J Comput Virol Hack Tech (2013) 9:137–157
DOI 10.1007/s11416-013-0183-6

ORIGINAL PAPER

Detecting machine-morphed malware variants via engine attribution

Radhouane Chouchane · Natalia Stakhanova · Andrew Walenstein · Arun Lakhotia

Received: 7 September 2012 / Accepted: 6 February 2013 / Published online: 5 March 2013
© Springer-Verlag France 2013

Abstract One method malware authors use to defeat detection of their programs is to use morphing engines to rapidly generate a large number of variants. Inspired by previous works in author attribution of natural language text, we investigate a problem of attributing a malware to a morphing engine. Specifically, we present the malware engine attribution problem and formally define its three variations: MVRP, DENSITY and GEN, that reflect the challenges malware analysts face nowadays. We design and implement heuristics to address these problems and show their effectiveness on a set of well-known malware morphing engines and a real-world malware collection reaching detection accuracies of 96 % and higher. Our experiments confirm the applicability of the proposed approach in practice and indicate that engine attribution may offer a viable enhancement of current defenses against malware.

1 Introduction

In 2008 Symantec added 1.6 million new malware signatures to its malware database [67,74]. This number increased to 2.9 million in 2009. In 2010, the new addition to the database constituted 4.4 million new signatures, which translated to around 22,000 signatures per working day. These staggering numbers reveal the complexity of the problem that the anti-malware vendors face nowadays. This unusually fast pace of malware writers is mostly attributed to the wide availability of malware toolkits that allow malware authors to rapidly produce large numbers of new malware variants through the use of advanced obfuscation techniques.

The two most common obfuscation techniques favored by the malware writers today are polymorphism and metamorphism. Both techniques change the form of malware while retaining the same functionality across all malware variants. While polymorphism hides the code through a self-decrypting behavior, metamorphism uses mutation techniques aiming to produce syntactically different instances of malware [68].

Both techniques present a significant challenge for the traditional signature-matching detection engines. The classic polymorphic malware follows a path of syntactic transformations (compression and encryption essentially change the statistical properties of malware binary) and thus is easily detectable by byte-level statistics-based approaches [45,70], and other syntactics-based techniques [71]. Several methods in polymorphic obfuscation (e.g., targeted blending attacks) were designed to escape such statistics-based approaches [15]. They however can be addressed by semantics-based detectors [11,35].

As opposed to polymorphism, metamorphic obfuscation presents a larger challenge to anti-virus detectors. Even a basic metamorphic malware can easily escape statistics-based detectors relying on syntactic features of malware samples [43]. As such metamorphic obfuscation requires methods capable of advanced analysis of program semantics. Such analysis calls for sophisticated techniques that often rely on formal methods to reason about the poten-

R. Chouchane
CCT 430, Columbus State University, 4225 University Avenue, Columbus, GA31907, USA
e-mail: chouchane@columbusstate.edu

N. Stakhanova (✉)
University of New Brunswick, Fredericton, Canada
e-mail: natalia.stakhanova@unb.ca

A. Walenstein · A. Lakhotia
University of Louisiana at Lafayette, Lafayette, LA, USA
e-mail: walenste@ieee.org

A. Lakhotia
e-mail: arun@louisiana.edu

tial malicious functionality of the code [3,44,62]. Unfortunately, most of these techniques are impractical either due to prohibitively high complexity of analysis or additional constraints on the environment.

In this work we propose to take an alternative approach to detection of machined morphed malware based on authorship attribution. Authorship attribution is a technique, well-established in social science, aiming to determine an author of a document given some textual characteristics of the author's writing style extracted from his previous works [65]. These characteristics, often called *stylistic discriminators*, uniquely identify an author, on the one hand remaining constant among all his works, while on the other hand, varying between the works of different authors [23]. Authorship attribution technique has been actively used for plagiarism detection [66], author verification and profiling [2,32] and in the recent year in biometric research [18] an source code authorship analysis [16,39].

In our study, we draw an analogy between an author and a morphing engine, i.e. a malware obfuscation toolkit. Given a collection of programs generated by a set of engines, the goal of engine attribution is to associate a new malware instance to the engine (author) that created it. The main idea behind our approach is that a morphing engine during an obfuscation process follows a certain algorithm to create meaningful strings belonging to a language. As such it should be possible to extract stylistic discriminators that would appear in every malware instance obfuscated by this engine.

If such discriminators are found then instead of trying to recognize an individual malware variant based on its specific characteristic, a whole family of malware can be identified by recognizing its engine. The benefit of such approach is clear, instead of maintaining one signature per malware variant, we could effectively use one signature that would uniquely characterize all malware variants generated by the same engine.

This approach aims to complement and simplify a triage stage of an existing automated malware analysis, usually performed by antivirus vendors to identify threats that warrant further analysis. Morphed malware that mutates with each propagation, could easily escape the initial hash based filtering of triage. However, matching such mutated sample to an array of available engine signatures would allow to quickly diagnose the threat without requiring expensive program analysis.

In this work we present a malware engine attribution problem and define three variations of this problem: two general cases of attributing a malware to an engine, denoted as the $MVRP$ and the $DENSITY$ problems and one special case of determining whether a new sample is a descendant of a known variant created by a morphing engine, the GEN problem. All three problems take an advantage of the recognizable repertoire of morphing techniques employed by engines that result in predictable features embedded in their output (i.e., morphed malware).

The first two problems: the $MVRP$ and the $DENSITY$, consider a situation when malware engines are known, and the main question is to recognize whether an instance on hand presents a threat by attributing it to corresponding engines. In this work we present two algorithms for generation of engine signatures: $n$gram instruction frequency based calculation ($MVRP$ problem) and clue-density based computation ($DENSITY$ problem). Both heuristics take advantage of statistical properties of the instruction forms of malware program. An engine signature serves as a benchmark for malicious behavior, and a suspicious program is then filtered if its statistical properties significantly deviate from the available engine signatures.

The latter problem GEN addresses the situation when the engines are unknown and only variants generated by some engine are available. In this case it is important to determine whether given variants present a threat and if so, to outline malicious behavior that can be expected from these instances. This situation may arise if a morphing engine is altered to the extend that statistically it becomes undistinguishable from other legitimate constructors. In this context, the GEN problem would allow to determine whether or not a given program is a morphed variant of a known malware which would allow to determine a sample's potential behavior and diagnose a problem. We formalize the GEN recognition problem using Markov chain theory and propose a heuristic that models program properties changes as a transition matrix and relies on Markov identities to test to some fixed generation, whether or not a given program is a descendant of a known malware variant.

The main contributions of the paper may be summarized as follows:

- We present a new method for detecting large number of machine morphed malware variants using static signatures. The method uses authorship attribution analysis to relate the malware variants generated by a morphing engine to their corresponding author, i.e. engine.
- We present and formally define the malware engine attribution problem and three of its variations: $MVRP$, $DENSITY$ and GEN.
- We design and implement heuristics for each of these detection problems. We evaluate the proposed solutions on a set of well-known malware morphing engines and a real-world malware collection and show detection accuracies of 96 % and higher.

The rest of the paper is organized as follows. Section 2 gives a general background into the problem and formal definitions of morphing malware, and the three detection problems. Sections 3, 4 and 5 propose and evaluate

solution approaches to the three variations of engine detection problems. Section 3 proposes and evaluates an *n*gram based solution to the problem of attributing, to their engine, any of the morphed malware instances that are known to have been generated by a fixed, closed-world engine, one whose transformation procedures and rules do not change overtime (e.g., by uploading new transformation rules to the engine). Section 4 proposes and evaluates a clue-density based solution to the problem of recognizing morphed malware instances which are known to have been generated by closed-world, code-substituting morphing engine. Section 5 proposes and evaluates a Markov-chain-based solution to the problem of not only determining whether a given malware instance has been generated by a known, closed-world morphing engine, but also to attribute the instance to a specific generation of descendants of a known variant. The state of the research in the area of malware detection is given in Section 6, and limitations of our work are discussed in Section 7. Section 8 concludes the paper.

2 Background

The use of various self-protection techniques (i.e. encryption, compression) has always been a desired approach for virus writers to evade antivirus detection and challenge human analysts. In the recent years these techniques have significantly evolved allowing for sophisticated self-protected and self-distributing malware.

Among these advanced techniques, oligomorphism, polymorphism and metamorphism have emerged as the most popular solutions for bypassing traditional malware detection. Aiming to preserve semantic equivalence of produced malware variants, these techniques manipulate syntactic characteristics of a code (e.g., encryption, byte/instruction reordering, junk instruction insertion) generating syntactically different instances of malware while retaining the same functionality across all variants.

Both oligomorphism and polymorphism hide malicious code through encryption that is dynamically reversed right before the code execution. Since the decrypting routine has to be carried alone an encrypted code and has to remain clear, it presents a major weakness that is easily picked by signature-based detectors. To disguise decryptions, oligomorphic (so called semi-polymorphic) and more advanced polymorphic malware use mutated decryptors that change in each generation of malware [68].

As opposed to this, metamorphism is a code obfuscation strategy that does not use encryption and thus does not require a decryptor. Metamorphic malware uses various code mutation techniques to disguise its code, each time generating instances syntactically different from one another [68]. Such mutation can be achieved through the transformations modifying either data flow (e.g., rewriting rules, junk insertion, permutation, registers exchange) or control flow (e.g., branch insertions) [6,68].

The rapid development of self-protected and self-distributing techniques has resulted in a number of off-the-shelf obfuscation engines and virus generator kits. Among the most popular engines are ADMmutate [26], CLET [13], MetaSploit engines [72]: Shikata Na Gai, Fnstenv Mov and Call4dWord, and virus generating kits: NGVCK [51], noted to be one of the most effective in creating highly metamorphic code [81], and VCL [75].

ADMmutate, CLET and MetaSploit engines are the examples of morphing engines employing both encryption (to hide malware attack code) and mutation techniques (to disguise decryptors). Although all engines use XOR encryption (or similar type, e.g. ROR), the complexity of the mutation varies from instructions re-ordering (e.g., ADMmutate [26]) to registers exchange (CLET) [13]. Among the MetaSploit engines, only Shikata Na Gai is fully polymorphic according to the Metasploit documentation [28].

Typically, polymorphic engines generate a ready-to-use raw attack code (often in assembly language), that is injected into memory by exploiting a buffer overflow vulnerability. As opposed to these engines, virus generating kits allow to assemble a stand-alone executable. A virus source code is generated from a number of available library components (including various propagation and infection methods) and then obfuscated using basic self-protection techniques (i.e., encryption, antidebugging) and mutation [68]. Two of such kits are VCL (Virus Creation Laboratory) and NGVCK (Next Generation Virus Creation Kit). NGVCK is an advanced version of VCL that in addition to encryption also supports mutation, i.e. every new virus variant created with the kit is automatically morphed so that no two viruses look the same [68].

Our focus in this work is primarily on morphing engines that apply mutation techniques to disguise malware code (either in a raw code form or as a stand alone executable). As such the detection of encrypted malware code is beyond the scope of this paper.

2.1 Morphing malware: definitions and notations

In this section we give a formal definition of morphing engine and morphing malware that we will rely on for the rest of the paper.

Since it is a simple yet powerful way of expressing nondeterminism and efficient computation, we use the Turing machine as an underlying mathematical model to define a morphing engine and malware. A nondeterministic Turing machine (NDTM)[1] is a 6-tuple $M = (S, \Sigma, \Gamma, \delta, s_0, h)$,

[1] A detailed definition of an NDTM and of a polynomial time NDTM can be found in [63].

where S is a finite set of states, $\Sigma = \{0, 1\}$ is the input alphabet, Γ is the tape alphabet, $s_0 \in S$ is the start state, $h \notin S$ is a halt state, and $\delta : S \times \Gamma \rightarrow 2^{(S \cup \{h\} \times \Gamma \times \{\leftarrow, \rightarrow\})}$ is the transition function, where $\leftarrow$ and $\rightarrow$ encode the directions in which the tape's head is to move.

Based on this definition, a morphine engine can be viewed as any engine (Turing machine) that transforms at least one sequence of input symbols on *any* run through *any* set of non deterministic choices. Formally,

Definition 1 (*Morphing Engine*) A polynomial-time NDTM M is said to be a *morphing engine* if there exist distinct $v, v' \in \Sigma^*$ such that $(\sqcup, s_0, v) \vdash^* (\sqcup, h, v')$, where $(\sqcup, s_0, v)$ and $(\sqcup, h, v')$ are possible configurations of M. Every such v is called an *M-friendly* sequence.

Definition 2 (*M-friendly*) Given an engine M, a malware instance is *M-friendly* if M is capable of transforming it on at least one of its computations.

Since the main goal of morphing engines is to transform the appearance of malware instances, engine friendliness essentially refers to a level of transformability of an instance. In other words, it describes how much of an original instance can be transformed by this morphing engine. As such high transformability can be referred to as high engine friendliness and consequently, low transformability can be viewed as low engine friendliness.

There are a number of measures that can be defined to assess engine friendliness. For example, uniqueness of generated descendants, amount of overhead imposed on new variants or the amount of sequences of instructions in a sample transformed by an engine in a single computation.

Definition 3 (*Engine Friendliness Measure*) A *M-friendliness* $fr(M, p)$ of a program p is any measure that is proportional (not necessarily linearly) to the size of $M(p)$, which is the set of all the programs that can possibly be output by M on input p. One such natural measure, denoted $fr_0(M, p)$, is the size of $M(p)$ itself, that is, $fr_0(M, p) = |M(p)|$. Given two programs p and p', we will say that p is *at least as M-friendly* (with respect to M-friendliness measure $fr(., .)$) as p' if $fr(M, p) \geq fr(M, p')$.

Definition 4 (*Morphing Malware*) Let M be a morphing engine, s some arbitrary sequence, and fr some measure of engine friendliness. We say that s is an instance of a morphing malware, with respect to M and fr, if $fr(M, s) \geq a|M(s)|$, for some $a > 1$.

Morphing engines tend to preserve the level of transformability across generations, that is high transformability (i.e. high engine friendliness) will be preserved through all variants of malware.

Definition 5 (*High Friendliness Preservation*) M is said to be a high friendliness preserving morphing engine if all of the M-descendants of a high engine friendly instance s are *at least as highly M-friendly* (with respect to some M-friendliness measure $fr(., .)$) as s' if $fr(M, s) \geq fr(M, s')$.

Malware engine attribution problems In this work we introduce three variations of malware engine attribution problem: $MVRP$, $DENSITY$ and GEN.

Informally, $MVRP$, the Morphed Variant Recognition Problem, is the general problem of deciding membership in the set of all programs that can possibly be output by a given morphing engine M to which we have input/output (or black-box) access and to whose description we may have access. Formally, $MVRP$ of M can be stated as follows:

Definition 6 ($MVRP_M$) Given a Turing machine (TM) sequence v, does there exist a TM sequence u such that there is a computation of M which returns v on input u?

A more specific version can be formulated as $DENSITY$, a detection problem of descendants of an instance of a morphing malware.

Let (M, x) denote an instance of a morphing malware where x is a highly M-friendly sequence and M is a high friendliness preserving morphing engine. The set of all possible descendants of (M, x) is hence composed "mostly" of non-overlapping instances of sequences inserted by M into the descendant. The ratio of the sum of the sizes of the sequences inserted by M to the size of the descendant is high. This ratio can be seen as an "*engine signature*", indicating the potential involvement of the engine in the generation of an immediate parent of the descendant. We refer to this type of engine signature as clue-density based signature and define it as follows:

Definition 7 (*Clue-density-based engine signature*) Let (1) M denote a high friendliness preserving morphing engine in the sense given in Section 2.1, (2) v denote a highly M-friendly sequence in the sense that every instruction in the sequence is transformable by the engine, (3) $v_{witness}$ denote a sequence returned by M on input v, and (4) W denote the multiset of sequences inserted by M into v as a result of this run. The clue-density-based engine signature σ_M *of* M is given by

$$\sigma_M = \frac{\Sigma_{r \in W}|r|}{|v_{witness}|} \tag{1}$$

In other words, solving $DENSITY_M$ is deciding whether a suspect program p contains non-overlapping occurrences of code segments c_j each of which is identical to a sequence insertable by M, and such that the ratio of the sum of their sizes to that of the size of p is equal to the signature σ_M of M.

Formally, the $DENSITY$ detection problem of M-descendants of an instance of a morphing malware can be defined as follows:

Definition 8 ($DENSITY_M$, DD_M) For high-friendliness-preserving morphing engine M that may insert one or more of a finite set of sequences into its input sequence, we denote by $DENSITY_M$ the following problem: Given a sequence v and the set $R = \{r_1, \ldots, r_n\}$ of sequences known to be insertable by M into a malware variant, does there exist $(c_1, \ldots, c_m) \in (\Sigma^*)^m$ such that:

1. $\forall 1 \leq j \leq m, \exists 1 \leq i \leq n$ such that $c_j = r_i$,
2. $\exists w_1, w_2, \ldots, w_n, w_{n+1} \in \Sigma^*$ such that $v = w_1 c_1 w_2 c_2 \cdots w_n c_n w_{n+1}$, and
3. $\sigma_M = \frac{\sum_{i=1}^{m} |c_i|}{|p|}$.

Both problems $MVRP$ and $DENSITY$ can be generalized to finding a specific generation of malware variants produced by engine M, as stated by the problem GEN.

Definition 9 (GEN_M^n) For every positive integer n and morphing engine M, we denote by GEN_M^n the following problem: Given two sequences v and $v' \in \Sigma^*$, is v' an nth-generation M-descendant of v?

A more general definition of this problem can be extended to a set of all possible descendants of a malware variant.

Definition 10 (GEN_M^*, D_M) For morphing engine M, we denote by GEN_M^* the following problem: Given two sequences v and $v' \in \Sigma^*$, does there exist a positive integer n such that v' is an M-descendant of v?

3 Morphed variant recognition problem ($MVRP$)

In this section we present our approach to address the Morphed Variant Recognition Problem (*MVRP*). Specifically, we describe a method for computing an engine signature for the morphing engine, given a training sample of programs known to have been generated by a given engine.

We propose to adapt the $n$gram frequency vector based method. This method was also successfully employed to attribute natural language documents to their human authors [30], and to determine whether a suspect program is malicious, although without any attempt to attribute it to an engine [1]. N-grams have also been used to attribute binary files to datatype that follow predetermined criteria (type signatures) [61].

In the context of attributing morphed malware to its engine, we propose to use optimized $n$gram frequency vectors (for an appropriately chosen positive integer n) of a program's opcodes as a feature vector for that program. Treating these features as unique characteristics of a morphing engine, we generate an engine signature following a modified Rocchio classification algorithm [58]. The Rocchio classifier is a simple centroid-based algorithm known to be one of the best for document classification [21]. This algorithm relies on a centroid vector to represent documents of each class, assigning each new document to a class that is most similar to a centroid vector. In spite of its simplicity, the algorithm has been shown to consistently outperform other algorithms such as the k-nearest-neighbors and Naive Bayesian [21].

3.1 nGram-based attribution of morphed malware to its engine

Let our alphabet A be the finite set $\{a_1, a_2, a_3, \ldots, a_m\}$ of opcodes for the computing platform for which we intend to run a procedure that is capable of attributing programs to one or more members of a fixed, finite set of known morphing engines. Let $NG(A)$ denote the set of all of A's $n$grams. Given any two distinct $n$grams ng_i and ng_j of A, we have $ng_i \prec ng_j$ or $ng_j \prec ng_i$, where $\prec$ is a total order relation on A. ($\prec$ is guaranteed to exist since A is finite).

We first process an assembly language program p by removing all of the non-empty strings occurring in P, except for the opcodes. Let O_p denote the sequence of opcodes obtained as a result of this first processing stage. We denote by $|O_p|$ the length of this sequence (i.e., the number of opcodes occurring in the sequence).

The (normalized) 1gram instruction frequency vector $1.NFV$ of P is the tuple

$$1.NFV(p) = \left(\frac{f_i}{\Sigma_{j=1}^{m} f_j} \right)_{1 \leq i \leq m}, \tag{2}$$

where, for $1 \leq i \leq m$, f_i is the frequency (or the number of occurrences) in O_p of opcode a_i.

More generally, for $n > 2$, the (normalized) $n$gram instruction frequency vector *n.NFV* of P is the tuple

$$n.NFV(p) = \left(\frac{f_{ng_i}}{\Sigma_{j=1}^{k} f_{ng_j}} \right)_{1 \leq i \leq k}, \tag{3}$$

where $k = \frac{n!n!}{m!(m-n)!}$ is the number of distinct $n$grams that can be generated using A's opcodes, and for $1 \leq i < j \leq \frac{n!n!}{m!(m-n)!}$, $ng_i \prec ng_j$. f_{ng_i} and f_{ng_j} are the frequencies (or the number of occurrences) in O_p of $n$grams ng_i and ng_j, respectively.

For any given positive integer n, we define the $n$gram engine signature ES_n of a given morphing engine M as the arithmetic average of the NFV's of a set $S = (p_1, p_2, \ldots, p_s)$ of programs known to have been generated by the engine. In other words,

$$ES_n = \sum_{i \in \{1,\ldots,s\}} n.NFV_i / |S|, \quad (4)$$

where, $n.NFV_i$ is the $n.NFV$ of program P_i.

Attribution of a suspect program to any one of a given set of morphing engines is carried out by measuring the distance between the $n.NFV$ of the program to each of the $n$gram engine signatures ES_n of the engines. For any given positive integer n and the instruction frequency vectors $n.NFV_1$ and $n.NFV_2$, we will use the following distance measure [30] to compute the dissimilarity $D(n.NFV_1, n.NFV_2)$ between these vectors:

$$D(n.NFV_1, n.NFV_2) = \sum_{k=1}^{|NG(A)|} \left(\frac{2 \times (n.NFV_1[k] - n.NFV_2[k])}{(n.NFV_1[k] + n.NFV_2[k])} \right)^2. \quad (5)$$

The engine whose signature ES_n is closest to the suspect program's $n.NFV$ is declared to have authored the program.

Implicit in an $n$gram frequency vector is information about the probability that the nth instruction in an $n$gram will follow the $n-1$ instructions preceding it in the $n$gram. This observation shows that choosing $n = 2$ will enable the detector to do just as well as the method proposed by Wong and Stamp [81] that models the generation process (by the engine) of any malware variant as a first order Markov process, where the instruction following the current instruction in the malware could be predicted with a certain probability. For $n > 2$, the authorship attribution method proposed in this section is able to capture more information (than for $n \leq 2$) about how the engine generates the different instructions that compose a morphed malware instance. So, for example, a trigram frequency vector for a morphed malware instance captures the probability that any given instruction will follow any given bigram. However, this also significantly increases a size of frequency vectors to be considered. As such, for a given integer n, the number of components of a program's NFV is equal to the nth power of the instruction set of the platform (x86) or, if we choose to ignore those opcodes which do not occur in the program, to the nth power of the number of distinct opcodes within the program. To reduce the size of the $n$gram instruction frequency vectors to be computed we chose for the proposed method $n = 2$.

3.2 Evaluation

This section presents the evaluation of our attribution method for engine signature generation. The evaluation has been performed in two stages. In the first stage, we analyzed the effectiveness of our signature-based classifier to attribute malware instances generated by seven morphing engines to the corresponding engines. In the second stage, we evaluated the general applicability of our approach on a set of malware samples collected from the wild.

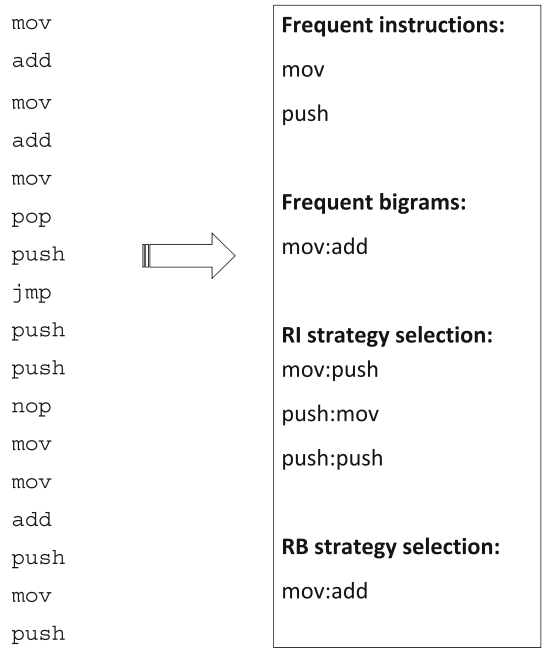

Fig. 1 An illustration of the *RI* and *RB* strategies

3.2.1 Subject

To evaluate the proposed attribution method, seven morphing engines and kits introduced in Sect. 2 were analyzed: ADMmutate [26], CLET [13], NGVCK [51], VCL [75] and three MetaSploit engines [72]: Shikata Na Gai, Fnstenv Mov and Call4dWord. For each of the engines and kits one hundred malware instances were generated for experiments (in a form of binary code for Metasploits engines and assembly language for the rest).

Note, that the set of benign data was not included in our experiments intentionally. An analysis of legitimate programs for an authorship would require a knowledge of an author (more precisely generating engine) which is in this case unknown.

3.2.2 Instruction selection strategies

In evaluation we consider two strategies for selecting the most relevant opcodes: the strategy based on the most relevant instructions, *RI strategy*, and the strategy based on the relevant bigrams, *RB strategy*.

In the *RI strategy*, we consider only those instructions which are "frequent enough" across a sample of programs used for the experiments. Then among all possible bigrams, this strategy retains only those that are composed of any two of the most frequent opcodes across the collected samples. For example, consider a set of instructions given in Fig. 1.

Based on a set of the frequent instructions, the *RI strategy* retains *mov:push*, *push:mov* and *push:push* bigrams, while discarding *mov:add*, despite the fact that *mov:add* bigram is the most frequent among samples. Since this strategy does not account the actual frequency of a formed bigram across a sample, for a small number of frequent instructions, the *RI strategy* might generate vectors of a significantly larger size.

An alternative way of finding the most relevant bigrams, followed by the *RB selection strategy*, is to directly select a subset of those bigrams which are the most frequent across the samples. Following our example, the *RB strategy* will directly choose *mov:add* bigram for the experiment. Note that this strategy also allows us to address the performance issue imposed on the *ES* classifier by the *RI strategy* by bounding the size of frequency vectors.

For each of these strategies the set of the relevant instructions is selected based on *cumulative occurrence*, i.e., a total number of times opcode/bigram appears across all samples, *frequency of occurrence*, i.e. the number of opcode/bigram occurrence in each malware instance and opcode/bigram *information gain* [49]. Figure 2 shows the signature vectors generated by both strategies based on cumulative occurrence. As it can be seen on the figures, with the *RI* strategy taking the 32 most frequent instructions yields very large *NFV*s. For example, the smallest vector composed of 303 numbers, each denoting a frequency for some bigram, is constructed for the VCL engine, while the largest *NFV* containing 550 numbers belongs to the NGVCK engine. The picture is different for the *RB* strategy: for the 1,000 most relevant bigrams, the size of vectors generally stays below 1,000. As such, the largest vector constructed for the NGVCK engine consists of 627 numbers. While theoretically the size of the *NFV* with the *RB* strategy should be linear in the number of the relevant bigrams, we reduce the size by explicitly maintaining only the bigrams with non zero frequencies.

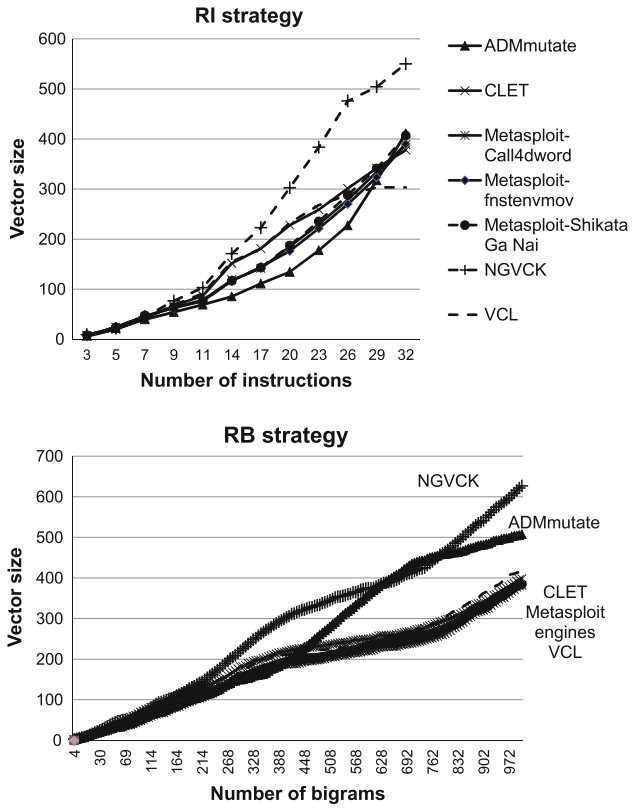

Fig. 2 The *ES* signature vector size (*cumulative occurrence* method)

Since the resulting vectors are similar in sizes, in our experiments we chose to retain the 32 most relevant instructions in the *RI* strategy and the 1,000 relevant bigrams in the *RB* strategy. Table 1 lists the selected instructions and bigrams. Although the selected instructions for both strategies are similar, the complete set of 1,000 bigrams includes bigrams containing instructions not appearing in the set selected by the *RI* strategy. Thus in spite of infrequency of those individual instructions, the formed bigrams appear to

Table 1 The selected relevant instructions for the *RI* strategy and the first 20 relevant bigrams selected by the *RB* strategy (in the order from the most frequent opcode to the least frequent opcode)

	Cumulative occurrence	Frequency of occurrence	Information gain
RI strategy	nop;add;mov;xchg;pop;push;xor;inc; cmp;call;stc;dec;sub;and;sar;int;pusha; je;jmp;ret;lea;clc;aas;aaa;jne; cld;fwait;das;cmc;cltd;cwtl;pushf	mov;xor;xchg;add;pop;push; and;sub;cmp;test;or;imul; sbb;je;adc;jmp;cld;nop; out;clc;in;jae;loop;stc; lea;inc;ja;call;int;fwait; cwtl;jb	call;dec;inc;pusha;ret;fnstenv; pushf;je;div;bound; rexy;rexyz;cmpw;fldz;insw; jc;jnc;jz;o;pushl;aaa;xorl; aas;outsw;jnz;movzwl;movb; popl;cmpb;imul;arpl;popa
RB strategy	nop:nop;add:add;mov:mov; xchg:xchg;xor:xor;mov:add; sar:mov;mov:stc;stc:sar; add:mov;xchg:pop;pop:xchg; push:push;cmp:cmp;pop:pop; call:cmp;add:push;mov:int; mov:xor;push:mov	mov:mov;xor:mov;mov:xor; xchg:mov;mov:sub;mov:and; je:imul;mov:add;inc:pop; pop:mov;add:mov;sub:mov; mov:xchg;pop:pop;inc:inc; mov:pop;push:push;mov:cmp; xor:add;pop:inc	dec:inc;inc:dec;push:inc; push:je;je:imul;add:inc; add:dec;inc:add;push:push; dec:add;inc:pop;push:dec; inc:push;call:mov;jmp:call; fnstenv:pop;nop:nop; call:call;je:jmp;push:imul

Table 2 Average accuracies of the k-nn classifier (*RI* selection strategy)

RI	3	4	5	6	7	8	9	10
1-nn	0.945	0.990	0.993	1	1	1	1	1
5-nn	0.948	0.988	0.993	1	1	1	1	1
10-nn	0.945	0.988	0.993	1	1	1	0.995	1
15-nn	0.915	0.955	0.960	0.963	0.965	0.965	0.923	0.945
20-nn	0.918	0.955	0.958	0.960	0.965	0.948	0.870	0.965

Table 3 Average accuracies of the k-nn classifier (*RI* selection strategy)

RB	3	4	5	6	7	8	9	10	11
1-nn	0.995	0.983	0.990	0.993	0.995	0.995	0.995	0.995	0.995
5-nn	0.995	0.978	0.985	0.993	0.995	0.995	0.995	0.995	0.995
10-nn	0.993	0.975	0.990	0.993	0.995	0.995	0.993	0.995	0.995
15-nn	0.993	0.980	0.955	0.958	0.960	0.960	0.960	0.960	0.958
20-nn	0.963	0.973	0.955	0.958	0.960	0.960	0.958	0.958	0.958
RB	12	13	14	15	16	17	18	19	20
1-nn	0.995	0.995	0.995	0.995	1	1	1	1	1
5-nn	0.995	0.995	0.993	0.993	1	1	1	1	1
10-nn	0.995	0.995	0.993	0.993	1	1	1	0.998	0.998
15-nn	0.958	0.960	0.958	0.958	0.965	0.965	0.965	0.963	0.963
20-nn	0.958	0.958	0.958	0.958	0.965	0.965	0.965	0.963	0.963

be more relevant, i.e., frequent across the samples. Since both strategies operate on the bigrams, we would expect the *RB* strategy be more accurate.

3.2.3 Detection effectiveness

To evaluate the proposed method using the *RI* strategy we considered bigrams which were composed of any two of the 32 most frequent opcodes across the 800 collected instances. A set of 2gram instruction frequency vectors was extracted from the available instances and a classifier for these *2.NFV*s was constructed.

To compute and then evaluate the signature for each of the seven morphing engines and kits, 10-fold cross validation was employed: the collected instances were divided into the training and testing sets in the following manner: 90 instances from each of the samples were set aside as the training sets for the corresponding engines and the remaining 10 instances from each sample were combined into a testing set (total of 80 instances).

The signatures ES_2 for each engine were computed using training sets as described in Sect. 3.1. The generated signatures were then evaluated by measuring a distance between the signature and the frequency vector *NFV* of the program from the testing set (see Sect. 3.1). The accuracy of this *ES* classifier was evaluated by measuring the ratio of the number of those test *NFV*'s which were found to be closer to the signature ES_2 of a sample whose label is different from theirs to the size of the testing set.

In addition to testing each of the *ES* classifiers, we evaluated several *k-nn* classifiers [31] to determine how well these widely used classifiers would be able to attribute each of the *NFV*'s in a testing set to their corresponding engines. A *k-nn* classifier, where k is a fixed positive integer, is an instance based classifier that predicts the class of a test instance by counting its k nearest neighbors, for a given distance measure, from a diverse set of labeled training instances, and then returning the class label (the name of a morphing malware engine in our case) that has the most number of representatives among the test instance's k nearest trainers. *k-nn* classifiers were run in Weka [20] using normalized Euclidean distance and the 10-fold cross validation method.

In the experiments with both classifiers, we chose to ignore the cases where $i = 1$ and $i = 2$, since bigrams which only contain either or both of the most frequent opcode and the second most frequent opcode do not bring much discriminating information.

The obtained results for the k-nn classifier for $k = 1, 5, 10, 15$, and 20 are shown in Tables 2 and 3. The corresponding results for the *ES* classifier are given in Fig. 3.

As the results in Tables 2 and 3 show, the k-nn classifiers performed very well with both strategies, reaching a perfect

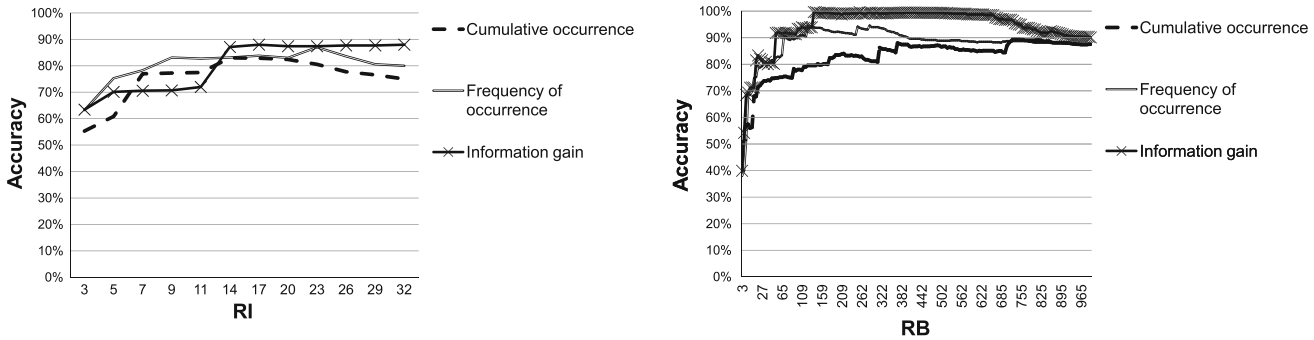

Fig. 3 Average accuracy of the *ES* classifier with the *RB* and *RI* strategies

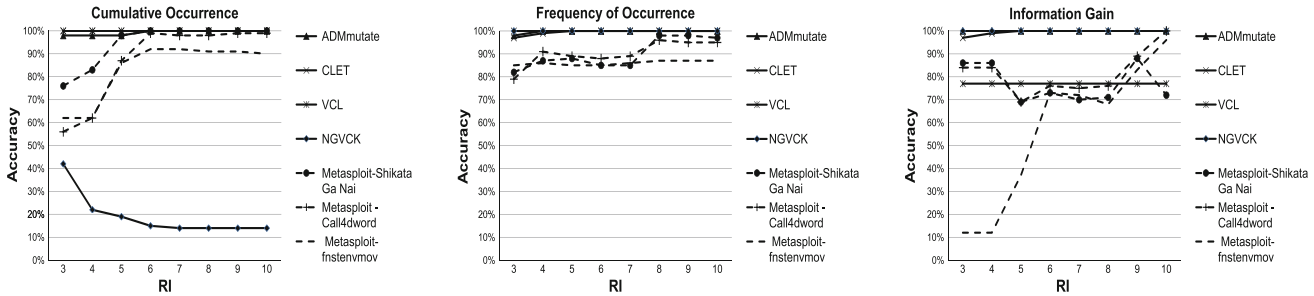

Fig. 4 Accuracy of the *ES* classifier (*RI* strategy)

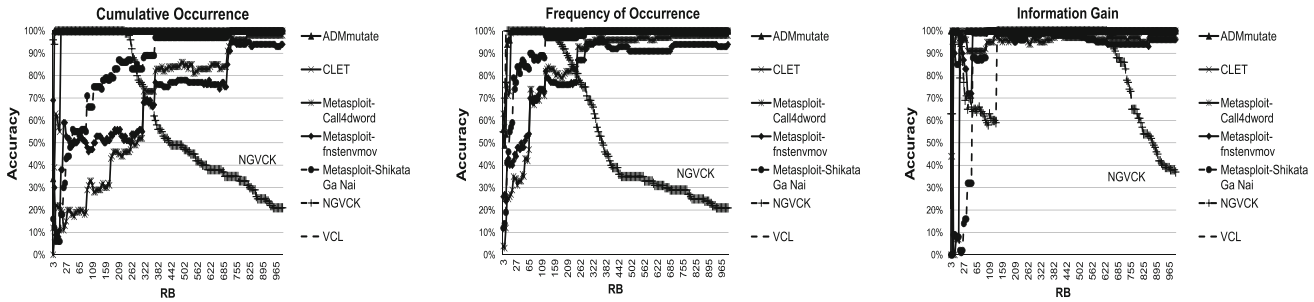

Fig. 5 Accuracy of the *ES* classifier (*RB* strategy)

filtering accuracy for certain choices of relevant instructions and bigrams.

The k-nn classifier was able to achieve 100 % accuracy in the first set of experiments for $RI = 6$ to $RI = 10$ for $k = 1, 5$ and for $RI = 6$ to $RI = 8$ for $k = 10$. In the second set of experiments, the perfect accuracy was reached at $k = 1$ and 5 for $RB = 16$ to $RB = 20$ and at $k = 10$ for $RB = 16$ to $RB = 18$.

In spite of its perfect accuracy, the k-nn classifier is known to incur a high computational cost mainly due to the necessity to compute the distance between a test instance and each of the training examples. In this context, the proposed *ES* classifier offers a better approach by essentially using one signature per engine.

The average *ES* classifier performance is shown in Fig. 3. As expected, our classifier performed better with the *RB* strategy reaching the accuracy of 99 % with the information gain selection method, 94 % with the frequency of occurrence selection method, and 90 % with the cumulative occurrence method. With the *RI* strategy the *ES* classifier was only able to achieve 88 % with the information gain method.

More insight into the *ES* classifier performance can be gathered from Figs. 4 and 5. The proposed classifier was able to reach perfect accuracy for VCL, NGVCK, CLET, ADMmutate engines in both strategies. However, the best performance was achieved in the *RB* strategy with the information gain method, when the classifier reached the 100 % accuracy for VCL, NGVCK, CLET, ADMmutate and Metasploit's Shikata Ga Nai engines and near perfect accuracy for the other two Metasploit engines (in the range from 96 % to 99 % for *RB = 400* to *RB = 600*).

In other words, the engine signatures capturing the frequencies of anywhere between 400 and 600 the most frequent

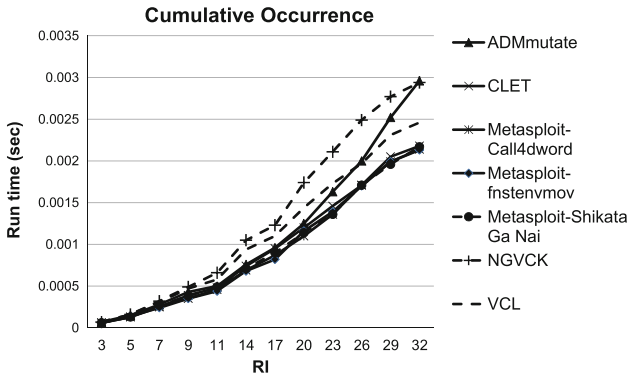

Fig. 6 Runtime performance of the *ES* classifier (*RI* strategy)

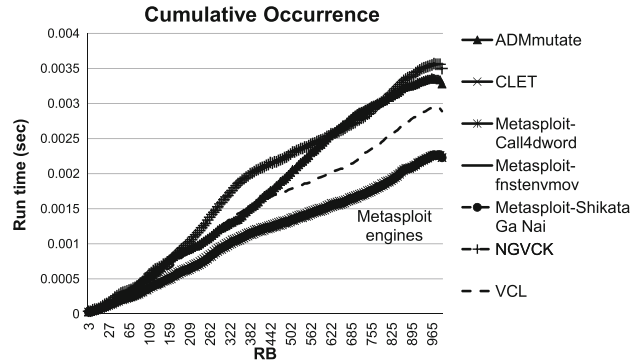

Fig. 7 Runtime performance of the *ES* classifier (*RB* strategy)

bigrams were sufficient to provide the near perfect detection rate for most of the engines.

This result is consistent with findings of Song et al. [64] that analyzed the variance of decryptors in instances generated by the polymorphic engines. They showed that both CLET and Metasploit's Fnstenv Mov contain artifacts that are always present in instances of these engines, while Metasploit's Shikata Ga Nai and Call4dWord generate similar blocks of code that are scattered throughout their decryptors.

In general, the classifier's performance in attributing malware to the engines improved with the signatures size. This is an expected behavior as larger signatures incorporate more information about the most frequent bigrams and consequently provide better differentiation between the engines. One exception to this rule is the NGVCK engine. The proposed classifier performance for the NGVCK engine sharply increases reaching 100 % accuracy and then slowly declines with the signature size. Since the large NGVCK engine signature accumulates bigram frequency information similar to that contained in other engines' signatures, it becomes less precise thus making the overall detection using the proposed signature-based method less accurate.

3.2.4 Performance

We evaluated the processing requirements of the proposed classifier using a system with an Intel(R) Xeon 2.67 GHz system. Figures 6 and 7 show the runtime processing requirements for both *RI* and *RB* strategies with the cumulative occurrence method. The other instruction selection methods performed similarly.

Due to the nature of the proposed classifier, the runtime performance is mainly affected by the size of the underlying frequency vectors. As such, the *RI* strategy performed significantly slower taking almost 0.0075 s for *RI = 10*, while the *RB* strategy only required 0.0035 s for *RB = 1,000*. This is mainly due to the actual size of the *NFV*s that are on average twice as large in the *RI* strategy as they are for *RB = 1,000*.

3.2.5 Experiments with the samples captured in the wild

We have tested our proposed engine signature based approach on the wild collection of malware samples. These samples were acquired from an anti-virus company that performed their initial processing (e.g. unpacking) and classification. We selected three malware families: W32.Agent, W32.Hupigon and W32.Pcclient; and used one hundred samples to represent each family. Note that, since we do not have a ground truth for this data, we simply rely on the classification provided by the anti-virus company and assume that all these samples were generated by the same morphing engine. For these experiments, we performed the same 10-fold cross validation method described above.

The classification accuracy results for these malware samples are shown in Figs. 8 and 9. Similar to the controlled experiments, in the experiments with the wild samples, the *RB* strategy gave consistently better results than the *RI* strategy, reaching 100 % accuracy for W32.Pcclient samples with the information gain method (*RB* = 3 and 4) and 99 % accuracy with the frequency of occurrence method (*RB* = 382 to *RB* = 592). Interestingly, all three instruction selection methods on average performed similarly in the *RB* strategy with the cumulative occurrence achieving the highest accuracy of 78 %, and the information gain reaching the lowest accuracy of only 73 %. This is different from the controlled experiment's results, where the information gain method on average had significantly better detection rate than the other two methods. Overall, the performance of our classifier varied more on the wild samples than it did in a controlled environment. In the lack of ground truth, we can attribute this variability to either incorrect classification of samples by the anti-virus company or perhaps unsuitability of our assumption. The latter would indicate that these sample came a number of morphing engines which would be challenging if impossible to confirm.

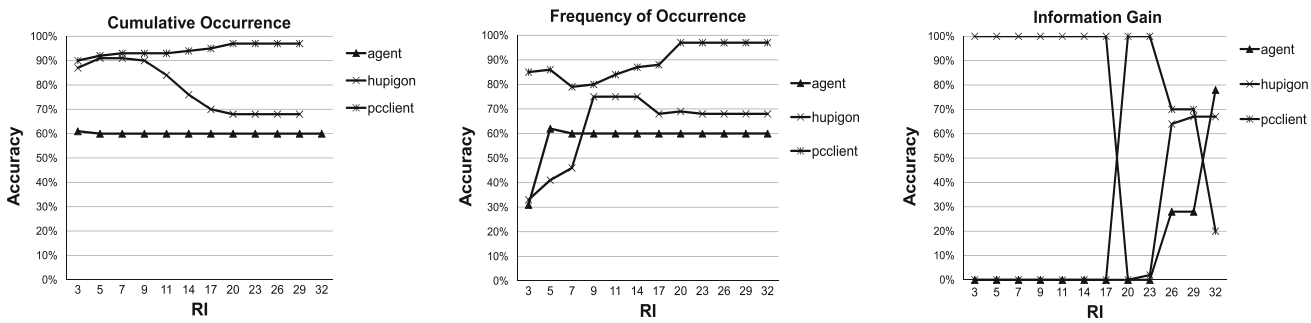

Fig. 8 Accuracy of the *ES* classifier on the malware samples captured in the wild (*RI* strategy)

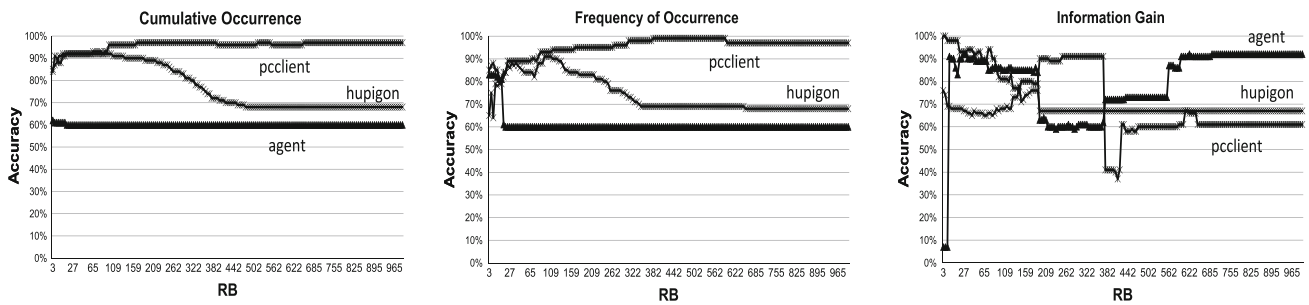

Fig. 9 Accuracy of the *ES* classifier on the malware samples captured in the wild (*RB* strategy)

3.2.6 Summary

In summary, we find that the proposed engine signature method of attributing morphed malware to its engine is a rather promising approach due to two reasons.

First, as opposed to traditional signature detection methods that store one signature per malware sample, the proposed method only requires one signature per engine. This is especially appealing to malware detectors which aim space and time efficiency. Since the number of morphing engines is several magnitudes smaller than ever-increasing number of malware samples, a small number of engine signatures is an attractive alternative solution.

Second, the proposed method exhibits a good filtering capacity overall and achieves the 100 % accuracy for some of the most popular engines. As expected, the *RB* strategy gives consistently better results, often significantly outperforming the *RI* strategy. The results of the relevant instruction selection methods are mixed. Although in the experiments with the seven engines, the information gain method unanimously achieved the best performance, in the wild sample set all three methods on average has similar results.

4 Clue-density-based engine signature generation: $DENSITY_M$

In this section we present another variation of a malware engine attribution problem, called $DENSITY$ problem. $DENSITY$ problem targets code substituting morphing engines, i.e., engines that insert into morphed malware instances code segments from a known set. In this case, deciding whether a suspect malware instance is generated by an engine is cast as deciding whether this program contains non-overlapping occurrences of code segments known to be insertable by an engine.

4.1 A code substituting morphing engine

Consider a closed-world morphing engine M that uses a fixed set of productions, each mapping a sequence of instructions (the *left hand side*) to a different sequence of instructions (the *right hand side*). Let T denote the fixed set of productions carried by M. Since M is a closed-world morphing engine, the set T is assumed to be extractable manually, or interactively, from M. Using the $fr_0(M, p)$ measure of engine-friendliness given in the Section 2.1, we say that a code segment is highly M-friendly if the ratio of the frequency of the left hand sides of T which occur in the segment to the number of instructions in the segment is greater than $1 - \epsilon$ for a small $\epsilon > 0$. M is also assumed to preserve high engine-friendliness among all of its generated descendants.

Writers of morphing malware sometimes attempt to achieve this by requiring that at least one occurrence of a left hand side occurs in each of the right hand sides [82]. The transformation step of the morphing engine performs a linear scan of the malware instance to be transformed. Upon visiting a code segment that is also the left hand side of one of T's

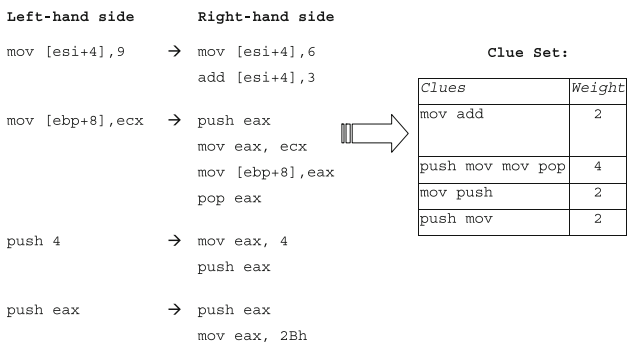

Fig. 10 An example of clue set construction for a subset of transformations of W32.Evol

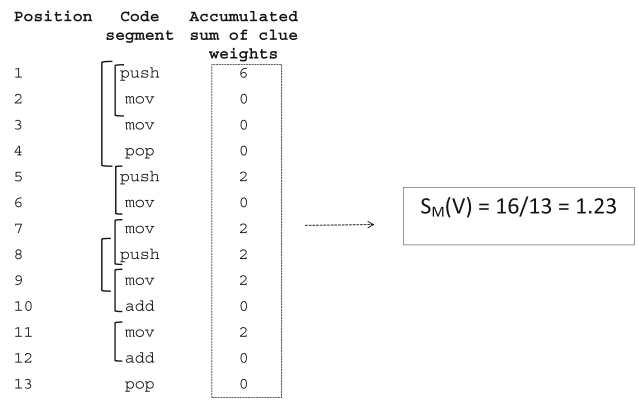

Fig. 11 An example of a clue-density computation on a code segment suspected of having been generated by Evol

productions, the engine probabilistically determines whether the segment should be replaced with its corresponding right hand side.

The high engine-friendliness of a morphing malware variant, coupled with the need to leave as small region as possible unchanged in the variant, imply that M will insert one or more of *a-priori* known code segments in place of the transformable region.

4.2 The clue-density based engine signature computation

Let M denote a morphing engine, equipped with a set T of productions as described in the previous section. Given a code segment V, we abstract each instruction in V to its opcode. (This 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 view the right hand sides of T as *clues* indicating M's potential involvement in producing a code segment.

The example of a clue set construction is illustrated in Fig. 10. The engine maps the sequence of instructions on left hand side to the corresponding code segments that contains one of more sequences of instructions (i.e. right hand side). The clue set is constructed from the right hand side sequences. Clues are chosen and assigned *weight*s equal to their lengths, i.e. number of instructions.

We define a scoring function S_M that takes as input a code segment V (a sequence of x86 opcodes) and returns a score for V. The score of V with respect to M, which we denote by $S_M(V)$, is considered to be a measure of the evidence linking V to M. The scoring function is computed as follows

$$S_M(V) = \frac{\sum_s \sum_c w_c e_{sc}}{\mid V \mid} \tag{6}$$

where $\mid V \mid$ is the instruction count of V, w_c is the weight of clue c (in this case, its instruction count), and $e_{sc} = 1$ 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 would accumulate the sum of the weights of these clues in some variable. It would finally divide the accumulated sum by the instruction count of V and then return the result of the division. Figure 11 gives an example using the scoring function. As such at position one instruction push is the beginning of two clues push, mov and push, mov, mov, pop, the sum of these clues' weights is six.

4.3 Evaluation

To evaluate our clue-density based approach to attributing malware samples to code substituting morphing engines, we experimented with the `W32.Evol` virus and its metamorphic engine `Evolve`. The `W32.Evol` is a full metamorphic virus and its engine `Evolve` is known to operate in manner described in Sect. 4.1 [68].

For our experiments, we implemented a simulator for the `W32.Evol` virus [68], and used it to evaluate the proposed scoring function (more details on the simulator implementation can be found in [8]). Using the simulated engine one hundred distinct instances of `W32.Evol`, spanning four generations of descendants of the original instance, were generated. In addition, a set of one hundred distinct benign programs were prepared. The benign programs were retrieved from http://download.com/, http://sourceforge.net/, and from a fresh installation of Windows Vista™; and processed to extract the opcode sequences.

The productions used by `W32.Evol`'s engine, abstracted to their opcode representations, have exactly 29 distinct right hand sides (i.e., clues), varying in size from one to six opcodes. These clues were ordered by size from smallest to largest. Twenty nine classifiers, C_1 though C_{29}, were then constructed. Each C_i was made to use clues c_1 through c_i, along with a suspect program, as inputs to its scoring function.

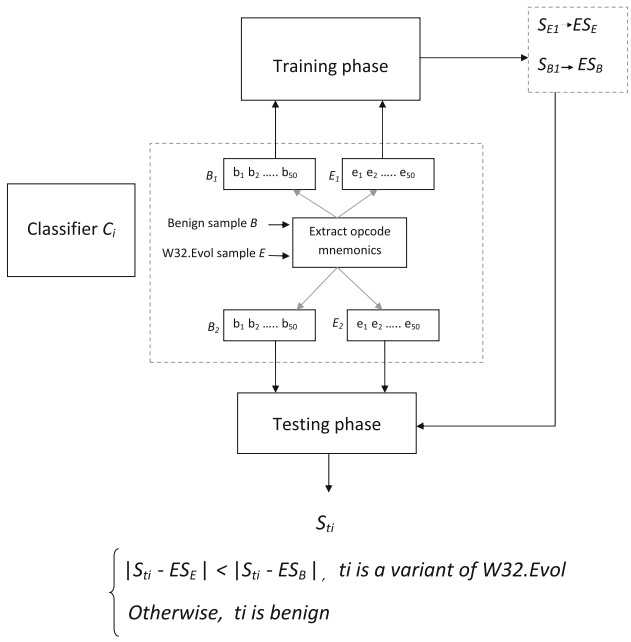

Fig. 12 Operation of the $Density_M$ classifiers

These classifiers were evaluated on the set of collected samples divided to form training and testing subsets. The set of benign opcode sequences B was divided into two disjoint subsets B_1 and B_2 of size 50 each. Similarly, the `W32.Evol` sample, E, was broken into two subsets E_1 and E_2 of same size. The operation of the classifiers based on these subsets is described in Fig. 12.

Each of the classifiers C_i was first trained by computing a score for instances of E_1 and B_1 using clues c_1 through c_i. These scores are what we call the `W32.Evol`'s engine signature, ES_E and the "benign engine signature", ES_B. E_2 and B_2 sets were then used for testing the classifier. For each of test instances ti in E_2 and B_2 the classifier computed a score. If the magnitude of the difference between the computed score of that instance and `W32.Evol`'s engine signature is smaller than that between a score of a test instance and the "benign engine signature" ($|S_{ti} - ES_E| < |S_{ti} - ES_B|$), then the classifier declared a test instance to be a variant of `W32.Evol`.

However, if a score of a test instance was found to be equidistant to both signatures, a class is chosen at random by the classifier.

Two fold cross validation was then used (by using the members of B_2 and E_2 as trainers, and those of B_1 and E_1 as testers) to cross check the accuracy (AC), the false positive rates (FP), and the false negative rates (FN), of each of the twenty nine classifiers (RC).

The evaluation results shown in Table 4 reveal that the proposed method is able to achieve 96 % accuracy with a false positives rate of 3 %. In general, the method's performance improved as more clues were used as input to the scoring function, reaching an accuracy of 96 % for a clue count of 25, while having only 56 % accuracy with one clue.

4.4 Discussion

With this classification accuracy, the proposed scoring method offers an attractive alternative solution to a malware detection and analysis, compared to the traditional signature-based approach. Specifically, we see the following advantages:

- This method requires a suspect program to be only disassembled, stopping early in the malware analysis pipeline.
- The storage requirements consist of (1) just one real number to be used as a signature for the engine, and (2) a small

Table 4 Filtering accuracy of the $DENSITY_M$ classifier

RC	1	2	3	4	5	6	7	8	9	10
FN (%)	46	22	13	22	20	14	14	13	14	13
FP (%)	42	32	31	32	32	32	28	26	24	21
AC (%)	56	73	78	73	74	77	79	80.5	81	83
RC	11	12	13	14	15	16	17	18	19	20
FN (%)	16	12	14	14	12	10	9	9	9	11
FP (%)	12	7	6	6	4	4	3	3	3	3
AC (%)	86	90.5	90	90	92	93	94	94	94	93
RC	21	22	23	24	25	26	27	28	29	
FN (%)	10	9	9	8	**5**	6	6	6	6	
FP (%)	3	3	3	3	**3**	3	3	3	3	
AC (%)	93.5	94	94	94.5	**96**	95.5	95.5	95.5	95.5	

set of clues to be used to compute the score of suspect programs.
- The worst case time complexity of S_M is a constant multiple of $|V|*|C|$, where $|V|$ is the frequency of the opcodes within a suspect program V, and $|C|$ is the number of clues that are to be inspected by the scoring function.

Although in this work we chose to experiment with clue weights equal to their instruction count, other kinds of weight assignments for the clues can be potentially beneficial. For example, a stand-alone garbage segment could be given more weight than a right hand side segment, since odds are low that a benign program contains a particular do-nothing segment, especially a large one, known to be routinely inserted by M at more than one location. 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, inducing smaller clues in the output program. 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 each, leaving malware authors with fewer transformation options to replace any given instruction.

5 Recognition of malware variant descendants: GEN_M^n and GEN_M^*

In this section we introduce GEN, the third variation of malware engine attribution problem. Informally, GEN is the problem of deciding whether a malware instance is a descendant of a known morphing malware variant *Eve* given that large-enough samples from known generations of descendants of *Eve* are available.

We propose solutions to two versions of this problem, GEN_M^n and GEN_M^* as defined in Sect. 2.1. Both solution heuristics leverage Markov chain theory to define quickly-checkable properties of a morphing engine. The key of this approach is to select properties that are indicative of certain morphing actions, and in particular can determine the frequency of these actions in subsequent generations of morphed malware.

In this work we target morphing engines that apply a fixed, finite set of transformation rules. These rules are used by an engine to probabilistically substitute instructions present in a mutated malware variant to a corresponding sequence of instructions thus generating a new variant's descendant.

The proposed solutions to GEN_M^n and GEN_M^* problems rely on program's optimized instruction frequency vectors as indicators of transformation rules applied in generation of *Eve*'s descendants [9]. More specifically, we propose a procedure that given an instruction frequency vector of a morphing malware predicts an average instruction frequency vector of n_{th} generation descendants of this variant. The intuition behind this approach stems from the observation that for this type of morphing engines average frequencies of instructions are likely to resemble many of the actual instruction frequency vectors of the n_{th} generation descendants of *Eve*. As such the defined engine's instruction frequency vectors serve as an engine signature allowing to calculate the likelihood of a suspicious program being a member of one of *Eve*'s descendants.

5.1 Modeling morphing engine using Markov models

To formalize the proposed approach we use Markov chain theory. For the purpose of this work, we model program property as a state, and map state transitions as predictable changes to that property [10].

- *State.* 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 instruction frequency vector. The instruction frequency vector of a program P, denoted $IFV(P)$, is the n-tuple each of whose components represents exactly one opcode and its frequency (or count) in P. No two components may represent the same opcode.
- *State transition probability.* Given two program states $\boldsymbol{\alpha}_{1,M}$ and $\boldsymbol{\beta}_{1,M}$, a transition probability from $\boldsymbol{\alpha}_{1,M}$ to $\boldsymbol{\beta}_{1,M}$ is a probability that, on input a program whose state is $\boldsymbol{\alpha}_{1,M}$, a morphing engine produces a program whose state is $\boldsymbol{\beta}_{1,M}$. In other words, given a morphing malware with an $IFV_{\boldsymbol{\alpha}_{1,M}}$, this probability will define how likely it is for the morphing engine to produce a descendant with an $IVF_{\boldsymbol{\beta}_{1,M}}$.

Following these two definitions we model a set of morphing transformations performed by an engine with the corresponding probabilities as an IFV transition matrix. Existing work on Markov chains [48] 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. The following two results suggest how and when an *IFV* transition matrix can be used to assist in solving GEN_M^n and GEN_M^*:

1. *Distribution prediction using the successive powers of the transition matrix.* 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 $\mathbf{u}$ denote a prob-

Table 5 An example rule set

	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`		

ability vector which holds the initial probabilities of a malware instance'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 $ijth$ 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 nth-generation descendant of *Eve* is in some state s_i after n transitions is the ith component of u^n.

2. *Convergence towards a stationary state distribution.* For every transition matrix T of a Markov chain with a finite space, there exists at least one *stationary distribution* $\boldsymbol{\pi}$, i.e., a row vector $\boldsymbol{\pi}$ satisfying $\boldsymbol{\pi} = \boldsymbol{\pi} T$. Furthermore, if T is irreducible and aperiodic, then it has a unique, a-priori computable, stationary distribution $\boldsymbol{\pi}$ given by $\lim_{n\rightarrow\infty} T^n = \mathbf{1}.\boldsymbol{\pi}$, where $\mathbf{1}$ is a column vector all of whose entries equal 1. Hence, for malware whose starting probability distribution on the set of *IFV*s happens to be a stationary distribution for its engine's *IFV* transition matrix, the corresponding states of the elements of every generation of descendants of the malware will be distributed as indicated by $\boldsymbol{\pi}$.

5.2 Computing an IFV transition matrix

For a class of morphing engines which use a *fixed* set of productions, an *IFV* transition matrix may be constructed directly, given just this set of productions, assuming that this set is equipped with a *fixed* production application probability and mapping of instructions to corresponding, possibly larger, code segments. The morphing engine of `W32.Evol` is an example of such class of engines [46]. One of `W32.Evol`'s rules is to insert segments of dead code within the code being transformed. This rule can be captured in a production by mapping an empty instruction to one or more segments of dead code.

For a given class of morphing engines, let $\mathscr{I} = \{I_1, I_2, \ldots, I_m\}$ denote the instruction set of a target computing platform (e.g., the IA-32 instruction set) and T denote a set of n productions used by a morphing engine to transform an input variant. Then,

$$T = \{l_i \rightarrow \{(\mathrm{Pr}_i^j, r_i^j) : 1 \leq j \leq i_{max})\}\} \tag{7}$$

where $l_i \in \mathscr{I}$ and $r_i^j \in \mathscr{I}^+$. In other words, whenever a morphing engine visits an instruction present on the left hand side l_i of some production the engine substitutes l_i for r_i^j with a probability Pr_i^j. An example set of productions' rules is given in Table 5.

In order to allow an engine to choose whether or not to transform an occurrence l_i, we require that exactly one r_i^j to be identical to l_i, and $\sum_{j=1}^{i_{max}} \mathrm{Pr}_i^j = 1$, 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 l_i will be replaced with r_i^j instance. We assume that the probabilities of use are *fixed* for each production and extractable interactively from a 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 maybe also implemented 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 (as it is in the `W32.Evol` and `W32.Simile` metamorphic viruses). If the latter is the case then we assume that the choices are uniform; $\mathrm{Pr}_i^j = \frac{1}{i_{max}}$ for each r_i^j.

Computing transition probabilities Given a fixed set of productions, a transition probability, $T(\boldsymbol{\alpha}_{1,M}, \boldsymbol{\beta}_{1,M})$, from some IFV $\boldsymbol{\alpha}_{1,M}$ to some IFV $\boldsymbol{\beta}_{1,M}$, is computed using the functions F, G, and H.

For instructions $(I_i, I_k) \in \mathscr{I}^2$, let $F_{i,k}(\beta)$ compute the probability that, on one input instance of I_i, the engine generates β instances of I_k.

$$F_{i,k}(\beta) = \sum_{j=0}^{i_{max}} Z(\beta - OCC(r_i^j, I_k)) \times \Pr_i^j. \quad (8)$$

where OCC is the function from $(\mathscr{I}^+, \mathscr{I})$ to $\mathbb{N}$ mapping each (P, I_i) pair to the frequency of instruction I_i in code segment P, and Z is the function from $\mathbb{R}$ to $\{0, 1\}$ which returns 1 if and only if its argument is 0.

Let $G_{i,k}(\alpha, \beta)$ compute the probability that, on α input instances of I_i, the engine generates β instances of I_k. $G_{i,k}(\alpha, \beta)$ returns 0 if $\alpha = 0$.

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 a set of all possible counts of instruction I_k that can be generated by an engine on one input instance of instruction I_i. Let S_β^α denote the set of α-tuples $\boldsymbol{\delta}_{1,\alpha} = (\delta_1, \delta_2, \ldots, \delta_\alpha)$ of elements of S such that $||\boldsymbol{\delta}_{1,\alpha}||_1 = \beta$[2]. $G_{i,k}(\alpha, \beta)$ is hence given by

$$G_{i,k}(\alpha, \beta) = \sum_{\boldsymbol{\delta}_{1,\alpha} \in S_\beta^\alpha} \prod_{j=1}^{\alpha} F_{i,k}(\delta_j), \quad (9)$$

Finally, $H_k(\boldsymbol{\alpha}_{1,m}, \beta)$ is a probability that, on input a program whose IFV is $\boldsymbol{\alpha}_{1,m}$, a probabilistic engine generates β instances of instruction I_k. $H_k(\boldsymbol{\alpha}_{1,m}, \beta)$ can be recursively computed by observing that for $1 < i < m$,

$$H_k(\boldsymbol{\alpha}_{i,m}, \beta) = \sum_{\delta=0}^{\beta} F_{i,k}(\delta) \times H_k(\boldsymbol{\alpha}_{i+1,m}, \beta - \delta) \quad (10)$$

The recursion stops when $i + 1 = m$, since $H_k(\boldsymbol{\alpha}_{m,m}, \beta - \delta) = G_{m,k}(||\boldsymbol{\alpha}_{m,m}||_1, \beta - \delta)$.

The transition probability from one IFV to another is then computed as follows:

$$T(\boldsymbol{\alpha}_{1,m}, \boldsymbol{\beta}_{1,m}) = \prod_{i=1}^{m} H(\boldsymbol{\alpha}_{1,m}, ||\boldsymbol{\beta}_{i,i}||_1). \quad (11)$$

Improving efficiency Operating on a full transition matrix produced by the described above procedure is not only computationally intensive but is also infeasible in practice due to a potentially infinite set of all possible *IFV*s. A more realistic approach is to impose certain constraints on the size of this set while preserving predictive power of a transition matrix. We offer the following two optimization heuristics that allow reducing a size of a full transition matrix:

1. *Selection of Relevant Instructions (RI).* This will allow to select and retain only instructions considered relevant, which will reduce the size of an IFV to the number of considered instructions. This effectively reduces the overhead of computing, storing and manipulating instruction frequency vectors.
2. *IFV Grouping.* A strategy for grouping IFV's into disjoint sets will reduce a column (and row) counts of a matrix. This grouping of IFV will make the transition matrix for each generation of *Eve*'s descendants capture the transition probabilities from one group of IFVs to another, instead of the transition probabilities from one IFV to another.

Following these two strategies will produce a smaller and more efficient IFV transition matrix.

5.3 Evaluation

In our experimental evaluation we primarily focused on the ability of an optimized *IFV* matrix to accurately attribute malware descendants to their respective generations.

To evaluate our approach, we experimented with the `W32.Evol` virus and its metamorphic engine `Evolve` that, as was mentioned earlier, is known to operate on a fixed set of productions. `W32.Evol` falls within the class of morphing malware for which an exact *IFV* transition matrix can be computed (though inefficiently) given just `W32.Evol`'s *Eve*, engine, and probabilistic productions. We however generate a reduced version of an *IFV* matrix by applying two optimization heuristics, the *RI* strategy and the IFV grouping strategy.

The application of *RI* strategy to the production set of `W32.Evol` revealed 19 distinct opcodes. Thus frequency vectors were only constructed for these 19 relevant instructions.

The *IFV* groupings were implemented using a Euclidian norm $||.||_2$[3] by splitting the interval $[\min(||IFV_i||_2), \max(||IFV_i||_2)]$, where $0 < i < v$ and v is the number of samples, into N equal intervals. In the experiments we tested values of N ranging from 2 to 20. This choice of N was primarily guided by performance considerations: since a transition matrix has N by N dimensions then the smaller the value of N the smaller the size of a transition matrix for any given generation.

For our experiments, we implemented a simulator for the `W32.Evol` virus, and used it to generate 120 `W32.Evol`-descendants of an `W32.Evol`'s variant that we obtained from the VX Heavens archive [76]. The descendants spanned the first four generations of descendants, and numbered 30

[2] The problem of finding an $\boldsymbol{\delta}_{1,\alpha} \in S^\alpha$ such that $||\boldsymbol{\delta}_{1,\alpha}||_1 = \beta$ 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 [17]. In practice, one may want to choose to use a polynomial time approximation scheme for computing each of the $G_{i,k}(\alpha, \beta)$.

[3] Euclidian norm shows a vector magnitude and in a given context allows to measure a difference between vectors.

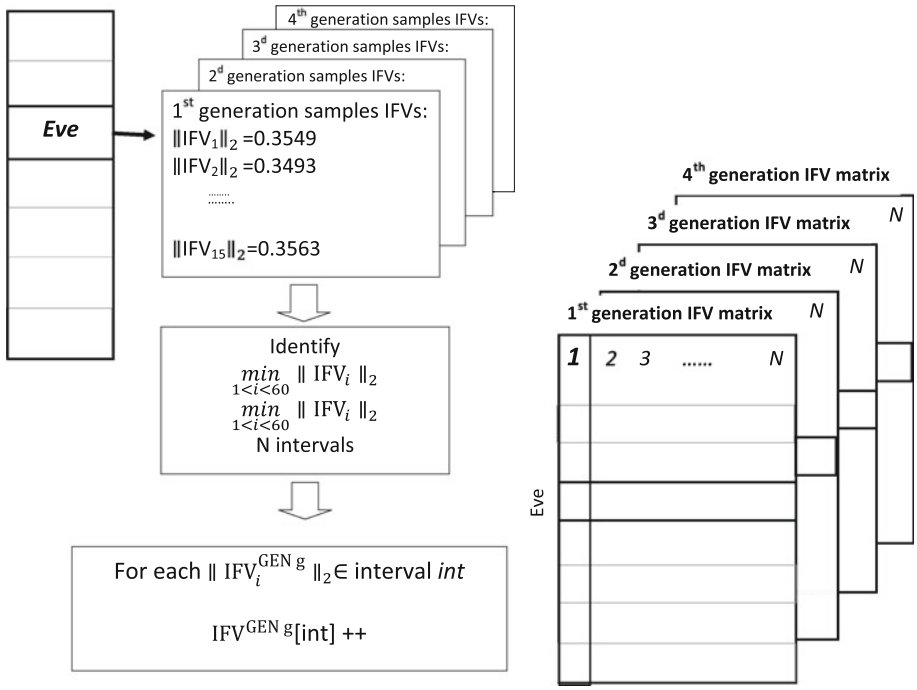

Fig. 13 An illustration of *IFV* transition matrices construction

descendants per generation. Among these descendants, 15 instances were used for building an *IFV* matrix while 15 were reserved for testing.

Having a single variant *Eve*, we illustrate a solution to GEN_M^n problem that can be easily extended to GEN_M^* if other variants are readily available. For a single instance *Eve*, only one row of each IFV transition matrices corresponding the given instance is necessary.

Since our experiments spanned four generations of descendants of a single *Eve*, four IFV transition matrices (each reduced to size $1 \times N$) for each of `W32.Evol`'s generations were built. These matrices were populated with the counts of descendants whose $||IFV||_2$ were represented by the corresponding groups. This procedure is illustrated in Fig. 13. These counts represent transition probabilities among different *IFV*s and define how likely it is for a morphing engine to produce a descendant with a corresponding frequency vector.

The accuracy of the generated *IFV* matrix was tested with the remaining 60 instances representing all four generations. The classifier operated by computing $||IFV||_2$ for each test instance and identifying an interval to which a computed norm belongs. A generation was predicted by choosing an IFV matrix that has the highest transition probability in identified interval. In case a test instance is equally likely to represent two generations the result generation was chosen at random. In these experiments two fold cross validation was employed.

The classification accuracies for selected values of N and relevant instructions, *RI* are shown in Table 6. The results are shown only for the values that yield higher classification accuracy.

5.4 Discussion

The evaluation results reveal that, for a 17×17 transition matrix size, an accuracy of 96.7 % was achieved in attributing morphed instances of `W32.Evol` to their respective generations. An accuracy of 98.3 % was achieved with 20×20 transition matrices.

The proposed method offers a quick decision support procedure for the malware detectors in cases when a morphing engine changes not only a malware appearance but also its behavior as it replicates.

Attributing such morphed malware instance to a given generation of descendants of some *Eve*, perhaps in order to determine what sort of malicious behaviors or instruction patterns are to be expected of the malware instance, is a desirable step in the detection process.

Aside from the one-time computational overhead of constructing a matrix for each generation of *Eve*'s descendants, the storage resources and CPU cycles needed to attribute a program to a generation of *Eve*'s descendants are considerably lower that those that would be needed if the detector were to maintain a signature for each descendant of *Eve*.

6 Related work

The problem of program authorship attribution is not new. Its feasibility has been shown in small pilot study by [33] and has since revolved around the idea of attributing source code through various characteristics of a programmer style [22]. Rosenblum et al. [59] has recently took this idea further and

Table 6 Filtering accuracy of GEN_M^n classifier

RI	$N = 10$	$N = 11$	$N = 12$	$N = 13$	$N = 14$	$N = 15$	$N = 16$	$N = 17$	$N = 18$	$N = 19$	$N = 20$
7	0.833	0.850	0.800	0.817	0.850	0.783	0.867	**0.967**	0.933	0.950	0.933
8	0.617	0.867	0.933	0.850	0.850	0.817	0.867	0.883	0.867	0.950	0.950
9	0.617	0.883	0.717	0.867	0.850	0.817	0.883	0.817	0.867	0.950	0.933
10	0.883	0.717	0.833	0.817	0.800	0.900	0.950	0.933	0.917	0.867	**0.983**

investigated the problem of binary attribution through extraction of stylistic features of program binaries. Although they envisioned the application of their approach in general security, it can potentially be used as a complement to our malware engine attribution approach in the relevant instructions selection phase.

There have been a number of other studies analyzing a suspect binary, without the binary execution or control flow graph (CFG) construction, for a purpose of extracting 'clues' to guide the process of discriminating between malicious and benign programs [4,31,41,47,60,69,83]. Among these are the works that employ byte *n*gram frequency vectors [60]. These methods do not attempt to link malware instances to a particular engine, and hence do not provide the decision support procedures that a malware detector needs to justify a necessity for potentially time consuming program analysis (such as emulation and control flow analysis) that is tailored for a particular engine.

Our approach, on the other hand, specifically focuses on building engine signatures to be able to map malware instance to the corresponding engine. In this context, several methods were proposed for constructing a string-based malware signature (of bytes or opcode mnemonics) to detect members of a given set of malware instances, regardless of whether they have been generated by a fixed engine [19,27,78]. These methods improve upon a similar method that was proposed in 1994 by Kephart and Arnold for automatically constructing, given a malicious binary, a sequence of bytes that may be used as a signature for the binary [29]. These methods are similar to the proposed approach as they aim to reduce a number of signatures stored by malware detectors, and do not require a program's CFG to be constructed, but they also do not provide means to link malware instances to a specific engine.

There has been also some research efforts on automatically building polymorphic malware signatures [42,71] focusing on network traffic analysis.

Program normalization is an alternative approach to reducing the size of the signature space. Normalization aims to remove the results of obfuscation, typically used in metamorphic malware, to allow a detector to analyze the program in its 'normal' form of a known morphing malware. Various types of normalizers were proposed. Program normalizers that do not require a construction of a suspect binary's CFG stop early in the malware analysis pipeline [77]. This allows them to proceed with the signature verification stage without having to solve potentially hard/unsolvable problems such as def-use analysis or halting behavior analysis of subprograms. Normalizers that do require binary's CFG, rely on simplifying a control flow graph by applying transformations similar to those performed by optimizing compilers to reduce the complexity of CFG [7,12].

One advantage that our proposed engine signature method has over a normalization method is that the construction of an engine signature does not require a malware variant be necessarily available. Moreover, it has been shown that no guarantees can be possibly made, that a normalizer will be able to generate a set of normal forms of any given (manageable) size for an arbitrary malware family [77].

Several approaches have used behavioral analysis for malware characterization. These approaches are typically divided into two types: (1) dynamic analysis, i.e. running a suspect program in an emulator and identifying what the program does on a given, carefully chosen set of inputs; and (2) static analysis, i.e., statically analyzing the suspect binary, often by disassembling it, extracting its CFG, focusing on what may signal a malicious content that can be potentially matched to the signature of a known malicious binary [73,79,80]. In general, the static analysis techniques have been shown to not perform well on obfuscated malware [36,38]. A number of static analysis methods that rely on a detector's ability to analyze a suspect's CFG have been suggested, and achieved various degree of success, under condition that the CFG construction is not considerably challenged through obfuscation and that computational resources are available for the detector to use any one of these methods to analyze each of the suspect programs that are submitted to it [5,7,11,24,34,37,40,56]. A taxonomy of these CFG-based malware detection methods is presented in [25]. The fundamental limitations of relying on static analysis to detect malware instances have been documented by [38] and by [50].

As opposed to static analysis, dynamic analysis showed to be more effective against the advanced obfuscation techniques [14,54,55]. Majority of this research focused on code injection attacks and polymorphic shellcode itself

[14,53–55]. However, as some of these dynamic analysis approaches are based on emulation techniques which is not scalable in practice and is vulnerable to certain types of evasion tactics [25,52,57], their potential adoption in real life defense tool is less likely.

7 Limitations

The proposed work focuses on morphing engines and specifically mutation techniques characterized as metamorphic obfuscation. Hence, detection of encrypted malware code is beyond the scope of this work. Although the evaluation of the proposed approach also included several polymorphic engines, the primary focus in these experiments was on decryptors often obfuscated with mutation techniques.

The proposed methods work exclusively at the opcode mnemonic level, since the experiments involved malware instances that were generated as assembly language programs. The methods are certainly applicable at the byte-level, which would relieve malware detector from having to attempt to disassemble suspect binaries that may have been crafted to force disassembly to fail. The accuracy of the methods would need to be reevaluated should one choose to apply them at the byte level.

Using opcode mnemonics to construct and mine signatures injects a measure of semantics-awareness to the proposed detection methods. Semantics-awareness may certainly be taken a step further by taking whole instructions or patterns of instructions into consideration, with the added computational cost of having to extract behavioral patterns from suspect programs and from morphing engines.

8 Conclusions and directions for future work

In this work we proposed and evaluated solution approaches to the three variations of engine detection problems inspired by existing works in forensic linguistics. The proposed solutions use one signature, that of the engine, to determine whether a suspect binary has been authored by a known morphing engine. The main goal of the proposed methods is to relieve the burden of having to extract, maintain, and distribute a signature for each of a large number of possibly obfuscated malware instances, be they known or never-before-seen. The proposed methods do not require that behavioral analysis (such as control flow analysis or emulation) of a suspect program to be performed by the detector. Instead, they only ask the detector be able to disassemble a suspect binary, extract the opcode mnemonics within the disassembly's instructions, and proceed with the attribution phase (of the suspect program to a known morphing engine) using as information about the suspect program only its sequence of opcode mnemonics. Detection accuracies of 96 % and above were achieved by each of the proposed methods, for engine signature sizes ranging from just one real number to a 17×17 matrix of real numbers.

The experimental results suggest that forensic linguistics literature may hold more authorship attribution methods that one may be able to successfully adapt to the context of attributing malware to malware-generating machines.

Acknowledgments This material is based upon work supported by the Air Force Office of Scientific Research under Award No. FA9550-09-1-0715. The authors would like to thank Edna Milgo and Sushma Vallabhaneni for their assistance in conducting the experiments.

References

1. Abou-Assaleh, T., Cercone, N., Kešelj, V., Sweidan, R.: N-gram-based detection of new malicious code. In: 28th Annual IEEE International Computer Software and Applications Conference, pp. 41–42 (2004)
2. Argamon, S., Koppel, M., Pennebaker, J.W., Schler, J.: Automatically profiling the author of an anonymous text. Commun. ACM **52**(2), 119–123 (2009)
3. Babić, D., Reynaud, D., Song, D.: Malware analysis with tree automata inference. In: Proceedings of the 23rd International Conference on Computer Aided Verification (CAV), pp. 116–131. Snowbird, UT (2011)
4. Bilar, D.: Opcodes as predictor for malware. Int. J. Electron. Secur. Digit. Forensics **1**(2), 156–168 (2007)
5. Bonfante, G., Kaczmarek, M., Marion, J.Y.: Architecture of a morphological malware detector. J. Comput. Virol. **5**(3), 263–270 (2009)
6. Borello, J.M., Me, L.: Code obfuscation techniques for metamorphic viruses. J. Comput. Virol. **4**, 211–220 (2008)
7. Bruschi, D., Martignoni, L., Monga, M.: Using code normalization for fighting self-mutating malware. In: Proceedings of International Symposium on Secure Software Engineering. IEEE (2006)
8. Chouchane, M.R., Lakhotia, A.: Using engine signature to detect metamorphic malware. In: 4th Workshop on Recurring Malcode (WORM) (2006)
9. Chouchane, M.R., Walenstein, A., Lakhotia, A.: Statistical signatures for fast filtering of instruction-substituting metamorphic malware. In: 5th Workshop on Recurring Malcode (WORM) (2007)
10. Chouchane, M.R., Walenstein, A., Lakhotia, A.: Using Markov chains to filter machine-morphed variants of malicious programs. In: Proceedings of the 3rd International Conference on Malicious and Unwanted Software (Malware'08) (2008)
11. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-aware malware detection. In: Proceedings of the 2005 IEEE Symposium on Security and Privacy S&P, pp. 32–46 (2005)
12. Christodorescu, M., Kinder, J., Jha, S., Katzenbeisser, S., Veith, H.: Malware normalization. Department of Computer Science, The University of Wisconsin, Technical Report (2005)
13. Detristan, T., Ulenspiegel, T., Malcom, Y., Underduk, M.S.V.: Polymorphic shellcode engine using spectrum analysis. Phrack **61** (2003)
14. Egele, M., Wurzinger, P., Kruegel, C., Kirda, E.: Defending browsers against drive-by downloads: mitigating heap-spraying code injection attacks. In: Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, DIMVA '09, pp. 88–106. Springer, Berlin (2009)

15. Fogla, P., Sharif, M., Perdisci, R., Kolesnikov, O., Lee, W.: Polymorphic blending attacks. In: Proceedings of the 15 th USENIX Security, Symposium, pp. 241–256 (2006)
16. Frantzeskou, G., Gritzalis, S., Macdonell, S.G.: Source code authorship analysis for supporting the cybercrime investigation process. In: Proceedings of 1st International Conference on e-Business and Telecommunications, Networks (ICETE04), vol. 2, pp. 85–92 (2004)
17. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., San Francisco (1979)
18. Gavrilova, M.L., Yampolskiy, R.V.: Applying biometric principles to avatar recognition. In: Proceedings of the 2010 International Conference on Cyberworlds, CW '10, pp. 179–186. IEEE Computer Society, Washington, DC, USA (2010)
19. Griffin, K., Schneider, S., Hu, X., cker Chiueh, T.: Automatic generation of string signatures for malware detection. In: Kirda, E., Jha, S., Balzarotti, D. (eds.) Recent Advances in Intrusion Detection. Lecture Notes in Computer Science, pp. 101–120. Springer, Berlin (2009)
20. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The weka data mining software: an update. SIGKDD Explor. **11** (2009)
21. Han, E.H., Karypis, G.: Centroid-based document classification: analysis and experimental results. In: Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery, PKDD '00, pp. 424–431. Springer, London, UK (2000)
22. Hayes, J.H., Offutt, J.: Recognizing authors: an examination of the consistent programmer hypothesis. Softw. Test. Verif. Reliab. (2009)
23. Holmes, D.: Authorship attribution. Comput. Humanit. **28**, 87–106 (1994). doi:10.1007/BF01830689
24. Holzer, A., Kinder, J., Veith, H.: Using verification technology to specify and detect malware. In: 11th International Conference on Computer Aided Systems Theory (2007)
25. Jacob, G., Debar, H., Filiol, E.: Behavioral detection of malware: from a survey towards an established taxonomy. J. Comput. Virol. **4**(3), 251–266 (2008)
26. K2: Admmutate. http://www.pestpatrol.com/zks/pestinfo/a/admmutate.asp (2005)
27. Karim, M.E., Walenstein, A., Lakhotia, A., Parida, L.: Malware phylogeny generation using permutations of code. Eur. Res. J. Comput. Virol. **1**(1–2), 13–23 (2005)
28. Kennedy, D., O'Gorman, J., Kearns, D., Aharoni, M.: Metasploit: The Penetration Tester's Guide. No Starch Press, USA (2011)
29. Kephart, J.O., Arnold, W.C.: Automatic extraction of computer virus signatures. Virus Bull (1994)
30. Kešelj, V., Peng, F., Cercone, N., Thomas, C.: N-gram-based author profiles for authorship attribution. In: 6th Conference of the Pacific Association for, Computational Linguistics, pp. 256–264 (2003)
31. Kolter, J.Z., Maloof, M.A.: Learning to detect and classify malicious executables in the wild. J. Mach. Learn. Res. **7**, 2721–2744 (2006)
32. Koppel, M., Schler, J., Bonchek-Dokow, E.: Measuring differentiability: unmasking pseudonymous authors. J. Mach. Learn. Res. **8**, 1261–1276 (2007)
33. Krsul, I., Spafford, E.H.: Authorship analysis: identifying the author of a program. Comput. Secur. (1996)
34. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: 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, Berlin (2005)
35. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Polymorphic worm detection using structural information of executables. In: Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection, RAID'05, pp. 207–226. Springer, Berlin (2006)
36. Lakhotia, A., Kumar, E.U., Venable, M.: A method for detecting obfuscated calls in malicious binaries. IEEE Trans. Softw. Eng. **31**(11), 955–968 (2005)
37. Lakhotia, A., Mohammed, M.: Imposing order on program statements to assist anti-virus scanners. In: Proceedings of the 11th Working Conference on Reverse, Engineering (2004)
38. Lakhotia, A., Singh, P.K.: Challenges in getting 'formal' with viruses. Virus Bull. (2003)
39. Layton, R., Watters, P., Dazeley, R.: Unsupervised authorship analysis of phishing webpages. In: 2012 International Symposium on Communications and Information Technologies (ISCIT), pp. 1104–1109 (2012)
40. Leder, F., Steinbock, B., Martini, P.: Classification and detection of metamorphic malware using value set analysis. In: 2009 4th International Conference on Malicious and Unwanted Software MALWARE, pp. 39–46. IEEE (2009)
41. Li, W.J., Wang, K., Stolfo, S.J., Herzog, B.: Fileprints: identifying file types by n-gram analysis. In: Information Assurance Workshop (2005)
42. Li, Z., Sanghi, M., Chen, Y., Kao, M.Y., Chavez, B.: Hamsa: fast signature generation for zero-day polymorphic worms with provable attack resilience. In: 2006 IEEE Symposium on Security and Privacy, pp. 15–47 (2006)
43. Lin, D., Stamp, M.: Hunting for undetectable metamorphic viruses. J. Comput. Virol. **7**(3), 201–214 (2011)
44. Lo, R.W., Levitt, K.N., Olsson, R.A.: Mcf: A malicious code filter. Comput. Secur. **14**, 541–566 (1995)
45. Lyda, R., Hamrock, J.: Using entropy analysis to find encrypted and packed malware. IEEE Secur. Priv. **5**(2), 40–45 (2007)
46. Mathur, R., Maida, A., Palmer, C.E.: Normalizing metamorphic malware using term rewriting. In: Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM '06), pp. 75–84. Hill (2006)
47. Menahem, E., Shabtai, A., Rokach, L., Elovici, Y.: Improving malware detection by applying multi-inducer ensemble. Comput. Stat. Data Anal. **53**(4), 1483–1494 (2009)
48. Meyn, S., Tweedie, R.: Markov Chains and Stochastic Stability. Springer, London (1993)
49. Mitchell, T.M.: Machine Learning. McGraw-Hill, USA (1997)
50. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: 23rd Annual Computer Security Applications Conference (2007)
51. NGVCK: Ngvck download page. VXheavens-Virus eXchange Website. http://vx.netlux.org/vx.php?id=tn02
52. Paleari, R., Martignoni, L., Fresi, G., Bruschi, R.D.: A fistful of red-pills: how to automatically generate procedures to detect cpu emulators. In: Proceedings of the USENIX Workshop on Offensive Technologies (WOOT) (2009)
53. Payer, U., Teufl, P., Lamberger, M.: Hybrid engine for polymorphic shellcode detection. In: Proceedings of the Second International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, DIMVA'05, pp. 19–31. Springer, Berlin (2005)
54. Polychronakis, M., Anagnostakis, K.G., Markatos, E.P.: Network-level polymorphic shellcode detection using emulation. In: Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), pp. 54–73 (2006)
55. Polychronakis, M., Anagnostakis, K.G., Markatos, E.P.: Comprehensive shellcode detection using runtime heuristics. In: Proceedings of the 26th Annual Computer Security Applications Conference, ACSAC '10, pp. 287–296. ACM, New York, NY, USA (2010)
56. Preda, M.D., Christodorescu, M., Jha, S., Debray, S.: A semantics-based approach to malware detection. ACM Trans. Program. Lang. Syst. **30**(5) (2008)

57. Raffetseder, T., Kruegel, C., Kirda, E.: Detecting System Emulators. In: 10th Information Security Conference (ISC) (2007)
58. Rocchio, J.J.: Relevance feedback in information retrieval. In: Salton, G. (ed.) The smart retrieval system: experiments in automatic document processing, pp. 313–323. Prentice-Hall, Englewood Cliffs (1971)
59. Rosenblum, N., Zhu, X., Miller, B.P.: Who wrote this code? Identifying the authors of program binaries. In: Proceedings of the 16th European Conference on Research in Computer Security, ESORICS'11, pp. 172–189. Springer, Berlin (2011). http://dl.acm.org/citation.cfm?id=2041225.2041239
60. Shafiq, Z., Khayam, S.A., Farooq, M.: Embedded malware detection using Markov n-grams. Lect. Notes Comput. Sci. **5137**, 88–107 (2008)
61. Shaner, R.A.: Patent 5991714: method of identifying data type and locating in a file (1999)
62. Singh, P., Lakhotia, A.: Static verification of worm and virus behaviour in binary executables using model checking. In: Proceedings of the 4th IEEE Information Assurance Workshop, pp. 298–300. IEEE Computer Society, Los Alamitos, CA, USA (2003)
63. Sipser, M.: Introduction to the theory of computation. PWS (1997)
64. Song, Y., Locasto, M.E., Stavrou, A., Keromytis, A.D., Stolfo, S.J.: On the infeasibility of modeling polymorphic shellcode. Mach. Learn. **81**, 179–205 (2010)
65. Stamatatos, E.: A survey of modern authorship attribution methods. J. Am. Soc. Inf. Sci. Technol., pp. 538–556 (2009)
66. Stein, B., Lipka, N., Prettenhofer, P.: Intrinsic plagiarism analysis. Lang. Resour. Eval. **45**(1), 63–82 (2011)
67. Symantec: Global Internet Security Threat Report (2009)
68. Ször, P.: The Art of Computer Virus Research and Defense, 1st edn. Symantec Press, Addison Wesley Professional, Reading (2005)
69. Tabish, M., Shafiq, Z., Farooq, M.: Malware detection using statistical analysis of byte-level file content. In: Proceedings of the ACM SIGKDD Workshop on Cyber Security and Intelligence Informatics, pp. 23–31 (2009)
70. Tang, Y., Chen, S.: Defending against internet worms: a signature-based approach. In: INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE, vol. 2, pp. 1384–1394 (2005)
71. Tang, Y., Xiao, B., Lu, X.: Signature tree generation for polymorphic worms. IEEE Trans. Comput. **60**(4), 565–579 (2011)
72. Team, M.D.: Metasploit Project. http://www.metasploit.com (2006)
73. Toth, T., Kruegel, C.: Accurate buffer overflow detection via abstract payload execution. In: Proceedings of the Recent Advances in Intrusion Detection, RAID, pp. 274–291 (2002)
74. Triumphant, Inc.: The world-wide malware signature counter (2010). http://www.triumfant.com/Signature_Counter.asp
75. VCL: Vcl download page. VXheavens: Virus eXchange Website. http://vx.netlux.org/vx.php?id=tv03
76. VX heavens. http://vx.netlux.org
77. Walenstein, A., Mathur, R., Chouchane, M.R., Lakhotia, A.: Constructing malware normalizers using term rewriting. J. Comput. Virol. (2008). doi:10.1007/s11416-008-0081-5
78. Walenstein, A., Venable, M., Hayes, M., Thompson, C., Lakhotia, A.: Exploiting similarity between variants to defeat malware. In: Proceedings of Black Hat Briefings. Black Hat (2007)
79. Wang, X., Chan Jhi, Y., Zhu, S., Liu, P.: Still: Exploit code detection via static taint and initialization analyses. In: Proceedings of the Computer Security Applications Conference, ACSAC, pp. 289–298. IEEE Computer Society (2008)
80. Wang, X., Pan, C.C., Liu, P., Zhu, S.: Sigfree: a signature-free buffer overflow attack blocker. In: Proceedings of the 15th Conference on USENIX Security Symposium, vol. 15. USENIX Association, Berkeley, CA, USA (2006)
81. Wong, W., Stamp, M.: Hunting for metamorphic engines. J. Comput. Virol. **2**(3), 211–229 (2006)
82. Z0mbie: some ideas about metamorphism. http://vx.netlux.org/lib/vzo20.html
83. Zhou, Y., Inge, M.: Malware detection using adaptive data compression. In: AISec '08: Proceedings of the 1st ACM Workshop on Workshop on AISec, pp. 53–60 (2008)