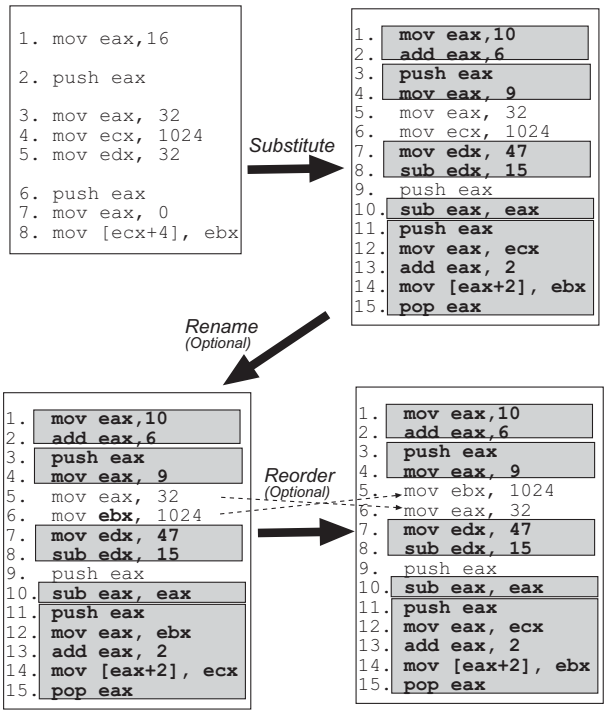

Figure 1: Examples of considered transformations

Let $\mathcal{M}$ execute by selecting some subset S of instructions to attempt to transform based on the outcomes of the probability function applied to each instruction that it may transform. Let S_S be the method that $\mathcal{M}$ uses to select S, and let S_T be the traversal method that $\mathcal{M}$ uses to apply transformations on S (S_T can be seen as the iterator method and S_S the iteration set constructor). At each transformation point matching the left hand side l_i of a certain rule t_i, with probability $\mathcal{P}_i$ the engine transforms the instruction. Once this decision is made, with probability $\mathcal{P}_i^{i_j}$ the engine substitutes l_i for $r_i^{i_j}$. These probabilities are assumed to be fixed, or exactly learnable from the description of the engine, for each rule of the transformation system.

Figure 2 provides a fragment of an example rule set. The set in the figure uses just one possible syntax for instruction forms in which special symbols `reg` and `imm` are non-concrete; for instance, both of the instructions `push %ebp` and `push %ebx` are of the same instruction form, namely `push reg`.

Note that some of the rules may be viewed as "garbage"-inserting, i.e., as inserting code that is irrelevant to the overall behavior of the output program. The third rule of Figure 2 is an example of one such rule. The right hand side of this rule has different operational semantics than the left hand side: the `mov reg, imm` instruction stores a value in register `reg`. This assignment may corrupt the semantics

	l_i	$\rightarrow$	$\{r_i^{i_1}$	$r_i^{i_2}$	$r_i^{i_3}\}$
1	`mov [reg1+imm], reg2`	$\rightarrow$	`push reg`	`push reg`	
			`mov reg, imm`	`mov reg, reg1`	
			`mov [reg1+reg], reg2`	`add reg, imm1`	
			`pop reg`	`mov [reg+imm2], reg2`	
				`pop reg`	
2	`mov reg, imm`	$\rightarrow$	`mov reg, imm1`	`mov reg, imm1`	`mov reg, imm1`
			`add reg, imm2`	`sub reg, imm2`	`xor reg, imm2`
3	`push reg`	$\rightarrow$	`push reg`		
			`mov reg, imm`		
4	`sub reg, reg`	$\rightarrow$	`xor reg, reg`		

Figure 2: A fragment of a modeled transformation set using abstracted assembly notation

of the program being transformed. This risk, however, is non-existent in the case of `W32.Evol` because its variants are crafted such that the rule modifies register `eax`, which is never live after each `push eax` instruction in the code of any of its variants (of course modulo some programming bugs).

The above model is general enough to allow for a variety of known and envisioned implementation styles for metamorphic engines. A particular engine, for example, might implement the selection and traversal functions S_S and S_T by performing a linear sweep disassembly and then sequentially searching for applicable transformation rules for each item in the linear sweep. The real engine might implement the rules t_i as separate rules such that each t_i can be seen as a logical grouping based on the instruction form of the l_i. The model is also general because it makes no statements about additional rule constraints, and permits several other additional types of transformations. These can be seen as relaxations on more strict models.

First, the model makes no statements about additional constraints on the transformations. For example, certain transformations may be selected by the engine only when certain conditions hold. The garbage-inserting rule described above is an example of when an engine may simply make an *assumption* that the condition "immediately after each `push eax` instruction, the contents of register `eax` may be safely altered" is always true. See Walenstein *et al.* [17] for more on the design choices of metamorphic malware. These constraints are permitted because the probabilistic procedure presented in this paper works regardless of the effect of the transformations on the semantics of the malware. The set of variants generated under the restriction that certain conditions must hold is simply a subset of that which can be generated if transformations are unconditionally applied.

Second, the engine is allowed to perform transformations apart from the modeled ones so long as they do not alter their *instruction-composition*. That is, the engine may choose to perform extra transformations to the variants being transformed as long as they do not change the distribution of the instruction forms. Permitted transformations, therefore, include register renaming and instruction reordering.

3. STATISTICAL SIGNATURE-BASED CLASSIFIER

It is typical that variants of metamorphic malware are themselves transformable, giving rise to *generations of descendants* of a given variant. We will use the term *archetype* to refer to the variant of the malware that one has at hand and whose descendants one would like to be able to detect. Note that, for a given positive integer n, a program P is an n^{th}-generation descendant of the archetype if there exists a sequence of n engine runs, feeding the output of each run as input to the next run, starting at the archetype and returning P. Note also that a given program may belong to one or more generations of descendants of the archetype.

Once a particular probabilistic instruction-substituting metamorphic program is modeled, it is possible to predict the frequency with which various instructions may be found in any of its descendants. With such predictions it is possible to define a classifier which uses these predictions to decide whether a given sample is likely to be a descendant of the given metamorphic engine.

3.1 Instruction form distribution prediction

In what follows, the *actual distribution vector*, denoted $ad(P)$, of a program P refers to a vector of tuples such that its first component represents an instruction form, and the second the number of occurrences of that instruction form in P. All of the instruction forms of P's instruction set are represented in the distribution vector, and no redundancies are allowed.

A procedure is defined below that *predicts*, for given positive integer n and archetype a of an instruction-substituting malware, the *average instruction form distribution* of n^{th}-generation descendants of a. The predicted distribution for generation n of archetype a is denoted $pd_n(a)$. This distribution is simply a vector of tuples as per the $ad(P)$, but whose counts represent the expected average of counts for the instruction forms which may occur within the descendants. This is the vector we propose to use as a "statistical signature" with which to calculate the likelihood of being a variant derived via the metamorphic engine.

The motivation for choosing this distribution measure derives, in part, from the observation that the average frequencies of the instruction forms in any given variant output by an instruction-substituting engine $\mathcal{M}$ can be *exactly*

predicted when given the frequencies of the instructions in the archetype and the full description of the probabilistic transformation system of $\mathcal{M}$. The expectation is that the averaged vector is likely to resemble many of the the actual distribution vectors of the n^{th}-generation descendants of the archetype.

The predicted distribution vector for the first generation, $pd_1(P)$ is returned by the procedure described in Algorithm 1. The procedure takes as input a metamorphic engine model in the form of T (the transformation set) and the actual distribution vector $ad(A)$ for the malware archetype A. It returns a predicted distribution vector of first generation descendants of A. The expression $v[i]$ is the component of distribution vector v that represents instruction form i. This component holds the frequency of form i in the code segment whose distribution vector is v. Implicit in the procedure is the availability of the probabilistic transformation system of $\mathcal{M}$. (The rule application probabilities P_i and probabilities of use $P_i^{i_j}$ are defined in Section 2.) The algorithm can be optimized to handle only the subset of the instruction set composed exclusively of those instructions which appear in its input probabilistic transformation system and malware variant. This way a smaller vector containing only the frequencies of these instructions would have to be constructed and manipulated.

Algorithm 1: Distribution Prediction.
Input: $(T, ad(a))$
Output: $pd_1(a)$

foreach instruction I
  if I is identical to the left hand side l_i of some rule of T
  $pd_1(a)[I]$+=$\lceil(1-\mathcal{P}_i)\times ad(a)[I]\rceil$
  foreach $r_i^{i_j}$
    foreach instruction I' in $r_i^{i_j}$
      $f_i^{i_j}[I']$= count of I' in $r_i^{i_j}$;
      $pd_1(a)[I']$+=$\lceil f_i^{i_j}[I']\times ad(a)[I]\times\mathcal{P}_i\times\mathcal{P}_i^{i_j}\rceil$;
    else
      $pd_1(a)[I]$+=a$d(a)[I]$;

return $pd_1(a)$;

The outer `foreach` loop of Algorithm 1 can be nested into a new loop that would run it n times, for some positive integer n. After each i^{th} run, the new loop would make the inner loop use $pd_i(A)$ vector as its $ad(A)$ vector. This augmentation allows the basic prediction procedure of Algorithm 1 to produce the predicted frequency vector $pd_n(A)$ of n^{th}-generation descendants of its input archetype.

3.2 Classifier construction

A classifier can be constructed that can be used to match a sample to the metamorphic engine that is most likely to have constructed it, if any. The approach uses the predicted instruction distributions as a type of signature. The method works by constructing predicted instruction distributions $pd_1, \ldots, pd_k$ for k different generations, and testing to see if the distance to the actual distribution found is greater than some threshold. The k selected is specific to the engine in question.

More formally, the classifier is defined as follows. Define a decision vector

$$DV = ((pd_1, \epsilon_1), (pd_2, \epsilon_2), \ldots, (pd_m, \epsilon_m))$$

where each pd_n is the predicted instruction form distribution of generation n, and each ϵ_n is a threshold associated with generation n. A *distance* measure $d_n(A, P)$ is defined between the n^{th}-generation predicted instruction distribution for A, $pd_n(A)$, and a measured or actual distribution of the suspect program P, $ad(P)$. One of many distribution comparison methods could be chosen to implement the distance measure. We chose one based on the Euclidean norm to measure vector magnitude. The Euclidean norm, $||x||$, of a vector $x = (x_1, x_2, ..., x_m)$ is $\sqrt{\sum_{j=1}^{m} x_j^2}$. The distance, $||x - y||$, between x and some vector $y = (y_1, y_2, ..., y_m)$ is $\sqrt{\sum_{j=1}^{m}(x_j - y_j)^2}$. $ad(P)$ is computed by extracting from P the frequency of each of its instructions, which must include the selected set S. Its distance, $d_n(A, P)$, to the predicted distribution vector $pd_n(A)$ of n^{th}-generation descendants of a given malware archetype A is

$$d_n(A, P) = \frac{||ad(P) - pd_n(A)||}{||pd_n(A)||}.$$

On input P, the classifier works by looking for the value of n that maximizes the value of $d_n(A, P)$ such that $d_n(A, P) \geq \epsilon_n$. An alternate scheme selects only some subset of DV as the statistical signature.

4. CASE STUDY

A small case study was performed in order to explore how well the classifier works, i.e., how well the statistical signatures can separate the classes of variants and non-variants. The general design involves extracting the predicted instruction form distributions for several generations, examining how well these matched various simulated variants, and then exploring how well the classifier works. The study was performed using the metamorphic virus `W32.Evol`. This study is limited since it uses only one engine; it serves only as a proof-of-concept that the approach may be used to discriminate arbitrary code from possible descendants of metamorphic malware using a non-trivial metamorphic engine.

4.1 Subjects and Preparation

`W32.Evol` was chosen to illustrate the distribution prediction approach because (a) it is considered to be a typical example of a complex metamorphic engine [14], and (b) it meets our definition of malware equipped with an instruction-substituting metamorphic engine. The archetype of `W32.Evol` that was used was manually extracted from an infected executable in the *VX Heavens* archive [15].

The transformation rules of `W32.Evol` were manually extracted [12]. Then they were abstracted and collected into a probabilistic transformation model as described in Section 2. Hand analysis of the `W32.Evol` code revealed that its engine chooses to transform instructions that happen to be left hand sides of its transformation system by reading a string of three bits at some memory location randomly computed at run time and transforming if the bit string is `000`. We hence assigned the value .125 to each rule application probability P_i. Once the decision is made to transform, the corresponding right hand sides are equally likely to be chosen. We therefore assigned the value $1/|r_i^{i_j}|$ to each probability of use, $P_i^{i_j}$.

The probabilistic transformation model was used to construct a probabilistic generator of simulated descendants.

Buckets for Various Distance Ranges

gen	[0.,.05[	[.05,.10[	[.10,.15[	[.15,.20[	[.20,.25[	[.25,.30[	[.30,.35[	$\geq$.35
1	267	729	4	-	-	-	-	-
2	-	398	598	4	-	-	-	-
3	-	-	406	593	1	-	-	-
4	-	-	-	204	775	21	-	-
5	-	-	-	-	14	703	281	2

Table 1: Counts of samples falling in various distance ranges (per generation)

5,000 simulated descendants were created using this simulator; these consisted of 1,000 simulated samples from 5 different generations. Another 5,000 simulated "non-variant" code segments were also generated. These segments were made up exclusively of those instructions that are processable by the engine. They were generated in order to challenge the distribution-based predictor; completely random simulated programs are unlikely to contain many of the instruction forms from the predicted distribution, and even ordinary Windows executables may not contain many instruction forms from the predicted distribution since the compilers may not generate them.

8,421 benign executable and DLL files were selected from a typical and clean Windows XP Pro installation. The were selected by collecting all such files together and keeping only those that disassemble using the `objdump` program from GNU binutils.

4.2 Apparatus

A small program was written to extract out the actual instruction form distributions from executables; it disassembles the executables using `objdump` before abstracting the instructions and collecting the distribution of the resulting instruction forms.

A distance calculator was created that takes as input a predicted instruction form distribution and an actual distribution, and then outputs the distance between them, as outlined in Sections 3.1 and 3.2. Next, a classifier was constructed as per Section 3.2 such that, when given a threshold, predicted distribution, and actual distribution for a program P, classifies P as a variant or non-variant.

4.3 Protocol

Distances between simulated samples and the predicted distribution were recorded for each of the 5 generations. That is, for generation 1, the distance between $pd_1(A)$ and $ad(j)$ was recorded for the 1,000 simulated samples for that generation.

A distance threshold of .6 was selected for the classifier based on an examination of the distances observed in the above. The actual instruction form distributions for the 8,421 authentic benign files were extracted using the program described above. Then these actual distributions plus those of the 5,000 simulated descendants and 5,000 simulated benign files were submitted to the classifier. The classifier used the predicted distributions from generations 1 through 5 to classify these inputs, and the classification decisions were recorded. In addition, a simple receiver operating characteristic (ROC) graph [6] was constructed by varying the threshold from .05 to 1.0 in .05 increments.

4.4 Results

The first part of the study provides an indication of what distances can be expected in matching the averaged, generation-specific predicted distribution to actual distributions of variants. The results are reported in Table 1. Each row i of the table reports how many simulated descendants fell within given distance ranges. For example, for the third generation, 496 simulated third-generation variants had their actual distributions within $[.10, .15[$ of the predicted predicted distribution.

The ROC graph showed that two classes were cleanly separable using the distance measure. That is, when the threshold was lower than the lowest non-variant distance, then precision and recall were perfect, except in those cases where the threshold was also less than the maximum distance for the variants, in which case recall was less than perfect. This is expected, as the lowest non-variant distance recorded was .946, which is greater than the maximum distance recorded in Table 1.

4.5 Discussion

The study is limited in a number of important ways. It does not measure the effects of host programs in the specific case of parasitic file-infecting viruses. Nonetheless, even though the study is of a single case, we expect that the `W32.Evol` case is fairly typical of a practically important class of metamorphic engines. The perfect classification scores and low distances measured still provide evidence of the basic feasibility of the statistical signature-based approach.

The source of the classification power in distinguishing descendants from non-descendants may simply be that the repertoire of instruction forms for `W32.Evol` are highly restricted and, thus, not substantially similar to the distributions one would find in benign files—or perhaps even other malware. This information may be more discriminating than using simply bytes or operations (without operands), which has been common in other statistical treatments in the field [1, 13]. For example, as Bilar [1] showed, the distributions of the operations alone are unlikely to be distinguishing enough. Instruction forms, however, carry more specific information, and a typical metamorphic engine may be expected to generate only a restricted subset of these. The study does not shed light on whether the instruction form distributions provide a strong enough signal to reliably detect in executables with file-parasitic viruses infecting an executable. If it does not, it may not be necessary to actually separate the parasitic code from its host before detection. If this cannot be done, the technique may need to be restricted to only the non-parasitic metamorphic malware, such as stand-alone Trojans, downloaders, etc.

Table 1 indicates that the heuristic of averaging the distributions may be reasonable since the distance between actual and average tends to be small even for several generations. In addition, the table indicates that, for this particular metamorphic engine, we may expect the classification accuracy to drop as the number of generations increase. This may be an artifact of `W32.Evol`'s particular engine, as it is "divergent" in the sense that it contains several instruction substitutions that replace single instructions with multiple instructions. It may also be in part due to the statistical nature of the test. For a generation $i > 1$, the range of possible instruction form distributions in the succeeding generation may be more "spread out" in the sense that the distances between actual distributions and the averaged distribution may be greater. To wit, the distance of first generation descendants to their predicted average may be the smallest because the average distribution is predicted on input a relatively smaller number of modifications of the archetype. This noted, for the study the loss is nevertheless not too severe that the classifications could not be practically useful, as `W32.Evol` grows too quickly between generations to permit variants with long descendant histories to be seen in the wild.

5. CONCLUSIONS AND OPEN ISSUES

The paper defines an approach for constructing classifiers that could be used to rapidly decide whether a suspect program is likely to be a variant of a metamorphic program, and thus worth spending time on more costly—and presumably more precise—analysis. The classifier is "engine-aware" in the sense that the predicted instruction form distributions it uses are generated using knowledge of a particular metamorphic engine's transformation rules and their attached probabilities. A small case study on a representative metamorphic virus provides a proof-of-concept test that the statistical signatures may be feasible and effective in practice. The fact that the classes were so cleanly separated in the study, in fact, gives us reason to wonder whether if the approach, by itself, may function adequately as a detector, in some instances, instead of being restricted to serving as a cheap filter to shield more costly analyses.

The approach and study also leave a list of open issues. The study is limited in scope, so it is difficult to know which metamorphic engines will create fundamental problems, i.e., what the defeats are. One possible defeat for the classifier is to ensure the instruction form distributions match large numbers of benign files. This would likely require that the repertoire of the engine mimic classic instruction form profiles to a much higher degree than `W32.Evol` does. This places constraints on the metamorphic engine designer, and it would be advantageous to know just how strong these constraints are. In those cases, in order to avoid too many false negatives the filter threshold may need to be higher. It is an open question as to how frequently the threshold would need to be set so high that the false positive rate renders this quick test approach ineffective as a filter for the more costly techniques.

Another limitation of the approach, as it is presented, is that a much needs to be known about the metamorphic engine (and archetype) in order to make it work. Apart from the transformations themselves, the instruction selection methods be known. That is, one must know how to disassemble the suspect programs so as to construct the actual instruction form distribution. If the transformation system can be assumed to be modeled, then it is perhaps not any additional stretch to assume the disassembly technique is also duplicated. It is an open issue, though, as to how prevalent and serious a concern this would be in practice.

It has been argued that engine-aware approaches for metamorphic engines may be feasible in practice since the number of known metamorphic engines is considerably smaller than the number of actual malicious programs, and because they evolve relatively slowly [16]. Thus, although modeling metamorphic engines is an expense, engine-aware approaches may still scale as well as other techniques such as signature-based methods, since they involve analyzing each malicious program individually rather than the metamorphic engines as a class. Nonetheless, the method outlined in the paper may not be the only feasible way to generate statistical signatures. Specifically, the signatures are averages of the predicted instruction form distributions. Appealing to standard sampling theory, the true average distribution should be rapidly approximated by collecting sample variants and measuring their instruction form distributions. That is, if sufficient numbers of metamorphic variants can be collected, then a mechanical process may be used to extract the necessary statistical signature. This is perhaps not an unlikely possibility in the case of an anti-virus company responding to an outbreak involving possibly hundreds of collected samples. Whether this approach would be feasible (particularly for parasitics) is still an open question.

Acknowledgements

Thanks to Rachit Mathur for extracting the transformations and probabilistic methods of `W32.Evol`. Funding for this work was provided in part by the Louisiana IT Initiative.

6. REFERENCES

[1] D. Bilar. Statistical structures: Fingerprinting malware for classification and analysis. In *Blak Hat*. Black Hat, 2006.

[2] D. Bruschi, L. Martignoni, and M. Monga. Using code normalization for fighting self-mutating malware. In *Proceedings of International Symposium on Secure Software Engineering*. IEEE, March, 2006.

[3] M. R. Chouchane and A. Lakhotia. Using engine signature to detect metamorphic malware. In *4th Workshop on Recurring Malcode (WORM)*, 2006.

[4] 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)*, pages 32–46, 2005.

[5] M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, and H. Veith. Malware normalization. Technical report, Department of Computer Science, The University of Wisconsin, 2005.

[6] T. Fawcett. ROC graphs: Notes and practical considerations for researchers. Technical report, HP Laboratories, Palo Alto, U.S.A., Jan. 2003.

[7] I. Krsul and E. Spafford. Authorship analysis: Identifying the author of a program. *Computers and Security*, 16(3):233–257, 1997.

[8] 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.

[9] A. Lakhotia, A. Kapoor, and E. U. Kumar. Are metamorphic viruses really invincible? - partI. *Virus Bulletin*, pages 5–7, December 2004 2004.

[10] A. Lakhotia and M. Mohammed. Imposing order on program statements to assist anti-virus scanners. In *Proceedings of the 11th Working Conferenceon Reverse Engineering*, 2004.

[11] A. Lakhotia and P. K. Singh. Challenges in getting 'formal' with viruses. *Virus Bulletin*, pages 15–19, September 2003 2003.

[12] R. Mathur. Normalizing metamorphic malware using term-rewriting. Master's thesis, Center for Advanced Computer Studies, University of Louisiana at Lafayette, Dec. 2006.

[13] S. J. Stolfo, K. Wang, and W.-J. Li. Towards stealthy malware detection. In *Malware Detection*, volume 27 of *Advances in Information Security*. Springer, 2007.

[14] P. Ször. *The Art of Computer Virus Research and Defense*. Symantec Press. Addison Wesley Professional, 1st edition, 2005.

[15] VX heavens. `vx.netlux.org`.

[16] A. Walenstein, R. Mathur, M. R. Chouchane, and A. Lakhotia. Normalizing metamorphic malware using term-rewriting. In *6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, 2006.

[17] A. Walenstein, R. Mathur, M. R. Chouchane, and A. Lakhotia. The design space of metamorphic malware. In *2nd International Conference on i-Warfare and Security (ICIW)*, 2007.