Using Markov Chains to Filter Machine-morphed Variants of Malicious Programs

Mohamed R. Chouchane, Andrew Walenstein, and Arun Lakhotia
Center for Advanced Computer Studies
University of Louisiana at Lafayette
mrc@ieee.org, walenste@ieee.org, arun@louisiana.edu

Abstract

Of the enormous quantity of malicious programs seen in the wild, most are variations of previously seen programs. Automated program transformation tools—i.e., code morphers—are one of the ways of making such variants in volume. This paper proposes a Markov chain-based framework for fast, approximate detection of variants of known morphers wherein every morphing operation independently and predictably alters quickly-checked global program properties. Specifically, identities from Markov chain theory are applied to approximately determine whether a given program may be a variant created from some given previous program, or whether it definitely is not. The framework is used to define a method for finding telltale signs of the use of closed-world, instruction-substituting transformers within the frequencies of instruction forms found in a program. This decision method may yield a fast technique to aid malware detection.

1. Introduction

The number of malicious programs found in circulation has been growing at a tremendous rate [21]. This growth is causing many troubles, not only because of sheer volume [23], but because it is difficult to produce a defensive response that is both rapid enough and comprehensive enough to catch all the new threats.

The fact is, though, that the vast majority of the new malicious programs are not wholly novel programs: they are *variants* of previously seen programs. The variants can be created via a number of methods, including: hand modification, using kits or frameworks, using packers or encryptors, or using automated program transformers, i.e., using *morphers*. These morphers can be utilized on the malware writer's desktop, on distribution servers implementing so-called "server-side polymorphism" [20], and within self-replicating malware itself, which are commonly called "metamorphic" malware [22]. Considering the problems that variants can cause, the use of automated morphing is likely to remain desirable on the part of malware authors [19].

Both dynamic and static methods have been proposed for aiding detection of morphed variants of existing programs. For example, *normalization* has been proposed to remove obfuscating program variations [7, 16], and for "reversing" the effects of known morphers [25]. Even if these are effective, it can be expensive to apply powerful variation-detection methods, particularly on suspect programs for which they are not necessary. It would be beneficial to have a fast filter that can quickly determine whether a given input program need not be considered for more involved processing [4].

One idea for building such fast filters is to consider the morphers as "authors" and recognize their authorship [3]. This idea is to take advantage of cases where the morphing engines have *idiosyncratic* output due to their recognizable repertoire of morphing techniques. However even if the morphing engine is not particularly idiosyncratic, it may still be *predictable*. An interesting question, then, is if the predictability of a given morpher can be used to generate a fast filter that can quickly discard programs that are definitely not a variant produced by it. It has been suggested that Markov chain theory, for example, might be used to define a framework for doing such detections [5].

This paper shows for certain classes of program morphers, Markov chain theory can be used to formalize an approach for recognizing whether a given program may be a morphed variant, or is not. In particular, the approach applies to morphers that satisfy specific properties including: independence of morphing operations, and the predictable alteration of a rapidly-checkable program property that drives further morphing. A key to the framework is that the program property must make it possible to predict the effects of morphing actions on

that property. Filtering is done by encoding the property changes as a transition matrix and then using Markov identities to test, to some fixed generation, whether or not a given program is a descendant of some known ancestor. Because of the Markov properties, the approach produces no false negatives. However, in some cases an exact solution is expected to be too expensive to compute exactly, and well-known approximation algorithms can be used instead. We introduce the framework using a particular example of its definition for detecting signs of closed-world, instruction-substituting morphers within the frequencies of instructions within programs.

Section 2 motivates the solution and reviews previous attempts to use global properties of instruction frequencies for fast filtering. Section 3 introduces the Markov-based decision framework; it is elaborated in Section 3.2 it by way of the specific example of an instruction frequency-based filtering approach. Section 4 introduces generalizations of the framework and discusses methods for reducing the space needed for the transition tables.

2. Fast filtering of morphed variants

It is known that the problem of recognizing families of related programs created by a grammar-based generator can be modeled formally as a language recognition problem [10, 12]. A morpher, upon input of a specific ancestor program, generates a *language* or "viral set" [8] consisting of all the different variants it can produce. Depending upon the morpher's qualities, as Filiol showed [10], the recognition problem can be classified into one of several categories based on Chomsky's language hierarchy. Even if the morpher is relatively simple linguistically, however, it is an important practical problem of knowing how to create an efficient recognizer.

One approach for recognizing the viral set is to recognize *commonalities* in either form, behaviour, or semantics. This approach relies upon the morpher not modifying the commonalities being sought. For example, one might look for common patterns of byte or operation sequences (e.g., Karim *et. al* [13]), or of calls or control flow (e.g., Zhang *et. al* [26]), or of semantic patterns (e.g., Christodorescu *et. al* [6]). A limitation of all such approaches is that the morpher may serve to remove those commonalities that are being sought, a goal known to morpher authors [9]. Another way of saying this is that the viral set may not uniformly contain a pattern of commonality of the type being considered.

A second general approach to such recognition is to *normalize* the input programs with the aim of making the modifications introduced by the morpher transparent and moot. If the specific morphing engine is known, then precise normalization engine may be possible—it may even be possible to generate the normalizers automatically from a model of the morpher [25]. If the morpher is not known, it may be possible to use general transformations that try to remove added variations [7] and impose order on unconstrained variations [16].

A third approach is to recognize the output of the morpher by its properties [3]. This is akin to authorship analysis for text [14] or programs [15], only instead of a human author one is considering that the morpher is a type of author modifying an original text. The central idea for authorship recognition is the detection of the idiosyncrasies of the author; in morphing terms, this amounts to searching for unique or limited repertoire of the morpher [4].

As pointed out by Chouchane *et. al* [4], no matter the efficacy of the detector for morphed variants, it may be too computationally inefficient to use on every suspect program. For example, a semantics-based approach such as that of Christodorescu *et. al* [6] involves heavyweight program analysis. But in many cases such deep analysis is unnecessary, as many of the samples under consideration (in a system scan or within network traffic) are not, in fact, going to be malicious morphed variants that require the heavyweight detection. An approximate but fast filter may therefore be extremely useful so that the more resource-hungry approaches are used only infrequently—ideally, only when needed.

Chouchane *et. al* [4] proposed a heuristic method for such a filter for metamorphic programs. It is based on measuring statistical properties of the instruction forms of the program. Specifically, the filtering was based on the *instruction form frequencies*. The different generations of a metamorphic program were characterized (approximately) by averaging the frequencies of their instruction forms. A suspect program is then heuristically filtered if the difference between its instruction frequency vector (IFV) and all averaged vectors is higher than some threshold.

A problem with such past approaches is that, even if they are fast, they are heuristic in the sense that they could guarantee neither false positive nor false negative rates. A key reason is the heuristic nature of the test. But a secondary reason is that it relies on finding the idiosyncrasies of the morphing engine. While the data from Chouchane *et. al* [4] suggests that instruction *form* frequencies may indeed be more indicative of the morpher than simple instructions [2] or bytes [1], it still may be the case that the morpher is altered so that it is not easy to statistically distinguish it from other legitimate constructors.

For a fast filter, false positives are perfectly acceptable if the false positive rate is reasonable—even a 20% false positive rate means that 80% the time no costly analysis is wasted on irrelevant suspect programs. How-

ever it is particularly important to have an extremely low false negative rate—guaranteed no false negatives, if possible. In particular, it would be helpful to have a general mathematical framework within which one can prove the false negative rates are desirably low.

3. Fast recognition using Markov models

A framework is proposed for using Markov theory to define fast filters based on detecting telltale signs of the morphing within program properties that are quickly checkable. Using Markov chain theory it is possible to show that false negatives will not occur up to some generation of morphing. The model is defined on morphing processes that: (1) satisfy the Markov property, and (2) are such that each morphing operations alters a rapidly-checkable program property in predictable ways, and such that this property is a determinant of the frequencies of morphing actions in subsequent iterations.

The framework is introduced by way of an example filter model for probabilistic, closed-world, instruction-substitution engines. The example uses IFVs as the comparison property. After describing the specific example, its generalization is described. Introducing the framework using this example helps in exposition, but it also demonstrates that the framework can be applied to an interesting class of morphers, since such instruction-substitution engines are known. Background notation is briefly introduced, a model of the morpher is introduced, and the Markov-based filter is described. The Markov-based analysis is then generalized to a framework.

3.1. Morpher modeling

In this section, a formal model of a target morphing engine M is presented. It is based on the model from Chouchane *et. al* [4]. We target the class of closed-world morphing malware that uses a morphing engine that applies a fixed, finite set of transformation rules, each rule mapping an instruction (the left hand side) to a sequence of multiple instructions (the right hand side). These rules are used by the engine to probabilistically substitute, in the variant being transformed, occurrences of the left hand sides of the rules with one of their corresponding right hand sides. These probabilities are assumed to be fixed, or exactly learnable from the description of the engine, for each rule of the transformation system. In order to reduce the very large number of possible assembly language instructions which are identical except for the register names or variable values that they use, the model may abstract an instruction to just its opcode mnemonic. In the sequel we will loosely use the phrase "code segment" to refer to the abstraction of the actual code segment.

Given a positive integer n and a sequence $\vec{x} = (x_1, x_2, ..., x_n) \in \mathbb{N}^n$, we define the norms $||\vec{x}||_1 = \sum_{i=1}^{n} |x_i|$ and $||x||_\infty = max(|x_1|, |x_2|, ..., |x_n|)$. $\mathcal{I} = \{I_1, I_2, ..., I_m\}$ denotes the instruction set of a target computing platform. OCC denotes the function from $(\mathcal{I}^+, \mathcal{I})$ to $\mathbb{N}$ mapping each (P, I_i) pair to the frequency of instruction I_i in code segment P. Z denotes the function from $\mathbb{R}$ to $\{0, 1\}$ which returns 1 if and only if its argument is 0. $T = \{l_i \rightarrow \{(\Pr_i^j, r_i^j) : 1 \leq j \leq i_{max})\}\}$ denotes the set of n productions used by the morphing engine to transform its input. It is assumed that $l_i \in \mathcal{I}$, $r_i^j \in \mathcal{I}^+$, exactly one r_i^j is identical to l_i, and $\sum_{j=1}^{i_{max}} \Pr_i^j = 1$, for all $1 \leq i \leq n$. $\Pr_i^j$ denotes the probability of use of right hand side element r_i^j (i.e., the probability that r_i^j is chosen to replace an instance of l_i). For instructions I_i and I_k, $F_{i,k}(\beta)$ returns the probability that, on input one instance of I_i, the engine generates β instances of I_k.

The probabilities of use are assumed to be *fixed* for each production and extractable interactively from the morphing engine. Furthermore, if the engine is not available, these probabilities may also be *estimated* from large corpora of programs, where each corpus contains members of some specific generation of descendants of some *Eve*. Probabilities of use may for example be implemented (as in the `W32.Evol` and `W32.Simile` metamorphic viruses) using a random number generating procedure that is part of the engine and that makes its choices at run time by reading arbitrary memory locations. If the latter is the case then we assume that the choices are uniform; $\Pr_i^j = \frac{1}{i_{max}}$ for each r_i^j. Whenever the morphing engine visits an instruction that is also the left hand side l_i of some production, with probability $\Pr_i^j$ the engine substitutes l_i for r_i^j. A sample set of this type of rule appears in Figure 1.

Let P denote some program and n the number of distinct instructions occurring in P. The instruction frequency vector of P, denoted $IFV(P)$, is the n-tuple each of whose components represents exactly one opcode mnemonic and holds the frequency (or count) of that mnemonic in P. No two components may represent the same mnemonic. For instructions I_i and I_k, $\alpha \in \mathbb{N}$, and $\beta \in \mathbb{N}$, $G_{i,k}(\alpha, \beta)$ returns the probability that, on input α instances of I_i, the engine generates β instances of I_k. It returns 0 if $\alpha = 0$. For $1 \leq i \leq k \leq m$, $\vec{\alpha}_{i,k}$ denotes the segment (subvector) of a program's IFV which starts at instruction I_i and ends at instruction I_k. For $\beta \in \mathbb{N}$, $1 \leq i \leq j \leq m$, and $1 \leq k \leq m$, $H_k(\vec{\alpha}_{i,j}, \beta)$ returns the probability that β instances of instruction I_k are generated by the engine on input a program with IFV $\vec{\alpha}_{1,m}$, where all of $\vec{\alpha}_{1,m}$'s components are 0 except, perhaps, for subvector $\vec{\alpha}_{i,j}$. Finally, $T(\vec{\alpha}_{1,m}, \vec{\beta}_{1,m})$ returns the probability that, on input a program with IFV $\vec{\alpha}_{1,m}$,

	l_i	$\rightarrow$	$\{r_i^{i1}$	r_i^{i2}	$r_i^{i3}\}$

1	`mov [reg1+imm], reg2`	$\rightarrow$	`push reg`
			`mov reg, imm`
			`mov [reg1+reg], reg2`
			`pop reg`
				`push reg`
				`mov reg, reg1`
				`add reg,imm1`
				`mov [reg+imm2], reg2`
				`pop reg`	

2	`mov reg, imm`	$\rightarrow$	`mov reg, imm1` `add reg, imm2`	`mov reg, imm1` `sub reg, imm2`	`mov reg, imm1` `xor reg, imm2`
3	`push reg`	$\rightarrow$	`push reg` `mov reg, imm`		
4	`sub reg reg`	$\rightarrow$	`xor reg, reg`		

Figure 1. Example rule set.

the probabilistic instruction substituting engine generates a program with IFV $\vec{\beta}_{1,m}$.

Note that, in general, a morpher may iteratively apply the morphing rules in order to create a variant. A server-side morpher may, for example, start with the same original program and create variants by multiple application of the substitution rules. In such cases, we still say that the morphing engine produces variants of different "generations", where a generation is an iteration through the substitution rules.

3.2. Detection using a Malware's IFV

Given the preceding model of the instruction-substituting morpher, a property is selected that is predictably altered by the morpher and such that this property dictates iterated morphing properties. We expand on the suggestion of Chouchane *et. al* [5] by elaborating the formal Markov model for the detection or filtering; generalization of the model to a framework is performed in the subsequent section. The trick to mapping the problem to a Markov chain problem is to treat that property as a "state", and then map state transition changes as predictable changes to that property.

State and State Transition. To be consistent with the terminology used in Markov chain theory, we will use the term "state" (normally called abstraction) to refer to a program's IFV. Figure 2 illustrates a simple example of a state transition induced on a code fragment by some probabilistic instruction substituting engine. Note that a state transition is a vector (IFV) transformation.

State Transition Probability. Given two program states $\vec{\alpha}_{1,m}$ and $\vec{\beta}_{1,m}$, the transition probability from $\vec{\alpha}_{1,m}$ to $\vec{\beta}_{1,m}$ is the probability that, on input a program whose state is $\vec{\alpha}_{1,m}$, the morphing engine produces a program whose state is $\vec{\beta}_{1,m}$. This probability will depend upon the instruction frequencies, the transition rules that can morph those instructions, and the probabilities of those rules. Since the transition probabilities do not depend upon the history of transitions, such a morphing process obeys the Markov property and can be modeled as a Markov chain.

State Transition Matrix. Given a (finite) set of states of size k, the k by k transition matrix is the matrix constructed where row i and column j entry is the state transition probability between state i and state j.

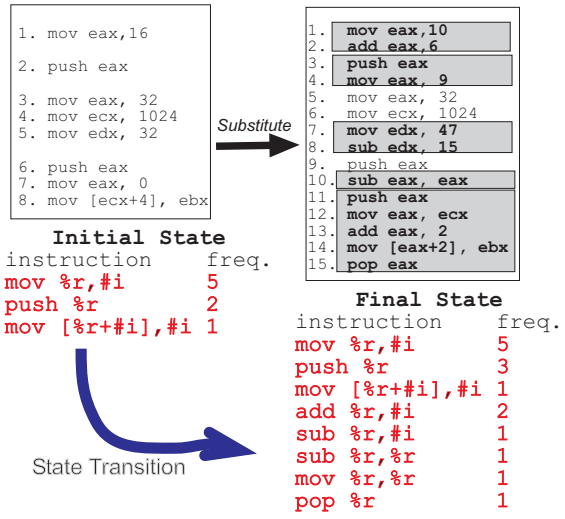

Figure 2. Simple state transition.

The Existing work on Markov chains [18] has identified certain interesting classes of chains and ways of using a chain's transition matrix to infer useful information about the process it represents. In particular, successive powers of the transition matrix can be used for predicting the distribution of instruction forms. Specifically, a Markov chain T is typically started in a state chosen by a probability distribution on the set of states, called a *probability vector*. Let $\vec{u}$ denote a probability vector which holds the initial probabilities of a malware's state. The powers of T are known to give interesting information about the evolution of these distributions from one malware generation to the next: For any positive integer n, the ij^{th} entry $(T^n)_{ij}$ of T^n gives the probability that the chain, starting in state s_i, will be in state s_j after n

steps. More generally, if we let $u^n = uT^n$, then the probability that an n^{th}-generation malware descendant is in some state s_i after n transitions is the i^{th} component of u^n.

This result can be used as the basis for filtering. Assume that a known existing malicious program is suspected to have given rise to variants through application of the morpher. The original malicious program *eve* can be considered the initial probabilities, that is, $\vec{u}$ will have 0 for all states except for $IFV(eve)$, the IFV of *eve*. Given a suspect program q, it will have a particular IFV, $IFV(q)$. The matrix entry of T^n $IFV(eve), IFV(q)$ will record the probability that q is a nth generation variant of eve. Significantly, if that entry is 0, then it is impossible that q could have been generated by the morpher in n generations. If the entry is non-zero, it the probability that the morpher would have created q from the eve. Given a constant number of generations, k, the transition matrices up to k define the possibility that q could have been generated by M in up to k generations; the probabilities can also be examined to determine if it is likely that q is a given generation descendant of eve. This basic results provides the mathematical basis for a filter based on the predictable morphing actions of M.

3.3. Computing transition probabilities

The state transition probability, $T(\vec{\alpha}_{1,m}, \vec{\beta}_{1,m})$, is computed using the probabilities $F_{i,k}$, $G_{i,k}$, and H_k defined above. $F_{i,k}$ can be computed by observing that, for all β,

$$F_{i,k}(\beta) = \sum_{j=0}^{i_{max}} Z(\beta - OCC(r_i^j, I_k)) \times \Pr_i^j.$$

We view the probabilistic instruction substitution process of α instances of instruction I_i as α independent events (individual substitutions of each of the α instances of instruction I_i). The outcome of each of these events may yield zero or more occurrences of instruction I_k. Let $S = \{\delta : F_{i,k}(\delta) \neq 0\}$ denote the set of all possible counts of instruction I_k that can be generated by the engine on input an instance of instruction I_i. Let S_β^α denote the set of α-tuples $\vec{\delta} = (\delta_1, \delta_2, ..., \delta_\alpha)$ of elements of S such that $||\vec{\delta}||_1 = \beta$. $G_{i,k}(\alpha, \beta)$ is given by

$$G_{i,k}(\alpha, \beta) = \sum_{\vec{\delta} \in S_\beta^\alpha} \prod_{j=1}^{\alpha} F_{i,k}(\delta_j).$$

Unfortunately, the problem of finding a $\vec{\delta} \in S^\alpha$ such that $||\vec{\delta}||_1$ is equal to β is an NP-complete one. In fact, computing any α-tuple x whose $||.||_1$ equals a fixed β is an instance of the SUBSET_SUM problem and is hence NP-complete. In practice, one may then want to choose to use a polynomial-time approximation scheme [11] for computing each of the $G_{i,k}(\alpha, \beta)$.

$H_k(\vec{\alpha}_{1,m}, \beta)$ can be recursively computed by observing that for $1 < i < m$, $H_k(\vec{\alpha}_{i,m}, \beta) = \sum_{\delta=0}^{\beta} F_{i,k}(\delta) \times H_k(\vec{\alpha}_{i+1,m}, \beta - \delta)$. The recursion stops when $i+1 = m$, that, is when we ask for the value of $H_k(\vec{\alpha}_{m,m}, \beta - \delta)$. This value is computed by observing that $H_k(\vec{\alpha}_{m,m}, \beta - \delta) = G_{m,k}(||\vec{\alpha}_{m,m}||_1, \beta - \delta)$.

The IFV transition probabilities are finally computed by observing that

$$T(\vec{\alpha}_{1,m}, \vec{\beta}_{1,m}) = \prod_{i=1}^{m} H(\vec{\alpha}_{1,m}, ||\vec{\beta}_{i,i}||_1).$$

3.4. Matrix optimization

In theory, the set of all possible IFVs is infinite, since infinite growth in size is a theoretical option for morphing malware. It is hence necessary in practice to at least impose an upper bound on the size of this set while limiting, as much as possible, the deterioration of the predictive and classification power of the transition matrix.

Possible heuristics for doing so include, but may not be limited to, (1) reducing the size of the instruction set by abstracting an assembly language instruction to its opcode mnemonic or by ignoring register names and variable values, (2) imposing an upper bound on the possible frequency of each individual instruction, (3) imposing an upper bound on an IFV's norm $||.||_\infty$, and (4) abstracting each component of an IFV to one of two values ("low" or "high"), depending on whether that component is less than or greater than some threshold.

4. Generalization

The IFV-based model described above can be generalized to expand the set of morphers that the model can apply to. Furthermore, the basic Markov-based framework can be divorced from the specifics of the IFV so that it can be applied to the larger class of morphers that need not be instruction-substituting; indeed, the same basic Markov framework can apply to any morphing process that fits certain Markov process properties.

4.1. Generalization of mutation model

In addition to allowing garbage inserting transformations which are modelable as fixed transformation rules (i.e., which map an instruction to a code segment of finite, fixed size), we allow the following relaxations to the target probabilistic instruction-substituting engine.

1. *The engine may not perform any condition checking.* The engine is free to rewrite any or all of the left hand side instances without the need to perform condition checking. Rule 3 of Figure 1 is garbage-inserting rule and is a good example of the engine simply making the *assumption* that the condition "immediately after each `push eax` instruction, the contents of register `eax` may be safely altered" is `TRUE`. This relaxation is based on the observation that condition checking may not be an optimal design choice of morphing malware since it typically increases the engine's transformation time; it may require the engine to statically solve potentially costly (and sometimes unsolvable) problems. Allowing this relaxation adds non-condition-checking malware to our targeted class of malware. See Walenstein *et al.* [24] for more on the design choices of morphing malware.

2. *The engine may perform additional transformations as long as they are not IFV-altering.* The engine may perform extra transformations to the variant being transformed as long as they do not change the IFV of that variant. Such transformations include register renaming and instruction reordering. Examples are shown in Figure 3. These transformations (except perhaps for the instruction reordering transformation) are often used by malware writers since they are relatively easy to implement and contribute to changing the code of the variant being transformed. Allowing this relaxation adds non-IFV-altering malware to our targeted class of malware.

The first relaxation is reasonable because the malware signature prediction and detection procedure proposed in this paper is independent of the semantics of the malware.

The second relaxation is reasonable because this procedure is specifically designed to detect the effects of the instruction substitution transformation and assumes that only instruction substitution affects the IFV of the malware code, i.e., any other subsequent transformations performed by the engine do not affect the IFV of the malware code.

Since we are dealing with engines which substitute an instruction by code segments of size greater than or equal to one, the size of a descendant of a given variant may be greater than that of the variant. It is clearly a plausible requirement of good design practices of morphing malware to prevent a malware variant size from "growing too fast" as its code gets morphed. A number of methods can be used to have some control over the increase in the sizes of the successive descendants for a given variant. One way of doing this, in the context of probabilistic instruction-substituting malware, is to assign low rule application probabilities to rules that increase the size of the variant being transformed. Higher rule application probabilities would be assigned to rules transforming a single instruction into another single instruction. The `IA-32` instruction set is rich enough to allow for a large number of rules (such as the fourth rule in Figure 1) mapping an instruction to an equivalent but syntactically different instruction.

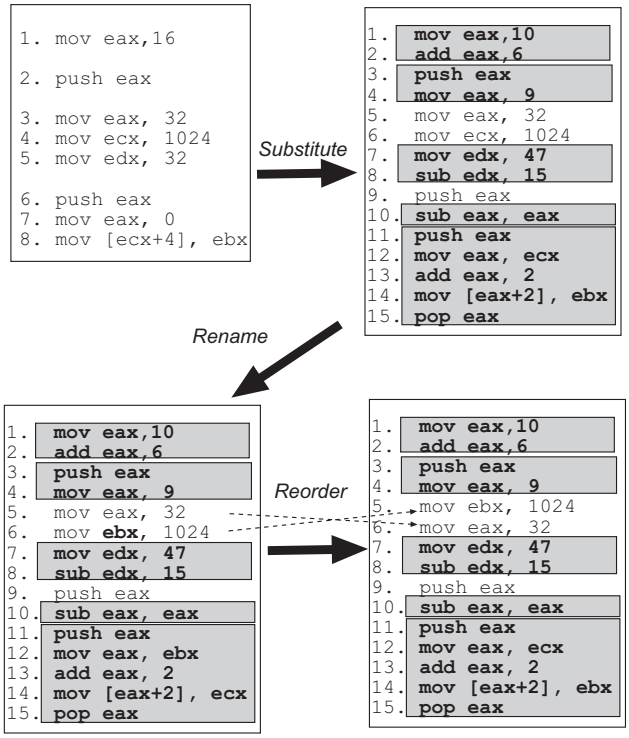

Figure 3. Morphing transformations.

4.2. Generalization to Markov framework

The example used in the previous section is that of an instruction-substituting morphing engine. It is possible to generalize the entire work by carefully noting how the formalization to use Markov chain theory was accomplished:

- **Markov property** The morphing engine obeyed the Markov property in that the probability of the rules firing did not depend upon the history of firing.

- **State and state transition mapping** The state in our model was the IFV of the program. This is a quickly calculable abstraction of the program itself. Morphing actions (state transitions) could be mapped to state transitions.

- **Morphing dependence on property** Note that for the transition mappings to be possible, the morphing actions must depend upon the abstracted state. In the case of the instruction-substituting engines, it is clear that the substitutions that may be performed depend wholly upon the frequencies of the instructions present.

Thus, while the proposed IFV method of fast filtering itself may be useful (for instruction substituting morphers), a more general result is the Markov-based framework. It can be noted that many other morphing processes—including ones that are *not* fully automatic—may be approximated as a Markov process. If there is a controlling property of the morphing, and this property is quickly checkable, then a fast filter may be defined.

5. Conclusions

The IFV-based detection model proposed in the paper circumvents the inherent limitations of many static and dynamic malicious program detection approaches. It is a semantics-independent solution to the problem of detecting morphed malware variants that overcomes the limitations of having to exactly solve unsolvable problems or efficiently solve provably intractable problems. Furthermore, the individual limitations of existing technology and research on malware detection, and the fact that morphing engines are difficult to write and tend to be reused, typically by writing a new malicious program to be morphed by an existing engine, suggest the usefulness of the proposed method, since it uses information about a morphing malware's engine (transformation process) to detect variants of the malware by approximately deciding membership in the set of programs that can be generated by the engine.

It is perhaps intuitively expected that a predictable morphing engine will have an exploitable weakness simply by virtue of the predictable nature of the morphing actions. The proposed framework defines cases where fast filtering is possible; specifically the framework provides the mathematical basis for defining accurate filters (no false positives) for variants of known programs built from known morphing engines that obey specific Markov properties. This can raise the bar for malware production by easing the detection of the potentially large volumes of variations that could be generated using simple morphers. While important implementation and optimization issues are outside the scope of this paper, the key step for defining a fast filter is taken by the framework in the use of a quickly computable property of a program that can be used to look up probabilities of being constructed by the Markov generation process.

References

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

[2] D. Bilar. Opcodes as predictor for malware. *Int. J. Electronic Security and Digital Forensics*, 1(2):156–168, 2007.

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

[4] M. R. Chouchane, A. Walenstein, and A. Lakhotia. Statistical signatures for fast filtering of instruction-substituting metamorphic malware. In *5th Workshop on Recurring Malcode (WORM)*, 2007.

[5] M. R. Chouchane, A. Walenstein, and A. Lakhotia. Metamorphic authorship recognition using Markov models. *Virus Bulletin*, May 2008.

[6] 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.

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

[8] F. Cohen. Computational aspects of computer viruses. *Computers and Security*, 8(4):325, June 1989.

[9] M. Driller. Metamorphism in practice, 2004. `http://vx.netlux.org/29a/29a-6/29a-6.205`.

[10] E. Filiol. Metamorphism, formal grammars and undecidable code mutation. *International Journal of Computer Science*, 2(1):70–75, Apr. 2007.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[12] Z. hong Zuo, Q. xin Zhu, and M. tian Zhou. On the time complexity of computer viruses. *IEEE Transactions on Information Theory*, 51(8), Aug. 2005.

[13] M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida. Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1(1-2):13–23, November 2005.

[14] D. V. Khmelev and F. J. Tweedie. Using markov chains for identification of writers. *Literary and Linguistic Computing*, 16(4):299–307, 2001.

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

[16] 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.

[17] H. Martin, editor. *Proceedings of Virus Bulletin Conference 2007*, Vienna, Austria, 2007. Virus Bulletin Inc.

[18] S. Meyn and R. Tweedie. *Markov Chains and Stochastic Stability*. Springer-Verlag, London, 1993.

[19] M. Schipka. A road to big money: evolution of automation methods in malware development. In Martin [17].

[20] R. Sherstobitoff. Server-side polymorphism: Crimeware as a service model (CaaS). *ISSA Journal*, page 46, May 2008.

[21] Symantec. Symantec global internet security threat report volume XIII: Trends for july–december 07, Apr. 2008.

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

[23] J. Telafici and D. Gryzanov. What a waste — the AV community DoS-ing itself. In Martin [17].

[24] 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.

[25] A. Walenstein, R. Mathur, M. R. Chouchane, and A. Lakhotia. Constructing malware normalizers using term rewriting. *Journal in Computer Virology*, January 2008 2008. doi:10.1007/s11416-008-0081-5.

[26] Q. Zhang and D. S. Reeves. MetaAware: Identifying metamorphic malware. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC'07)*, pages 411–420, 2007.