SAS: Semantics Aware Signature Generation for Polymorphic Worm Detection

[1,3]Deguang Kong, [2]Yoon-Chan Jhi, [4]Qihe Pan, [2]Sencun Zhu, [3]Peng Liu, and [1]Hongsheng Xi

[1] Dept. of Automation, University of Science Technology of China, Hefei, China, Email: kdg@mail.ustc.edu.cn, xihs@ustc.edu.cn
[2]Dept. of Computer Sicence and Engineering, Pennsylvania State University, University Park, PA 16802, Email:{jhi,szhu}@cse.psu.edu
[3]College of Information Sciences and Technology, Pennsylvania State University, University Park, PA 16802, Email: pliu@ist.psu.edu
[4]Dept. of Electrical Engineering, Pennsylvania State University, University Park, PA 16802, Email:qup100@psu.edu

Abstract. String extraction and matching techniques have been widely used in generating signatures for worm detection, but how to generate effective worm signatures in an adversarial environment still remains challenging. For example, attackers can freely manipulate byte distributions within the attack payloads and also can inject well-crafted noisy packets to contaminate the suspicious flow pool. To address these attacks, we propose SAS, a novel S*emantics* A*ware* S*tatistical* algorithm for automatic signature generation. When SAS processes packets in a suspicious flow pool, it uses data flow analysis techniques to remove non-critical bytes. We then apply a Hidden Markov Model (HMM) to the refined data to generate state-transition-graph based signatures. To our best knowledge, this is the first work combining semantic analysis with statistical analysis to automatically generate worm signatures. Our experiments show that the proposed technique can accurately detect worms with concise signatures. Moreover, our results indicate that SAS is more robust to the byte distribution changes and noise injection attacks comparing to Polygraph and Hamsa.

Key words: Worm Signature Generation, Machine Learning, Semantics, Data Flow Analysis, Hidden Markov Model

1 Introduction

The computer worm is a great threat to modern network security despite various techniques that have been proposed so far. To thwart worms spreading out over Internet, pattern based signatures have been widely adopted in many network intrusion detection systems; however, existing signature-based techniques are facing fundamental countermeasures. Polymorphic and metamorphic worms (for brevity, hereafter, we mean both polymorphic and metamorphic when we say polymorphic) can evade traditional signature-based detection methods by either eliminating or reducing invariant patterns in the attack payloads through attack-side obfuscation. In addition, traditional signature-based detection methods are forced to learn worm signatures in an adversarial environment where the attackers can intentionally inject indistinguishable noisy packets to misled the classifier of the malicious traffic. As a result, low quality signatures would be generated.

Although a lot of efforts have been made to detect polymorphic worms [1], existing defenses are still limited in terms of accuracy and efficiency. To see the limitations in detail, let us divide existing techniques against polymorphic worms into two categories. The first type of approach is the pattern based signature generation, which uses patterns to identify the worm traffic from the normal traffic as a signature of the invariant part of malicious packets, such as substring and token sequence, etc. For example, systems such as Autograph [2], Honeycomb [3], EarlyBird [4], Polygraph [5], and Hamsa [6] extract common byte patterns from the packets collected in the suspicious flow pool. This approach enables fast analysis on live traffic, but can be evaded by polymorphic worms since the instances of a well-crafted polymorphic worm could share few or no syntactic patterns in common. Moreover, such a syntactic signature generation process can be misled by the allergy attack [7], the red herring and pool positioning attacks [8], and also by the noisy packets injected into the suspicious flow pool [9]. The second approach is to identify the semantics-derived characteristics of worm payloads, as in Cover [10], TaintCheck [11], ABROR [12], Sigfree [13], Spector [14], and STILL [15]. Existing techniques in this approach perform static analysis and/or dynamic analysis (e.g., emulation-based analysis [16]) on the packet payloads to detect the invariant characteristics reflecting semantics of malicious codes (e.g., behavioral characteristics of the decryption routine of a polymorphic worm). This approach is robust to the above evasion attempts because it considers more about semantics. However, the semantics analysis [17] may introduce non-trivial performance overheads, which is often intolerable in network-based on-line detection. Also, the payload analysis could be hindered by anti-static techniques [15] or anti-emulation techniques [18, 19]. Our technique aims at a novel signature that is more robust than the pattern-based signatures and lighter than the prior behavior-based detection methods.

In this paper, we focus on the polymorphic worms can be locally or remotely injected using the HTTP protocol. To generate high quality signatures of such worms, we propose SAS, a novel S*emantics* A*ware* S*tatistical* algorithm that generates semantic-aware signatures automatically. SAS introduces low overhead in signature matching process, thus it is suitable for the network-based worm detection. When SAS processes packets in the suspicious flow pool, it uses data flow analysis techniques to remove non-critical bytes irrelevant to the semantics of the worm code. We then apply a Hidden Markov Model (HMM) to the refined data to generate our *state-transition-graph (STG)* based signatures. Since modern polymorphic engines can completely randomize both the encrypted shellcode and the decryptor, we use a probability STG signature to defeat the absence of syntactic invariants. STG, as a probability signature, can adaptively learn token changes in different packets, correlate token distributions with states, and clearly express the dependence among tokens in packet payloads. Besides this, after a signature is generated, the detector is free of making sophisticated semantic analysis, such as emulating executions of instructions on the incoming packets to match attacks. Our experiments show that our technique exhibits good performance with low false positives and false negatives, especially when attackers can indistinguishably inject noisy bytes to mislead the signature extractor. SAS places itself between the pattern-based signatures and the semantic-derived detection methods, by balancing between security and the signature matching speed. As a semantic-based technique, SAS is more robust than most pattern-based signatures, sacrificing a little speed in signature matching. Based on the statistical analysis, SAS might sacrifice subtle part of security benefits of in-depth semantic analysis, for which SAS gains enough acceleration to be a network-based IDS.

Our contribution is in three-fold.

- To our best knowledge, our work is the first one combining semantic analysis with statistical analysis in signature generation process. As a result, the proposed technique is robust to the (crafted) noisy packets and the noisy bytes.
- We present a state-transition-graph based method to represent different byte distributions in different states. We explore semantics-derived characteristics beyond the byte patterns in packets.
- The signature matching algorithm used in our technique introduces low overhead, so that we can apply SAS as a network-based worm detection system.

The rest of this paper is organized as follows. In Section 2, we summarize the attacks to prior automated signature generation techniques. We then present our semantics-aware polymorphic worm detection technique in Section 3. In Section 4, we discuss the advantages and limitations of SAS, before presenting the evaluation results in Section 5. The related works are reviewed in Section 6, followed by the conclusion in Section 7.

2 Attacks on Signature Generation

2.1 Techniques to Evade Detection

Metamorphism and polymorphism are two typical techniques to obfuscate the malicious payload to evade the detection. Metamorphism [20] uses instruction replacement, equivalent semantics, instruction reordering, garbage (e.g., NOP) insertion, and/or register renaming to evade signature based detectors. Polymorphism [20] usually uses a built-in encoder to encrypt original shellcode, and stores the encrypted shellcode and a decryption routine in the payload. The encrypted shellcode will be decrypted during its execution time at a victim site. The decryption routine can be further obfuscated by metamorphic techniques; the attack code generated by polymorphic engine TAPION [21] is such an example. We note that traditional signature based detection algorithm is easily to be misled by applying byte substitution or reordering. We also doubt if the invariants always exist in all the malicious traffic flows. In fact, we found that for the instances of the polymorphic worm Slammer [22] mutated by the CLET polymorphic engine, the only invariant token (byte) in all of its mutations is "\x04", which is commonly found in all SQL name resolution requests.

2.2 Techniques to Mislead Signature Generation

Besides the obfuscation techniques which aim to cause false negatives in signature matching, there are also techniques attempting to introduce false positives and false signatures. For example, the allergy attack [7] is a denial of service (DoS) attack that

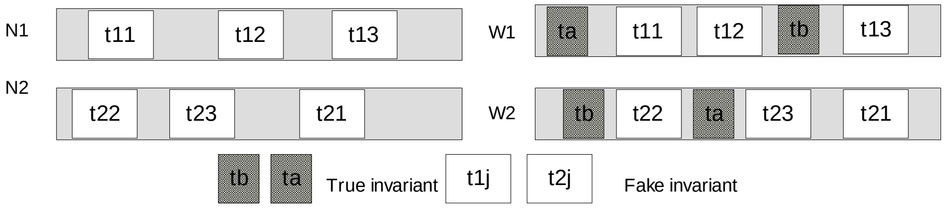
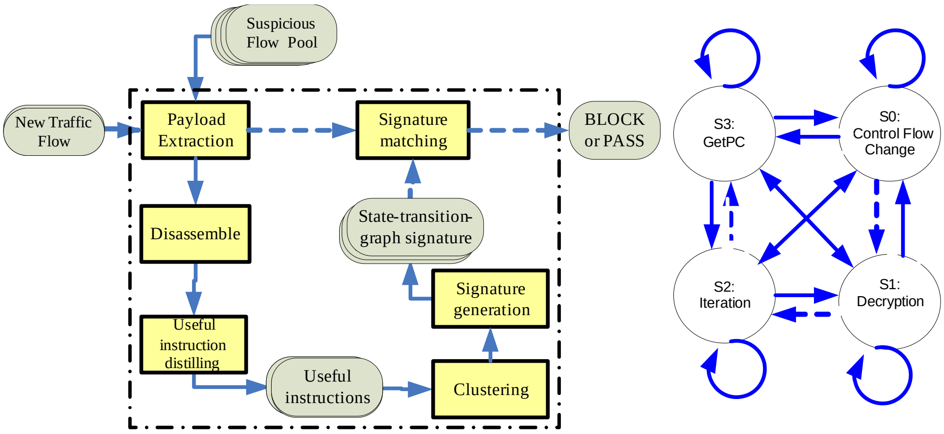


Fig. 1. Suspicious packet flow pool

misleads automatic signature generation systems to generate signatures matching normal traffic flows. Signature generation systems such as Polygraph [5] and Hamsa [6] include a flow classifier module and a signature generation module. The flow classifier module separates the network traffic flows during training period into two pools, the innocuous pool and the suspicious pool. The signature generation module extracts signatures from the suspicious flow pool. A signature consists of tokens, where each token is a byte sequence found across all the malicious packets that the signature is targeting. The goal of a signature generation algorithm is to generate signatures which match the maximum fraction of network flows in the suspicious flow pool while matching the minimum fraction of network flows in the innocuous pool. Generally, existing signature generation systems have two limitations. First, the flow classifier module is not perfect; thus, noise can be introduced into the suspicious flow pool. Second, in reality, the suspicious flow pool often contains more than one type of worms, thus a clustering algorithm is needed to first cluster the flows that contain the same type of worm. Polygraph [5] uses a hierarchical clustering algorithm to merge flows to generate a signature which introduces the lowest false positive rate at every step of clustering process. Hasma [6] uses a model-based greedy signature generation algorithm to select those tokens as a signature which has the highest coverage over the suspicious flow pool.

Let us illustrate the vulnerability of signature generators such as Polygraph and Hamsa when crafted noises are injected in the training traffic as shown in Figure 1. Here N_i denotes normal packets and W_i $(1 \leq i \leq 2)$ denotes the true worm packets. Let us assume the malicious invariant (i.e., the true signature) in the worm packets consists of two independent tokens t_a and t_b, and each of them has the same false positive rate p $(0 < p < 1)$ if taken as a signature. Let the worm packets also include the tokens t_{ij} $(1 \leq j \leq 3)$, each of which has the same false positive rate p as a token in a true signature, thus an attacker can craft normal packets N_is to contain t_{ij} $(1 \leq j \leq 3)$. If all these four flows end up being included in a suspicious flow pool, the signature generation process would be misled.

Setting 1: Let the ratio of the four flows (W_1, W_2, N_1, N_2) in the suspicious flow pool be (99:99:1:1). That is, there is only 1% noise in the suspicious flow pool. According to the clustering algorithm in Polygraph, it will choose to merge the flows that will generate a signature which has the lowest false positive rate. In this example shown in Figure 1, the false positive rate of using signature (t_{i1}, t_{i2}, t_{i3}) by merging flows (W_i, N_i) is p^3 and that of signature (t_a, t_b) by merging flows (W_1, W_2) is p^2. The former is smaller and thus the hierarchical clustering algorithm will merge the flows of W_i with N_i, and it will terminate with two signatures (t_{i1}, t_{i2}, t_{i3}).



(a) (b)

Fig. 2. (a) System architecture. (b) State-transition-graph (STG) model

Setting 2: Let the ratio of the flows (W_1, W_2, N_1, N_2) in the suspicious flow pool be (99:99:100:100). According to Hasma's model-based greedy signature generation algorithm, Hamsa selects the tokens with the highest coverage in the suspicious flow pool. In our example, the coverages for signature (t_{i1}, t_{i2}, t_{i3}) and (t_a, t_b) are 50% and 49.7%, respectively. Thus, Hamsa first selects token (t_{i1}), then (t_{i1}, t_{i2}), and (t_{i1}, t_{i2}, t_{i3}) as a signature as long as the false positive rate of signature (t_{i1}, t_{i2}, t_{i3}) is below a threshold.

From the above two cases, we can clearly see that if an attacker injects noises into the suspicious flow pool, the wrong signatures will be generated.

3 Our approach

3.1 Why STG Based Signature Can Help?

The fake blending packets mixed in a suspicious flow pool usually do not have many **useful** instruction code embedded in the packet unless they are truly worms. It is found that byte sequences that look like code sequences are highly likely to be dummies (or data) if the containing packet has no code implying function calls [13]. We use semantic analysis to filter those "noisy" padding and substitution bytes and thus improve the signature quality. Under some conditions, the suspicious flow pool can contain no invariants if we compute the frequency of each token by simply counting them. We find the distributions of different tokens are influenced by the positions of the tokens in packets, which are instruction-level exhibitions of semantics and syntax of the packets. In order to capture such semantics, we use different states to express different token distributions in different positions in the packets. It is more robust to the token changes in different positions of the packets, which correlates the tokens' distributions with a state, making token dependency relationships clear. One issue we want to emphasize here is that different from but not contrary to the claim in [23], our model is based on the remaining code extracted from the whole packets instead of on the whole worm packets.

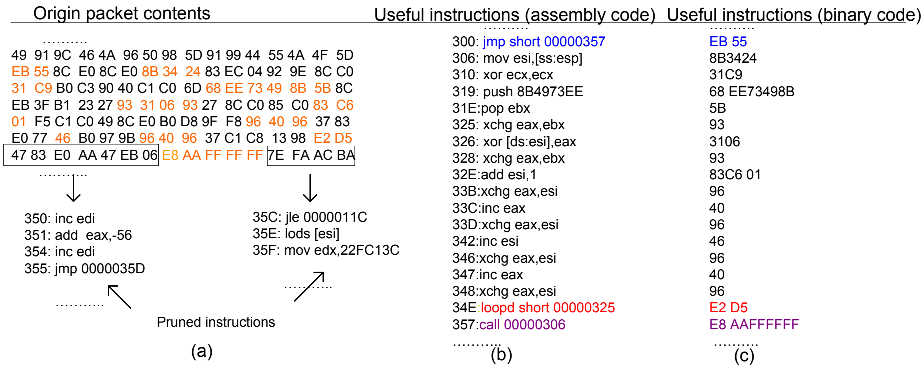


Fig. 3. (a) Original packet contents. (b) Useful instructions (assembly code). (c) Useful instructions (binary code).

3.2 System Overview

In Figure 2, we describe the framework of our approach. Our framework consists of two phases, *semantic-aware signature extraction phase* and *semantic-aware signature matching phase*. The signature extraction phase consists of five modules: payload extraction, payload disassembly, useful instruction distilling, clustering, and signature generation. The signature matching phase is comprised of two modules: payload extraction and signature matching module. **Payload extraction module** extracts the payload which possibly implements the malicious intent, from a a flow which is a set of packets forming a message. For example, in a HTTP request message, a malicious payload only exists in Request-URI and Request-Body of the whole flow. We extract these two parts from the HTTP flows for further semantics analysis. **Disassembly module** disassembles an input byte sequence. If it finds consecutive instructions in the input sequence, it generates a disassembled instruction sequence as output. An instruction sequence is a sequence of CPU instructions which has only one entry point. A valid instruction sequence should have at least one execution path from the entry point to another instruction within the sequence. Since we do not know the entry point of the code when the code is present in the byte sequences, we exploit an improved recursive traversal disassembly algorithm introduced by Wang et al. [13] to disassemble the input. For an N-byte sequence, the time complexity of this algorithm is $O(N)$. **Useful instruction distilling module** extracts useful instructions from the instruction sequences. Useless instructions are identified and pruned by control flow and data flow analysis. **Payload clustering module** clusters the payloads containing similar set of useful instructions together. **Signature generation module** computes STG based signatures from the payload clusters. Upon completion of training, **Signature matching module** starts detecting worm packets by matching STG signatures against input packets. Shortly we will discuss these four modules in detail.

3.3 Useful Instruction Extraction

The **disassembly** module generates zero, one, or multiple instruction sequences, which do not necessarily correspond to real code. From the output of the **disassembly** module, we distill useful instructions by pruning useless instructions. Useless instructions are those illegal and redundant byte sequences using the technique introduced in SigFree [13]. Basically, the pruned useless byte sequences correspond to three kinds of dataflow anomalies: *define-define*, *define-undefine*, and *undefine-reference*. When there is an undefine-reference anomaly (i.e., a variable is referenced before it is ever assigned with a value) in an execution path, the instruction which causes the "reference" is a useless instruction. When there is a define-define anomaly (i.e., a variable is assigned a value twice) or define-undefine anomaly (i.e., a defined variable is later set by an undefined variable), the instruction that caused the former "define" is also considered as a useless instruction. Since normal packets and crafted noisy packets typically do not contain useful instructions, such packets injected in the suspicious flow pool are filtered out after the useful instruction extraction phase. The remaining instructions are likely to be related to the semantics of the code contained in the suspicious packets. An example of polymorphic payload analysis is shown in Figure 3. Here the leftmost part is the original packet content in binary, the middle one is the disassembly code of the useful instructions after removing the useless one, and the rightmost part is its corresponding binaries. For example, in Figure 3, the disassembly code *inc edi* appeared in address *350* is pruned because *edi* is referenced without being defined to produce an *undefine-reference* anomaly.

3.4 Payload Clustering

The useful instruction sequences extracted from polymorphic worms normally contain the following features: (F_1) GetPC: Code to get the current program counter. GetPC code should contain opcode "*call*" or "*fstenv*." We explain the rationale shortly; (F_2) Iteration: Obviously, a polymorphic worm needs to perform iterations over encrypted shellcode. The instructions that can characterize this feature include *loop*, *rep* and the variants of such instructions (e.g., *loopz*, *loope*, *loopnz*); (F_3) Jump: A polymorphic code highly likely to contain conditional/unconditional branches (e.g., *jmp*, *jnz*, *je*); (F_4) Decryption: Since the shellcode of a polymorphic worm is encrypted when it is sent to a victim, a polymorphic worm should decrypt the shellcode during or before execution. We note that certain machine instructions (e.g., *or*, *xor*) are more often found in decryption routine. The reason why we use these four features is that from our observations, nearly all self-modifying polymorphic worm packets contain such features even after complicated obfuscations.

A decryption routine needs to read and write the encrypted code in the payload, therefore, a polymorphic worm needs to know where the payload is loaded in the memory. To our best knowledge, the only way for a shellcode to get the absolute address of the payload is to read the PC (Program Counter) register [15]. Since the IA-32 architecture does not provide any instructions to directly access PC, attackers have to play a trick to obtain the value in the PC register. As far as we know, currently three methods are known in the attacker community: one method uses *fstenv*, and the other two use relative calls to figure out the values in PC.

In a suspicious flow pool, there are normally multiple types of worm packets. For a given packet, we first extract the instructions indicating each of the four features of polymorphic worms. However, simply counting such instructions is not sufficient to characterize a polymorphic shellcode. In reality, some feature may appear multiple times in a specific worm instance, while some others may not appear at all. This makes it complicated for us to match a worm signature to a polymorphic shellcode. If we measure the similarity between a signature and a shellcode based on the bare sequence of the feature identifying instructions, an attacker may evade our detection by distributing dummy features in different byte positions within the payload or by reordering instructions in the execution path. On the other hand, if we ignore the structural (or sequent) order of the feature-identifying instructions and consider them as a histogram, it might result in an inaccurate detection. So in this work we consider both of the structural and statistical informations in packet classification, and use a parameter δ to balance between them.

Specifically, we define two types of distances: (D_1) the feature distance; and (D_2) the histogram distance. We keep the sequent order of the features appearing in an instruction sequence, in a *feature vector*. Let $D_1(v_1, v_2)$ denote the feature distance between two feature vectors v_1, v_2. When v_1 and v_2 are of the same length, we define $D_1(v_1, v_2)$ as the Hamming distance of v_1 and v_2. For example, the feature vector of the instruction sequence shown in Figure 3 is $S = \{F_3, F_4, F_2, F_1\}$. Given another feature vector $S' = \{F_3, F_4, F_1, F_1\}$, the distance between S and S' is computed as $D_1(S, S') = 1$. When two feature vectors are of different lengths, we define the distance of the two feature vectors as $D_1(v_1, v_2) = \max(\text{length}(v_1), \text{length}(v_2)) - \text{LLCS}(v_1, v_2)$, where $\text{LLCS}(v_1, v_2)$ denotes the length of the longest common subsequence of v_1 and v_2 and $\text{length}(v_1)$ denotes the length of v_1. For example, if we are given $S'' = \{F_3, F_4, F_1, F_3, F_1\}$, distance $D_1(S, S'') = 1$. We also measure the histogram distance, the similarity based on the histograms of two feature vectors. Let $D_2(v_1, v_2)$ denotes the histogram distance between two feature vectors v_1, v_2. For example, the histogram of S above is $(1, 1, 1, 1)$ because every feature appears exactly once. Let us assume that the histogram of feature vector S' is given as $(1, 2, 0, 1)$. Then, we define $D_2(S, S')$ as the Hamming distance of S and S', which is 2.

Given two useful instruction sequences, we use both D_1 and D_2 to determine their similarity. We define the distance between two useful instruction sequences as $D = \delta D_1 + (1 - \delta) D_2$, where δ is a value minimizing the clustering error. Suppose there are M clusters in total. Let L_m be the number of packets in cluster m, where m $(1 \leq m \leq M)$ denotes the index of each cluster. When a new packet in a suspicious flow pool is being clustered, we determine whether to merge the packet into an existing cluster or to create a new cluster to contain the packet. We start by calculating the distance between the new packet and every packet in existing clusters. If we find one or more clusters with average distance below threshold θ, we add the new packet to the cluster with the minimum distance among them. Otherwise, we create a new cluster for the new packet. We repeat this process until all packets in the suspicious flow are clustered.

3.5 STG Based Signature Generation

After clustering all the packets in the suspicious pool, we build a signature from each of the clusters. Unlike prior techniques, our signature is based on a state transition graph in which each state is mapped to each of the four features introduced above (Figure 2). In our approach, the tokens (either opcode or operands in a useful instruction sequence)

Algorithm 1 State-Transition-Graph Model Learning Algorithm

Input: A cluster of the useful instructions of the payload $O_{[1..T]}$
Output: STG Signature $\lambda = \{\pi, A, B\}$ for the input cluster
Procedure:

```
map tokens O_t ∈ X(1 ≤ t ≤ T) to the corresponding states S_i ∈ {S_0, S_1, S_2, S_3}(0 ≤ i ≤ N − 1)
calculate initial probability distribution π based on the probabilities of the first token O_1 being on each state S_i ∈
    {S_0, S_1, S_2, S_3} // get π
generate the frequent token set for each state S_i ∈ {S_0, S_1, S_2, S_3} and calculate b_i(k)(1 ≤ k ≤ |X|) // get B
for i = 0 to N − 1 do
    for j = 0 to N − 1 do
        a_ij ← number(O_t ∈ Si ∧ O_{t+1} ∈ Sj) / number(O_t ∈ S_i) // get A, here predicate number denotes the frequency of a token
```

are directly visible. The tokens can be the output of any state, which means each state has a probability distribution over the possible output tokens. For example, in Figure 3, "EB" and "55" are tokens observed in different states. This matches exactly with the definition of Hidden Markov Model (HMM) [24], thus we use HMM to represent the state transition graph for our signature.

More formally, our STG model consists of four states ($N = 4$), which forms state space $S = \{S_0, S_1, S_2, S_3\}$. Let $\lambda = \{\pi, A, B\}$ denote this model, where A is the state transition matrix, B is a probability distribution matrix, and π is the initial state distribution. When a STG model is constructed from a polymorphic worm, we use the model as our STG-based signature. Our STG model is defined as follows:

- State space $S = \{S_0, S_1, S_2, S_3\}$, where state S_0 is the *control flow change state*, which correspond to the feature F_3. State S_1 is the *decryption state*, which corresponds to the feature F_4. State S_2 is the *iteration state*, which corresponds to the feature F_2. State S_3 is the *GetPC state*, corresponding to F_1.
- Transition matrix $A = (a_{ij})_{N\times N} = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix}$ where $a_{ij} = P(next\ state\ is\ S_j | current\ state\ is\ S_i)$, $a_{ij} \in \{S \times S \to [0,1]\}$, and a_{ij} satisfies $\sum_j a_{ij} = 1$ $(0 \leq i,\ j \leq N-1)$.
- Let Y be the set of a single byte and Y^i denote the set of i-byte sequences. $X = \{Y, Y^2, Y^3, Y^4\}$ is the token set in our system because a token in a useful instruction contains at most four bytes (e.g., "*AAFFFFFF*"), which corresponds to the word size of a 32-bit system. Let O_t $(1 \leq t \leq |X|)$ be a token that is visible at a certain state, and $O = \{O_t | O_t \in X\}$ be the visible token set at the state. For a real instruction sequence with T tokens in the useful instruction sequence, the t-length visible output sequence is defined as $O_{[1..t]} = \{O_1, O_2, ..., O_t\}$ $(t \leq T)$. Then, we can define the probability set B as $B = \{b_i(k)\}$, where $b_i(k) = P(visible\ \ token\ \ is\ O_k | current\ \ state\ \ is\ \ S_i)$. $b_i(k)$ is the probability of X_k on state S_i, thus satisfying $\sum_{1 \leq k \leq |X|} b_i(k) = 1$.
- Initial state distribution $\pi = \{\pi_0, \pi_1, \pi_2, \pi_3\}$, where $\pi_i = P(the\ first\ state\ is\ S_i)$.

Algorithm 1 is adopted from the segment K-means algorithm [24] to learn the structure of Hidden Markov Model. As the same token can appear at different states with

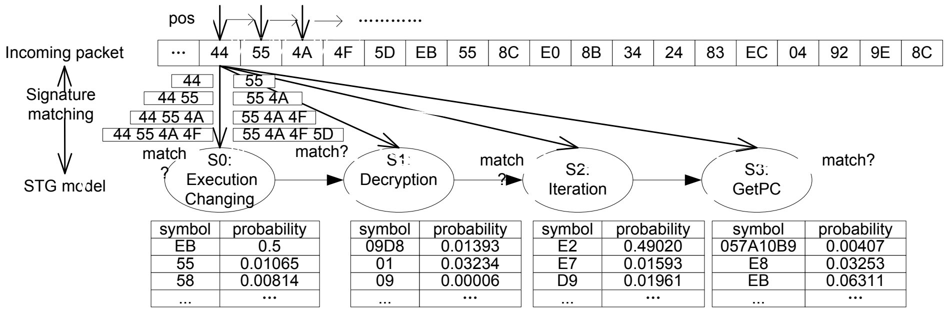


Fig. 4. ***STG*** signature matching process

different probabilities, we manage our model to satisfy $O_t \in S_i$ if $b_i(O_t) > b_j(O_t)$ for all $j \neq i$ (step 2 and step 3). We also remove noises by setting a threshold to discard less-frequent tokens. For example, if $\max_i b_i(O_t)$ is below the threshold (e.g.,θ_0), we ignore this token O_t while constructing a STG model.

3.6 Semantics Aware Signature Matching Process

After we extract STG-based signatures from the suspicious flow pool, we can use them to match live network packets. Given a new packet to test, our detector first retrieves the payload of the packet. Assuming that the payload length is *m* bytes, the detector checks whether this *m*-byte payload matches any of the existing signatures. If it does not match any signature (i.e., their deviation distance is above a threshold θ_2), it is considered as a benign packet. If it matches a single signature of certain type of a worm, it will be classified as the type of worm associated with the signature. If the packet matches multiple signatures, the detector classifies the packet as the one with the smallest distance among the matching signatures. An advantage of our approach is that we need not make complicated analysis on the live packets but match the packets byte after byte.

To measure the distance between an *m*-byte (input) payload and a signature, we try to identify the first token, starting from the first byte of the payload. We form four candidate tokens of length *i* (*i*=1, 2, 3, 4), where the *i*-th candidate token consists of the first *i* bytes of the payload. As is shown in Figure 4, we first select (44), (44,55), (44,55,4A), (44,55,4A,4F) as the candidate tokens. Then, for each candidate token O_t, we calculate its probability to appear in each of the four states in our STG model, and assign it to the state which gives the largest probability $b_i(O_t)$. Let $max(P_i)$ denotes the maximum of the four $b_i(O_t)$. If $max(P_i)$ is above a threshold θ_1, we choose the candidate token yielding $max(P_i)$ as the real token, and ignore the others. Otherwise, all of the four candidate tokens are ignored. In either case, we move to the next four bytes of the payload. As for the case in Figure 4, we will start to check the next four tokens (55), (55,4A), (55,4A,4F), (55,4A,4F,5D). We will repeat the above process until all *m* bytes are processed. Finally, we sum up the $max(P_i)$ and calculate its distance from the signature. The deviation distance D is defined as $D = ||\log P[O_{[1...m]}|\lambda] - mean||$ where $\log P[O_{[1...m]}|\lambda]$ is the matching probability value for a *m*-byte packet, $mean$ is an average matching value for a certain type of training packets. Assuming that

Table 1. Comparison of SAS with Polygraph and Hamsa

Comparison	Polygraph	Hamsa	SAS
Content (behavior) detection	content	content	content
Semantic related	semantic free	semantic free	semantic related
On-line detection speed	fast	fast	fast
(Crafted) noise tolerance	some	medium	good
Token-fit attack resilience	nearly no	medium	good
Coincidental attack resilience	nearly no	medium	good
Allergy attack resilience	some	some	good
Signature simplicity	simple	simple	complicated

there are l packets in a cluster of the same type, and the byte length for each packet is T_i $(1 \leq i \leq l)$, we have $mean = \frac{1}{l} \sum_{k=1}^{l} \log P[O_{[1\ldots T_k]}|\lambda]$. We do not show the detailed algorithm here due to the limited space.

4 Security Analysis

4.1 Strength

Our semantic based signatures can filter the noises in the suspicious flow pool and prune the useless instructions which are otherwise possibly learned as signature, thus it has good noise tolerance. As the STG signature is more complicated than previous signatures (e.g., token-sequence signature), it is much harder for attackers to ruin our automatic signature generation by crafting packets bearing both the tokens of normal and attack packets compared with previous signatures. Moreover, even if the hackers change the contents of the attack packets a lot, they can hardly evade our detection since our signature is not based on syntactic patterns but based on semantic patterns. In addition, the STG signature can match unknown polymorphic worms (which our detector has not been trained with) since it has learned certain semantics of the decryption routine from existing polymorphic worms. Our STG signature matching algorithm introduces low overhead (analysis throughput is more than 10Mbps), thus our detector is fast enough to match live packets. Some anti-disassemble techniques like junk byte insertion, opaque predicate, and code overlap all aim to immobilize linear sweep disassembly algorithms. The disassembler of the STG signature generation approach is a recursive traversal algorithm, which makes our approach robust to such types of anti-disassemble techniques. In Table 1, we summarize our benefit in comparison with other signature generation approaches. For STG, it is robust to the attacks filling crafted bytes in the wildcard bytes of the packets (e.g., coincidental-pattern attack [5] and the token-fit attack [6]) since these packets usually fail to pass our semantic analysis process. It is robust to the innocuous pool poisoning [5] attack and allergy attack [7] because our technique can filter the normal packets out for signature generation. As it is a probability based algorithm, the long-tail attack [5] will not thwart our matching process. Finally, by discovering meanings of each token (i.e., which token is exhibiting which feature), our approach explores beyond traditional signatures which leverage only the syntactic patterns to match worm packets.

Useful instruction (assembly code)	Bianry code
....	....
sub cl,2	80E9 02
dec ecx	49
dec ecx	49
je short 00000261	74 07
jmp short 00000206	EB AA
call 00000201	E8
....	A0FFFFFF
	....

Fig. 5. STG signature example. The bytes used by the signature are marked in red color.

4.2 Limitations

Here we discuss about the limitations of the proposed technique and possible methods to mitigate these limitations. First, based on static analysis which can not handle some state-of-the-art code obfuscation techniques (e.g., branch-function obfuscation, memory access obfuscation), we can not generate appropriate signatures if the semantic analysis fails to analyze the suspicious flow pool. This can be solved through more sophisticated semantic analysis such as symbolic execution and abstract interpretation techniques. Second, our technique can be evaded if smart attackers use more sophisticated encryption and obfuscation techniques such as doubly encrypted shellcode with invariant substitution. Also, for the non self-contained code [16], there may be absence of features for clustering to generate the signatures. To address these issues, emulation-based payload analysis techniques can be used in the signature extractor and the attack detector, however, state-of-the-art emulation-based techniques are still lack of performance to be used in a live packet analysis. Although one may doubt the utility of byte-level signatures (e.g., it could not handle the packed code), its performance is good for practical deployment compared with the emulation based approaches.

5 Evaluation

We test our system offline on massive polymorphic packets generated by real polymorphic engines used by attackers (i.e., CLET, ADMmutate, PexFnstenvMov) and on normal HTTP request/reply traces collected at out lab PCs. Both CLET and ADMmutate are advanced polymorphic engines which obfuscate the decryption routines by metamorphism such as instruction replacement and garbage insertion. CLET also uses spectrum analysis to counterattack the byte distribution analysis. PexFnstenvMov is a polymorphic engine included in Metaspoit [25] framework. Opcode of the "*xor*" instruction is frequently found in the decryption routine of PexFnstenvMov. PexFnstenvMov also uses the "*fnstenv*" instruction for the GetPC code.

In evaluation, we also use 100,000 non-attack HTTP requests/responses for two purposes: to compute false positive rate and to derive noisy flows to attack signature extraction. The normal HTTP traffic contains 100,000 messages collected for three weeks at seven workstations owned by seven different individuals. To collect the traffic, a client-side proxy monitoring incoming and outgoing HTTP traffic is deployed underneath the web server. Those 100,000 messages contain various types of non-attack data including JavaScript, HTML, XML, PDF, Flash, and multimedia data, which render diverse and realistic traffic typically found in the wild. The total size of the traces is over 1.77GB. We run our experiments on a 2.4GHz Intel Quad-Core machine with 2GB RAM, running Windows XP SP2.

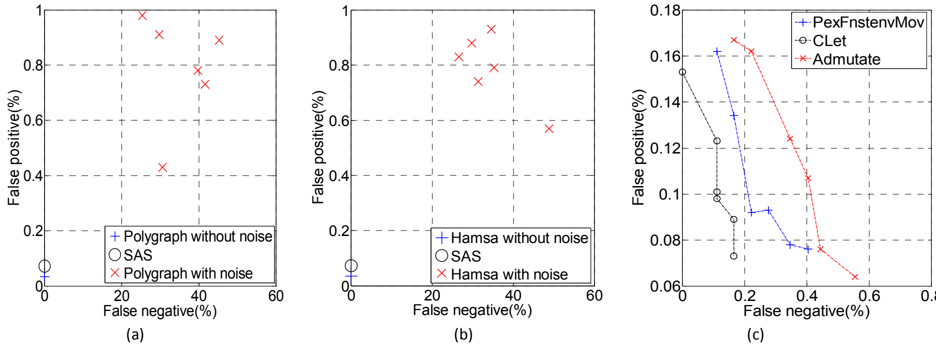


Fig. 6. (a) Comparison of SAS and Polygraph (b) Comparison of SAS and Hamsa (c) Impact of parameters

5.1 Comparison with Polygraph and Hamsa

In this section, we evaluate the accuracy (in terms of false positives and false negatives) of our algorithm in comparison with Polygraph and Hamsa. We compare the three systems in two cases: without noise injection attack, with noise injection attack.

Parameter Settings The parameters of Polygraph are set as follows. The minimum token length α is set to 2, the minimum cluster size is set to 2, and the maximum acceptable false positive rate during the signature generation process is set to 1%. Hamsa in our experiments is built from the source that we downloaded from the Hamsa hompage. The minimum acceptable false positive rate of Hamsa is set to $u = 0.01$ during the signature generation process. In our approach, the parameters θ_0,θ_1 are used to prune the tokens which have little probability to match with the STG signature; and parameter θ_2 is used to label the deviation distance during the packet matching process. These parameters are configured as follows: $\theta_0 = 0.016$, $\theta_1 = 0.016$, $\theta_2 = 12.000$.

Polymorphic Engine In this experiment, we use CLET because it implements spectrum analysis to attack the byte distribution analysis performed by existing signature extractors. We generate 1,000 worm instances from CLET, among which 400 instances are used as the training data to generate signatures, and 600 instances are used to compute the false negative rate. We also use 100,000 non-attack HTTP requests/responses to compute false positive rate.

Comparison Without Noise Injection We compare our method with Polygraph and Hamsa, without considering noise injection. Fed with the same 400 attack messages, the signatures generated by Hamsa and the conjunction signature generated by Polygraph are all '\x8b':1,'\xff\xff\xff':1,'\x07\xeb':1. The state transition path of our signature is $(S_0 \rightarrow S_1 \rightarrow S_0 \rightarrow S_3)$. Token sequences '\xff\xff\xff' and '\x07\xeb' are the only invariant tokens appearing in the useful instruction sequences (Figure 5).

Comparison Under Noise Injection We compare SAS, Polygraph, and Hamsa, assuming 1:1 attack-to-noise ratio in the suspicious flow pool. To add the crafted noise to the suspicious flow pool, we adopt the method used by Perdisci et. al. [9]. For each malicious packet w_i, we create the associated fake anomalous packet f_i by modifying the corresponding packet w_i. The way to make crafted noisy packet f_i is divided into the following six steps. (Step 1) f_i^0: create a copy of w_i. (Step 2) f_i^1: permute bytes f_i^0

randomly. (Step 3) $a[\,]$: copy k substrings of length l from w_i to array a, but do not copy the true invariant. (Step 4) f_i^2: copy the fake invariant substring into f_i^1. (Step 5) f_i^3: inject m-length substring of string v into (f_i^2), we generate n ($n > m$) bytes of string $v = \{v_1, v_2, ..., v_n\}$ by selecting the contiguous bytes in the innocuous packet which satisfy $0.05 < \mathrm{P(v|innocuous \quad packets)} < 0.20$. (Step 6) f_i^4: obfuscate the true invariant by substituting the true invariant bytes in the packet.

To craft non-attack derived noises, we use our 10,000 normal HTTP messages. The suspicious flow pool are composed of 400 CLET-mutated instances and 400 crafted noises. We configure the parameters of noise generator as $k = 3$, $l = 5$, $n = 6$, and $m = 3$. The parameters for SAS, Polygraph, and Hamsa are set as the same as in Case-1.

When we compare SAS with Polygraph, we ignore “true invariants” in Step 3 and 6 because we do not know the true invariants until the signature is generated. Instead, we permute the bytes more randomly to separate and distribute contiguous bytes before copying substrings of w_i in Step 3. Atop this, we use even more sophisticated noise injection when we compare SAS with Hamsa. Specifically, in Step 5, we choose a string v which satisfies $\mathrm{P(v|innocuous \quad packets)} < \mathrm{u}$ (u is parameter). Since we set u as described above, Hamsa’s false positive rate will not exceed u even if the injected noises are taken as signatures.

Comparison Results Figure 6(a) and Figure 6(b) show the false positive and false negative rates of SAS, Polygraph, and Hamsa in both experiment cases. In Case-1 experiment (i.e., without noise injection), all the three systems show similar accuracy. Although SAS shows slightly higher false positive rate than Polygraph and Hamsa, the false positive rates of all three systems are already very low (< 0.0008). In Case-2 experiment (i.e., with noise injection), the false positive and the false negative rates of SAS has not been affected by the crafted noise injected to the suspicious flow pool. In contrast, the signature generation process of Polygraph and Hamsa has been greatly misled to add fake invariants taken from the crafted noises, which results in extremely high false negative rate. As a result, the signatures generated by Polygraph and Hamsa miss more than 20% of attack messages. The false positive and false negative rates of Polygraph and Hamsa are still lower than 1%, which is because they have a threshold of maximum false positive rate (say 1%).

5.2 Per-Polymorphic Engine Evaluation

In this experiment, we evaluate the impact of different parameter settings on our approach. We use 3,000 worm instances generated by CLET, Admutate, and PexFnstenvMov. We feed our signature extractor with 1,200 out of the 3,000 worm instances (400 instances from each type of worm) to generate STG-based signatures. Then, we use the remaining worm instances to evaluate the extracted signatures. We also inject 10,000 normal packets into the suspicious flow pool to make the packet clustering more difficult.

The parameter δ is first set to an initial value, and then adjusted until all the packets are clustered correctly. In our setting, we find the structural information is more important than statistical information. When we set parameter $\delta = 0.8$, all the packets in the suspicious pool are grouped in the right cluster. We aim to find an appropriate δ for the

Table 2. Accuracy of the STG-based signatures generated by SAS

Polymorphic engine	False positive	False negative	State transition path of STG-based signature
PexFnstenvMov	0.075%	0.40%	$(S_3 \rightarrow S_1 \rightarrow S_2)$
CLet	0.072%	0.42%	$(S_0 \rightarrow S_1 \rightarrow S_0 \rightarrow S_3)$
Admutate	0.062%	0.55%	$(S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3)$

Table 3. Performance evaluation

Polymorphic engine	Training time(sec)	Matching time(sec)	Analysis throughput(Mbps)
PexFnstenvMov	22.901	1.783	10.534
CLet	31.237	2.879	13.655
Admutate	24.833	1.275	12.901

right clustering. The δ can be tuned based on the feedback of clustering result. We test the false negative and false positive rates using the remaining 1,800 attack instances and the 100,000 normal HTTP messages respectively (Table 2). We also evaluate the influence of parameter changes on signature matching as shown in Figure 6(c), where each data point stands for one group of parameter settings. We change the value of θ_1, θ_2 to see how false positive rate and false negative rate vary. In our experiment, we observe the lowest false positve rate when we set the parameters as $\theta_1 = 0.018$, $\theta_2 = 12.000$. Altough we do not present entire results due to page limit, our experiment results show that the false negative rate decreases as θ_1 increases. Also, the false positive rate increases as θ_2 increases. These observations are confirmed from the design of our Algorithm, as higher θ_1 would filter more noises while a higher θ_2 would block more normal packets. The parameters, in practice, can be tuned based on the feedback from training and testing datasets, so that we get a locally optimized false positive rate. Table 2 shows the best configurations obtained by the above method.

5.3 Performance Evaluation

The time complexity of the signature learning algorithm is $O(N^2TP)$, where T is the length of token sequence, P is the number of the suspicious packets in a clustering, and N is the number of states. The time complexity of our signature matching algorithm is $O(N^2S \cdot L)$, where L is the average length of token sequences in a signature, S is the total length of input packets to match, N is the number of states. The signature matching algorithm can be easily adapted to satisfy the requirements of online detection. The training time, matching time, and analysis throughput for each polymorphic engine are shown in Table 3. The training time includes the time to extract useful instructions from packets. The matching time is the total elapsed time to match 600 mutations generated by each polymorphic engine.

6 Related Work

Pattern Extraction Signature Generation There are a lot of work on pattern based signature generation, including honeycomb [3], earlybird [4], and autograph [2], which had been shown not to be able to handle polymorphic worms. Polygraph [5] and Hamsa [6] are pattern based signature generation algorithms, and they are more capable of detecting polymorphic worms, but vulnerable to different kinds of noise injection attacks. There are also rich researches on attacks against pattern-extraction algorithms. Perdisci et al. [9] present an attack which adds crafted noises into the suspicious flow to confuse the signature generation process. Paragraph [8] demonstrates that Polygraph and Hamsa are vulnerable to attacks as long as attackers can construct the labeled samples randomly to mislead the training classifier, and this attack can also prevent or severely delay generation of an accurate classifier. Allergy attacks [7] force the signature generation algorithm to generate signatures that could match the normal traffic, thus introducing high false positive rate. Gundy et al. [26] present a class of feature omission attacks on signature generation process that are poorly addressed by Autograph and Hamsa. Polymorphic blending attacks [27] are presented by matching the byte frequency statistics with normal traffic to evade detection. Theoretical analysis of limits of different signature generation algorithms are given in [28]. Gundy et al. [29] show that web based polymorphic worms do not necessarily have invariant bytes. A game-theoretical analysis on how a detection algorithm and an adversary could adapt to each other in an adversarial environment is introduced in [30]. Song et al. [23] studied the possibility of deriving a model for representing the general class of code that corresponds to all possible decryption routines, and concludes that it is infeasible. Our work combines the semantic analysis with the signature generation process, making it robust to many noise-injection attacks (e.g., allergy attack, red herring attack).

Semantic Analysis Researches have presented semantic based techniques by making static and dynamic analysis on the binary code. Polychronakis et al. [16] have presented emulation-based approach to detect polymorphic payloads by emulating the code and detecting decryption routines through dynamic analysis. Libemu [17] is another attempt to achieve shellcode analysis through code emulations. Compared with their works, our approach has higher throughput and can not be attacked by anti-emulation techniques. Brumley et al. [31] propose to automatically create vulnerability signatures for software. Cover [10] exploits the post-crash symptom diagnosis and address space randomization techniques to extract signatures. TaintCheck [11] exploits dynamic dataflow and taint analysis techniques to help find the malicious input and infer the properties of worms. ABROR [12] automatically generates vulnerability-oriented signatures by identifying typical characteristics of attacks in different program contexts. Sigfree [13] detects the malicious code embedded in HTTP packets by disassembling and extracting useful code from the packets. Spector [14] is a shellcode analysis system that uses symbolic execution to extract the sequence of library calls and low-level execution traces generated by shellcode. Christodorescu et al. [32] present a malware detection algorithm by incorporating instruction semantics to detect malicious program traits. Our motivation is similar, but our work is specific to network packet analysis instead of for file virus. STIIL [15] uses static taint and initialization analysis to detect exploit code embedded in data streams/requests targeting at web services. Kruegel et al. [33] present a technique based on the control flow structural information to identify the structural similarities between different worm mutations. Contrast to their work, our work is to generate signatures based on semantic and statistic analysis.

7 Conclusion

In this paper, we have proposed a novel semantic-aware probability algorithm to address the threat of anti-signature techniques including polymorphism and metamorphism. Our technique distills useful instructions to generate state transition graph based signatures. Since our signature reflects certain semantics of polymorphic worms, the proposed signature is resilient to the noise injection attacks to thwart prior techniques. Our experiment have shown that our approach is both effective and scalable.

Acknowledgments The authors would like to thank Dinghao Wu for his help in revising the paper. The work of Zhu was supported by CAREER NSF-0643906. The work of Jhi and Liu was supported by ARO W911NF-09-1-0525 (MURI), NSF CNS-0905131, AFOSR FA 9550-07-1-0527 (MURI), NSF CNS-0916469, and AFRL FA8750-08-C-0137. The work of Kong, Xi was supported by Chinese High-tech R&D (863)Program 2006AA01Z449, China NSF-60774038.

References

1. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: Proceedings of the 23rd Anual Computer Security Applications Conference. (2007)
2. Kim, H.A., Karp, B.: Autograph: Toward automated, distributed worm signature detection. In: Proceedings of the 13th Usenix Security Symposium. (2004)
3. Kreibich, C., Crowcroft., J.: Honeycomb: creating intrusion detection signatures using honeypots. In: Proceedings of the Workshop on Hot Topics in Networks (HotNets). (2003)
4. Singh, S., Estan, C., Varghese, G., Savage, S.: Earlybird system for real-time detection of unknown worms. Technical report, Univ. of California at San Diego (2003)
5. Newsome, J., Karp, B., Song, D.: Polygraph: Automatic signature generation for polymorphic worms. In: IEEE Symposium on Security and Privacy. (2005)
6. 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: IEEE Symposium on Security and Privacy. (2006)
7. Chung, Mok, A.K.: Advanced allergy attacks: Does a corpus really help. In: Recent Advances in Intrusion Detection (RAID), pages 236-255. Springer-Verlag. (2007)
8. Newsome, J., Karp, B., Song., D.: Paragraph: Thwarting signature learning by training maliciously. In: Recent Advances in Intrusion Detection (RAID), pages 81-105. Springer-Verlag. (2006)
9. Perdisci, R., Dagon, D., Lee, W.: Misleading worm signature generators using deliberate noise injection. In: Proceedings of The 2006 IEEE Symposium on Security and Privacy. (2006)
10. Liang, Z., Sekar., R.: Fast and automated generation of attack signatures: A basis for building self-protecting servers. In: Proceedings of the 12th ACM Conference on Computer and Communications Security. (2005)
11. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proceedings of Network and Distributed System Security Symposium. (2005)
12. Liang, Z., Sekar., R.: Automatic generation of buffer overflow attack signatures: An approach based on program behavior models. In: Proceedings of the Anual Computer Security Applications Conference. (2005)

13. Wang, X., Pan, C.C., Liu, P., Zhu, S.: Sigfree: A signature-free buffer overflow attack blocker. In: 15th Usenix Security Symposium. (2006)
14. Borders, K., Prakash, A., Zielinski., M.: Spector :automatically analyzing shell code. In: Proceedings of the 23rd Annual Computer Security Applications Conference, pages 501-514. (2007)
15. Wang, X., Jhi, Y.C., Zhu, S., Liu, P.: Still: Exploit code detection via static taint and initialization analyses. In: Proceedings of Anual Computer Security Applications Conference (ACSAC). (2008)
16. Krügel, C., Lippmann, R., Clark, A.: Emulation-based detection of non-self-contained polymorphic shellcode. In: Recent Advances in Intrusion Detection, 10th International Symposium. Volume 4637 of Lecture Notes in Computer Science., Springer (2007)
17. Baecher, P., Koetter, M.: Getting around non-executable stack (and fix). (http://libemu.carnivore.it/)
18. Szor, P.: The Art of Computer Virus Research and Defense,pages 112-134. Addison-Wesley (2005)
19. Bania, P.: Evading network-level emulation. (http://www.packetstormsecurity.org/papers/bypass/pbania-evading-nemu2009.pdf)
20. Collberg, C., Thomborson, C., Low., D.: A taxonomy of obfuscating transformations. In: Technical Report 148,University of Auckland. (1997)
21. Detristan, T., Ulenspiegel, T., Malcom, Y., Superbus, M., Underduk, V.: Polymorphic shellcode engine using spectrum analysis. (http://www.phrack.org/show.php?p=61&a=9)
22. Ray, E.: Ms-sql worm. (http://www.sans.org/resources/malwarefaq/ms-sql-exploit.php)
23. Song, Y., Locasto, M.E., Stavrou, A., Keromytis, A.D., Stolfo., S.J.: On the infeasibility of modeling polymorphic shellcode. In: Proceedings of the 14th ACM conference on Computer and communications security(CCS), pages 541-551. (2007)
24. Rabiner, L.R.: A tutorial on hidden markov models and selected applications in speech recognition. In: Proceedings of the IEEE, 77 (2): pages 257-286. (1999)
25. Moore, H.: The metasploit project. (http://www.metasploit.com)
26. Gundy, M.V., Chen, H., Su, Z., Vigna, G.: Feature omission vulnerabilities: Thwarting signature generation for polymorphic worms. In: Proceeding of Annual Computer Security Applications Conference (ACSAC). (2007)
27. Fogla, P., Sharif, M., Perdisci, R., Kolesnikov, O., Lee, W.: Polymorphic blending attacks. In: Proceedings of The 15th USENIX Security Symposium. (2006)
28. Venkataraman, S., Blum, A., Song., D.: Limits of learning-based signature generation with adversaries. In: Proceedings of the 15th Annual Network and Distributed System Security Symposium. (2008)
29. Gundy, M.V., Balzarotti, D., Vigna, G.: Catch me, if you can: Evading network signatures with web-based polymorphic worms. In: Proceedings of the First USENIX Workshop on Offensive Technologies (WOOT) Boston,MA. (2007)
30. Pedro, N.D., Domingos, P., Sumit, M., Verma, S.D.: Adversarial classification. In: 10th ACM SIGKDD Conference On Knowledge Discovery and Data mining, pages 99-108. (2004)
31. Brumley, D., Caballero, J., Liang, Z., Newsome, J., Song., D.: Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In: Proceedings of the 16th USENIX Security. (2007)
32. Christodorescu.M, Jha.S, Seshia, S., Song.D, Bryant, R.: Semantics-aware malware detection. In: 2005 IEEE Symposium on Security and Privacy. (2005)
33. Krugel, C., Kirda, E.: Polymorphic worm detection using structural information of executables. In: 2005 International Symposium on Recent Advances in Intrusion Detecion. (2005)