SECURITY AND COMMUNICATION NETWORKS
Security Comm. Networks (2012)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/sec.613

RESEARCH ARTICLE

Semantic aware attribution analysis of remote exploits

Deguang Kong[1,2]*, Donghai Tian[1], Qiha Pan[3], Peng Liu[1] and Dinghao Wu[1]

[1] College of Information Sciences and Technology, Pennsylvania State University, University Park, PA 16802, U.S.A.
[2] Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX, 76013, U.S.A.
[3] Department of Electrical Engineering, Pennsylvania State University, University Park, PA, 16802, U.S.A.

ABSTRACT

Web services have been greatly threatened by remote exploit code attacks, where maliciously crafted HTTP requests are used to inject binary code to compromise web servers and web applications. In practice, besides detection of such attacks, attack attribution analysis (i.e., to automatically categorize exploits or determine whether an exploit is a variant of an attack from the past) is also very important. In this paper, we present SA^3, a novel exploit code attribution analysis that combines semantics-based analysis and statistical modeling to automatically categorize given exploit code. SA^3 extracts semantic features from exploit code through data anomaly analysis and then attributes the exploit to an appropriate class on the basis of our statistical model derived from a Markov model. We evaluate SA^3 over a comprehensive set of shellcode collected from Metasploit and other polymorphic engines. Experimental results show that SA^3 is effective and efficient. The attribution analysis accuracy can be over 90% in different parameter settings with false positive rate no more than 4.5%. The novelty of SA^3 is that it combines semantic analysis with statistical modeling for exploit code attribution analysis.

KEYWORDS

malware classification; remote exploit; shellcode; attribution; mixture Markov model

***Correspondence**

Deguang Kong, College of Information Sciences and Technology, Pennsylvania State University, University Park, PA 16802, U.S.A.
E-mail: dkong@ist.psu.edu

1. INTRODUCTION

A great number of code injection attacks (e.g., buffer overflow attacks, format string attacks) are used by crafted HTTP requests to compromise different kinds of web services or web applications. From the CERT [1] and SecurityFocus [2] statistics, the remote code injection attack is still one of the major attacks these days. In remote code injection attacks, malicious HTTP requests/replies can be forged to inject malicious code by masquerading as normal requests/replies. Different kinds of shellcodes are representatives of exploit code, which can be injected into target services or applications through network connections. Worms can take advantage of these exploit code for infections and propagations. In this paper, the exploit code we focus on is remote shellcode, which can be used as the payload of a packet to spread via HTTP requests. Throughout the paper, we use the terms remote exploit code and shellcode interchangeably.

There are mainly two types of techniques used for shellcode analysis and detection: the emulation-based approach and statistics-based approach. The emulation-based approach (e.g., [3,4]) emulates the executions of instruction sequences, and thus, shellcode's behaviors are exposed in the virtual running environment. However, it can be antagonized by many kinds of anti-emulation techniques [5]. For example, drive-by-downloads web attacks [6], which target memory corruption vulnerabilities, have to prepare the environment before their successful launch. Improper emulations of the execution context will lead to incorrect executions of instruction sequences and thus fail to expose specific behaviors.

Statistical analysis is another promising method used in network intrusion detection systems including the remote shellcode detection and analysis [7,8]. The basic idea of the statistical approach is to extract distinguished features to differentiate between normal packets and various malicious packets. The payload of a packet and the payload header information (e.g., port number, protocol field) can be used as features for classification. The disadvantage of the statistical approach is that it usually lacks clear semantic information correlated with the packets whose contents may result in malicious behaviors, and therefore, it can also be evaded by different kinds of anti-statistic techniques [7,8].

From the aforementioned analysis, we can see that current exploit code detection and analysis techniques still have limitations. Meanwhile, lots of shellcode variants

appeared in the past several years according to AV-test's statistics [9]. Thus, in this paper, we present an automatic semantic aware attribution analysis of remote exploits.

Attribution analysis presented in this paper is to solve the problem of automatically categorizing exploits and determining whether an exploit is a variant of an attack from the past. The attribution analysis can be used in, for example, a shellcode scanner to identify different types of shellcode variants. We give more formal definition of attribution analysis problem in Section 3. The significance of this work is that it provides more information about an attack in addition to detecting the attack. We believe that this is also important besides telling whether a piece of code is malicious or not.

As far as we know, such shellcode attribution analysis is still lacking in the literature. Note that Hu *et al*. [10] present a function-call graph-based approach to index the large malware repositories, which can be viewed as a kind of malware attribution analysis. Ma *et al*. [11] present a methodology for inferring the phylogeny (i.e., evolution tree) of remote code injection exploits. Our motivation is similar to theirs, but the differences are also very clear: (1) our work is more specific for exploit code attribution analysis, and (2) we emphasize the combinations of semantics and statistics for solving the attribution analysis problem, which has not been done before.

Exploit code attribution poses several challenges. First, the emulation-based approach cannot be directly applied to this problem because we need quantitative metrics to measure the distances of different exploit code. Second, we cannot fully rely on the statistical approach because it is deceptive once the statistical features (e.g., the number of specific instructions or system calls) fail to reflect the security-critical operations, which are probably highly related with the shellcode behaviors. Third, how to extract the semantics, which determine the shellcode attribution, remains an open question. The emulation-based approach seems a good candidate for extracting the behaviors of different shellcode. However, it can miss trivial differences that existed in the behaviors of different classes of shellcode. For example, self-contained exploit code [4] often exhibits same behaviors by following the routine of "decrypt-loop" mode. Furthermore, if specific behaviors are absent in the emulation environment, it could produce more false negatives. Also, the time cost for the emulation-based approach is usually very high compared with static analysis.

1.1. Our approach

We present SA^3, a novel automatic semantic aware attribution analysis of remote exploit code. SA^3 first makes semantic analysis on the payload of packets, and then, a Markov-based model is used to model each type of shellcode. Specifically, for semantic analysis, we use static data anomaly analysis on the packet payload; for statistic analysis, we use a two-way of mixture Markov model. The statistical model is based on the refined exploit code sequences, which are pruned from the whole code sequences in the framework of static analysis. Once the model is built, any new code can be fed into the model to get an attribution analysis result.

Our approach is based on an observation that certain control and data flow information is preserved in exploit code. Thus, in our work, we use the preserved semantic-related "features" obtained from semantic analysis module for attribution analysis. We note that the attribution for a piece of exploit code has great correlations with the exploit code's semantic characteristics (e.g., the opcode sequence, the instruction sequence) and also its statistical characteristics (e.g., the number of instructions, the out-degree of control flow graph). The changes of the semantics also cause the changes of the statistic exposure in shellcode instructions. These observations motivate us to consider about the integration of semantic analysis with statistical analysis by taking advantages of both of them.

Our work stands between the semantic analysis and statistical analysis. Instead of using dynamic emulation techniques introduced before, our work uses the static data anomaly analysis by making static analysis on the instruction sequences. The advantage of this approach is that it can capture the semantics of the exploit code with moderate time cost. Also, it does not suffer from attacks by anti-emulation techniques [5]. Compared with only emulation-based approach, our work can also overcome some inherent defects (e.g., different shellcode may expose similar behaviors) by introducing the statistical analysis. Compared with only statistics-based approach, our analysis is more robust by incorporating the semantics to avoid "black-box" learning.

1.2. Contributions

The main merits of SA^3 are listed as follows. The novelty of our work is that it combines semantic analysis with statistical modeling for exploit code attribution analysis. Semantic analysis is used to extract the semantic-binding code with certain malicious intent. Statistical features can help capture the "whole" view of a packet from macroscopic point. These two different views complement each other. Our evaluation shows that our analysis result is better than purely statistical approach, which also refutes the conclusion of "impossibility of modeling polymorphic shellcode [12]" in some degree.

The rest of this paper is organized as follows. First, we introduce the related work in Section 2. Then, we formalize the problem in Section 3. Next, we present our approach SA^3 in Section 4, followed by the detailed introduction of semantic module and statistical module in Sections 5 and 6. In Section 7, we evaluate our approach by experiment and discuss about the advantages and disadvantages of our method. Finally, we conclude the paper in Section 8.

2. RELATED WORK

There is a large body of work in the area of exploit code analysis and detection. We focus on two areas most related to our work: semantics-based approaches for malware especially exploit code analysis and statistics-based approaches for malware analysis.

2.1. Semantics-based approaches

Malware including exploit code analysis has received considerable attention from different research views. Various kinds of semantic techniques have been explored by making static or dynamic analysis on the binary code for malware detection. Emulation-based approaches [4,13] can be used to detect polymorphic shellcode by emulating the code execution to recognize specific behaviors (e.g., decryption routines) through dynamic analysis. Libemu [3] is another attempt to achieve shellcode analysis through code emulations. Gu *et al*. [14] presented a new malicious shellcode detection methodology by analyzing snapshots of the processs virtual memory before input data are consumed. However, these emulation-based techniques can be antagonized by many anti-emulation techniques [5]. In our work, we use the static data anomaly techniques introduced in SigFree [15] to extract the semantics from the malicious code sequences.

Christodorescu *et al*. [16] presented a dependency-graph-based approach to mining the malicious behaviors present in a known malware that are not present in a set of benign programs, which can be used by malware detectors to detect malware variants. Also, Christodorescu *et al*. [17] used trace semantics to characterize the behaviors of malware as well as the program being checked for infection and used abstract interpretation to "hide" irrelevant aspects of these behaviors for malware detection and classification. However, these template-based approaches, which use some unification process between the program variables and malware symbolic variables, can only handle a limited set of obfuscations commonly used by malware writers. What we have presented here is specially tailored to classifying polymorphic code, which is orthogonal to malware classification in the sense that the obfuscated payload could be from an arbitrary known or unknown family of malware.

Another similar work to the semantic module we use is STIIL [18], which uses static taint and initialization analysis to detect exploit code embedded in data streams and requests targeting web services. Spector [19] is a shellcode analysis system that uses symbolic execution to extract the sequence of library calls and low-level execution traces generated by shellcode. TaintCheck [20] exploits dynamic dataflow and taint analysis techniques to help find the malicious input and infer the properties of worms. Kruegel *et al*. [21] present a technique based on the control flow structural information to identify the structural similarities between different worm mutations. This work is close to our technique in that it analyzes the variants of worms, but they target worms, not exploit code.

2.2. Statistics-based approaches

Song *et al*. [12] studied the possibility of deriving a model for representing the general class of code that corresponds to all possible decryption routines and concluded that it is infeasible. Our work combines the semantic analysis and statistical analysis for exploit code attribution analysis, making it robust to many noise-injection attacks (e.g., allergy attack [22]).

Ma *et al*. [11] analyzed the diversity of remote code injection exploits by inferring the phylogeny (i.e., evolution tree) of them. They use agglomerative clustering techniques to capture the inter-family and intra-family relationships of different exploits. Our work is different from their work in three respects: (1) our work emphasizes the extraction of code semantics before measuring the distances of different exploit code (a major contribution of our work), whereas they use canonical string representation in the Exedit/Edit distance for inferring the phylogeny; (2) our work profiles the exploits into a certain category based on the trained supervised learning model while they use an unsupervised agglomerative clustering model, and thus, the statistical models are totally different; and (3) we extract the code semantics from data anomaly analysis, whereas they use control flow graph to generate structure distance, still quite different.

Different statistical model have been explored for intrusion detection systems, for example, *N*-gram model [23] used in traffic anomaly detection, Markov chain model [8] used for web traffic anomaly detection, and support vector machine [6] used for detection of drive-by-downloads attacks. A game-theoretical analysis on how a detection algorithm and an adversary could adapt to each other in an adversarial environment is introduced by Pedro *et al*. [24].

For exploit code attribution analysis, pure statistical approach may not produce very good results because of the lack of semantic information. Recent work SAS [25,26] has looked at the combinations of semantic and statistical analysis to generate signatures for polymorphic worm detection. In contrast, our work is motivated for exploit code attribution analysis instead for polymorphic worm detection, and the statistical model is also different, leading to different strategies used for classification and detection.

3. PROBLEM STATEMENT AND ANALYSIS

This section first gives the formal definition of exploit code attribution problem and then analyzes the challenge of this problem.

3.1. Problem formalization

Let I be a set of different classes of exploit code and d be the total number of instances (variants) generated from a certain class. We use s_{ij} to denote the jth exploit code instance generated from class w_i, that is, $s_{ij} \in w_i$, $w_i \in I$. For example, in reality, a set of different types of exploit code can be generated from different polymorphic shellcode

engines $I = \{\text{CLET, CountDown, Pex, Tapion, } \ldots\}$. Different instances of the same type of exploit code can be generated using complicated obfuscation techniques such as polymorphism and metamorphism [27].

Definition. *Exploit Code Attribution Problem*

(1) For lots of exploit code instances s_{ij}, how is profiling for each category w_i be generated?
(2) For an unknown exploit code s, what is the attribution of s? That is, find i such that $s \in w_i$.

3.2. Challenges

According to the definition of exploit code attribution problem, Problem (1) is a training problem in shellcode classification and Problem (2) is a recognition problem after a profiling for each category of exploit code is built. Problem (1) is the key step, whereas Problem (2) can be easily solved after the learning model is built in Problem (1). These two problems match well with the standard machine learning problem. Naturally, we refer to machine learning techniques for a solution.

With the above analysis, it seems that any statistical approach can work in the context of exploit code attribution analysis. However, the statistics-based approach may not produce promising results. Song *et al.* [12] concluded that it is impossible to model the polymorphic shellcode (see Figure 1 for an example). Polymorphic shellcode accounts the largest part of the exploit code, and therefore, modeling all of the exploit code (e.g., for attribution analysis) is much more difficult.

Next, we will briefly explain why modeling polymorphic shellcode instances is difficult. The contents of the polymorphic shellcode instances usually consist of several parts: no operation (NOP) part (sled), decoder part, encrypted payload part, return address part, and padding part (Figure 1). Modeling the NOP part may amount to modeling random instructions because many instructions are semantically equivalent to "NOP." For example, for the shellcode generated by CLET [28], there are 55 kinds of sled used in the "NOP" part. In the return address part, there are also many variations of the target address by adding padding bytes before it. In the padding part, the binary code can be filled in, without influencing the execution results. Because of obfuscation, the padding bytes may have similar distribution to the normal traffic distribution. In the decoding part, different encryption keys can generate different encrypted exploit code.

Clearly, because of great varieties in each part of polymorphic exploit code, the variations for a whole exploit code packet can be even larger. These great variations may result in the following:

(R_1) No fixed patterns exposed in a whole packet.
(R_2) The attribution analysis process misguided by padding bytes and noisy bytes.

Figure 2 shows two examples of shellcode varieties. Figure 2(a) shows a spectral image, where each pixel

Figure 1. A demonstration of polymorphic shellcode instance.

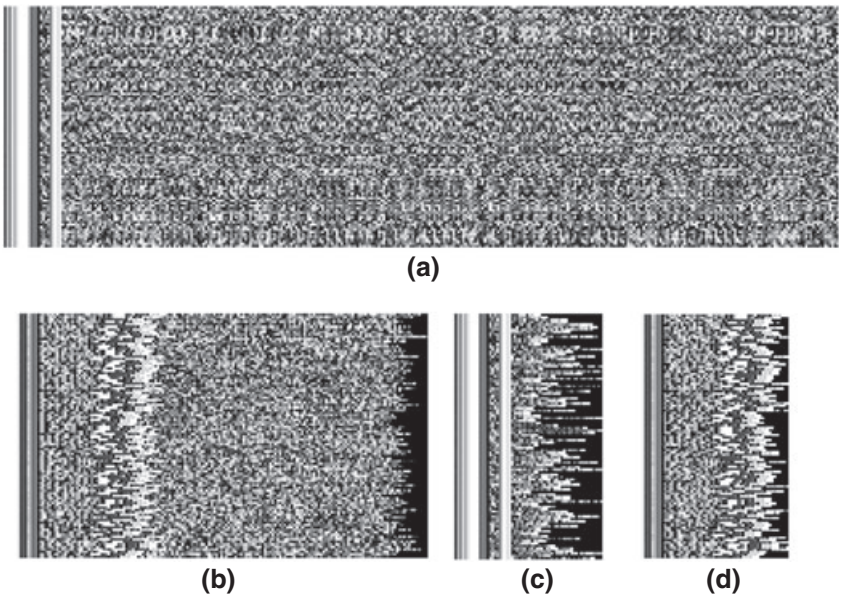

Figure 2. Varieties of shellcode instances. (a) Pex (each of 100 instances is 344 bytes); (b) CLET (each of 100 instances is 168 bytes); (c) Pex refined shellcode after semantic analysis (corresponding to (a), each shellcode is 60 bytes); (d) CLET refined shellcode after semantic analysis (corresponding to (b), each shellcode is 60 bytes). Each pixel of the image represents a byte obtained from a shellcode instance.

represents a byte from a shellcode sequence generated from polymorphic engine Pex [29]. Each row is corresponding to a shellcode sequence with 344 bytes in length and totally 100 instances form the image. Similarly, Figure 2(b) shows the spectral image formed by 100 sequences generated from polymorphic engine CLET [28], where each row is a shellcode sequence of 168 bytes in length. Clearly, these images demonstrate great varieties of different bytes in exploit code, which imply that the shellcode attribution analysis is a challenging problem.

4. OUR APPROACH

In this section, we present our approach SA^3, an exploit code attribution analysis—profiling malware into class and identifying the malware class of a given malware. We emphasize the combinations of semantic analysis with statistical analysis. Semantic analysis is based on identifying control and data flow behaviors of malware. Statistical behavior is based on measuring the distance (similarity metric) between the code on known malware and the code of a potential malware. The operator used to combine the semantic and statistical analysis is union/summation. Next, we give more detailed introduction of our method.

4.1. Framework

In Figure 3, we describe the framework of SA^3. The core modules of SA^3 are *semantic analysis module* and *statistical analysis module*. More specially, we use *data anomaly analysis* in the semantic analysis module and a *two-way mixture Markov (TWMM) model* in the statistical analysis module.

The whole workflow of SA^3 can be divided into the training stage (*with the real line*) and the recognition stage (*with the dashed line*). First of all, the same type of exploit code instances are fed into the semantic analysis module, and data anomaly analysis is conducted on them. We get the refined exploit code instances, which are actually the instruction sequences after useless instructions are pruned. Next, a TWMM model is built on the refined input instruction sequences. We construct a mixture Markov model corresponding to each category of exploit code. When a new exploit code instance comes, it will be first analyzed through the data anomaly analysis module. Thus, the refined code sequences are distilled as the input to the TWMM model. The decision result is obtained by attributing the exploit code sequence to the one with the most fitting value.

4.1.1. Semantic module.

For each category of the input instruction sequences, we prune semantic-unrelated code existed in the code sequences. Data anomaly analysis is used to capture the "semantics" of the exploit code through preserving the useful instructions while pruning the useless ones, which probably contain padding and noisy bytes in the packets. This module is used for solution of R_2 presented in Section 2.2.

4.1.2. Statistical module.

For the pruned instruction sequences, a TWMM model is built for the solution of R_1 (Section 2.2). On one hand, it is not very clear what kind of relationship exists in the instruction sequences. On the other hand, Markov model is very suitable to model the uncertainty existed in different context. Thus, we refer to a TWMM model, to model relationships between the instruction sequences. The property of "two-way mixture model" makes it more robust and powerful in representing the varieties of different categories of code.

4.2. Relations to semantics-based approach and statistics-based approach

So far we have mentioned (1) a data anomaly-based semantic module and (2) a Markov model-based statistical module used for code attribution analysis. Our approach is the union of the aforementioned two modules. Actually, each module can be viewed as an independent method and used for exploit code (attribution) analysis.

For semantic module, there are many alternatives for exploit code analysis, for example, control-flow graph-based approach [16], data-anomaly-based approach [18], and system-call-based approach [19]. Here, we adopt the data anomaly analysis because it is an efficient and effective way to capture the semantics of exploit code through static analysis. Note that the aforementioned semantics-based method has extracted the semantics from code; it, however, cannot be used directly for code attribution analysis because of the lack of statistical measurement. It must be combined with statistical metric (e.g., edit distance [11]) to compute the similarities of different exploit code. This is one reason leading to a natural combination of semantic analysis with statistical analysis.

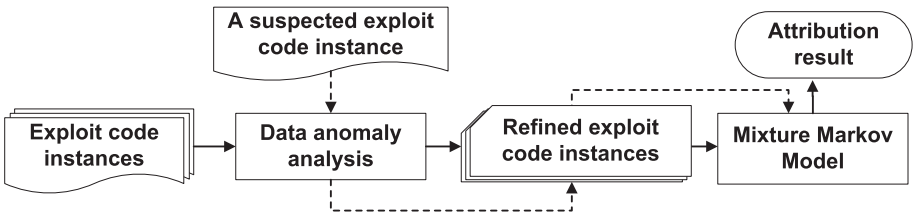

Figure 3. SA^3 flow graph (real line for training stage, dashed line for recognition stage).

For statistical module, there are also many alternatives for code similarities analysis, for example, naive-distance-based approach [11], *N*-gram model [23], and support vector machine [6]. Here, we present a two-way Markov model to capture the subtle variations among exploits within each family and great diversities across different families. Another reason why we use this model is that it can be easily adapted to deal with variable-length sequences with good performance. Note that this statistics-based method can be directly applied for exploit code attribution analysis either on the original exploit code or on the analyzed/refined exploit code sequences after semantic extraction. In the evaluation part, we make a comparison of the aforementioned two cases and show the effectiveness of semantic analysis before applying the statistical model.

Next, we will illustrate the semantic module (Section 5) and statistical module (Section 6) used in our approach in great detail.

5. DATA ANOMALY ANALYSIS TO CAPTURE CODE SEMANTICS

In this section, we introduce the semantic module used in SA^3. To be exact, we use data anomaly analysis to capture code semantics.

The key observation is that certain control and data flow information remain invariable to implement certain functions in the exploit code. We call those control and data flow information as "semantics." The data anomaly analysis is used to capture those semantics because it can preserve the useful instructions by pruning the useless ones, which contain padding and noisy bytes.

First of all, we use disassemble analysis to analyze the input binary instruction sequences. In this paper, what we focus on is HTTP message flows. In an HTTP request message, malicious payload only exists in Request-URI and Request-Body of the whole flow [15]. We extract these two parts from the HTTP flows for further semantic analysis. Then, we make disassemble analysis on these input sequences. If the disassemble module finds consecutive instructions in the input sequences, it generates the assembly instruction sequences 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. Because we do not know the entry point of the code when the code is present in the byte sequences, we explore an improved recursive traversal disassemble algorithm introduced by Wang *et al.* [15] to disassemble the input instruction sequences. For an *N*-byte sequence, the time complexity of this disassemble algorithm is $O(N)$.

After disassemble analysis, it may generate zero, one, or multiple instruction sequences, which do not necessarily correspond to real code. Next, we distill useful instructions by pruning useless instructions using the technique introduced in SigFree [15]. Useless instructions are those illegal and redundant byte sequences. By using the code abstraction, a static analysis technique, we can emulate the executions of instruction sequences.

There are six possibly states in the state transition graph generated from the code sequences. State *U* represents an undefined variable state; state *D* represents a defined but not referenced variable state and state *R* represents a defined and referenced variable state. The other three abnormal states are defined as follows: state *DD* represents an abnormal state *define–define*, state *UR* represents an abnormal state *undefine–reference*, and state *DU* represents an abnormal state *define–undefine*. Basically, the pruned useless byte sequences correspond to three kinds of dataflow anomalies: *UR*, *DD*, and *DU*. 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 that 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. Useful instructions are those instructions left after rigorous data anomaly analysis (as we mentioned earlier: undefine–reference anomaly, define–define anomaly, and define–undefine anomaly).

Because crafted noisy bytes in the packets typically do not contain useful instructions, such irrelevant bytes in the packets are filtered out after the useful instruction extraction phase. The remaining instructions are likely to be related to the semantics of the code kept in the exploit code sequences.

Next, we further explain our motivation for useful instruction extraction. From our observation, lots of "useful instructions" are left invariant across different shellcode instances even after complicated obfuscations (e.g., junk insertion, instruction replacement). For padding and noisy bytes, they still can be assembled into code sequences. However, usually, it lacks clear meanings and correlated relations for those coincidental instruction sequences. Thus, they will be pruned after rigorous data flow anomaly analysis. Moreover, we note that the remaining useful code sequences are more likely to be similar to those from the same category instead of those from the other categories.

5.1. Motivating example

An example of polymorphic code analysis is shown in Figure 4. Here, the leftmost part is the original packet content in binary; the middle part and the right part are the disassemble code and its corresponding binary code of the useful instructions after removing useless ones, respectively. For example, the disassembled code *inc ecx* appeared in address 42 is pruned because *ecx* is defined again in address 4c to produce a *define–define* anomaly. In address 44, the contents in the memory cell with address *ecx-4A* is referenced without being defined beforehand.

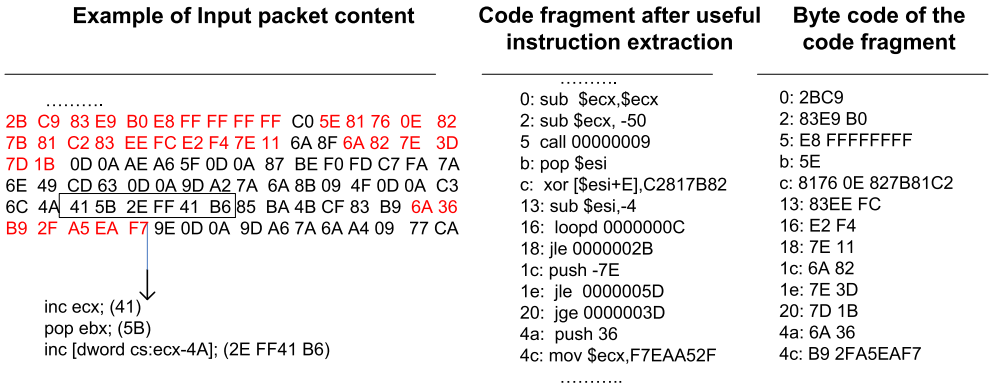

Figure 4. A motivating example to show the procedure of semantic analysis on the input code sequence.

Thus, we prune this instruction because it produces an *undefine–reference* anomaly.

Figure 2(c) and (d) shows two other examples. Figure 2(c) gives the spectral image formed by the remaining instructions of 100 instances corresponding to Figure 2(a). Similarly, Figure 2(d) gives the spectral image formed by the remaining instructions of 100 instances corresponding to Figure 2(b). In both images, each pixel represents a byte from the remaining instructions. Clearly, the lengths of the preserved code sequences are decreased. More importantly, the fixed patterns in the original code sequences are preserved, whereas the bytes located in different positions with large varieties are cut off.

From the aforementioned polymorphic shellcode example (Figure 4) and other instances, we find that the remaining code sequences usually consist of the following features: (F_1) GetPC: the code to get the current program counter, usually contains opcode "*call*" or "*fstenv*"; (F_2) Iteration: a polymorphic exploit code usually performs iterations over encrypted shellcode using the operations such as *loop*, *rep*, and the variants of such instructions (e.g., *loopz*, *loope*, *loopnz*); (F_3) Jump: a polymorphic exploit code usually contains conditional/unconditional branch statements (e.g., *jmp*, *jnz*, *je*); (F_4) Decryption: for the encrypted shellcode, certain machine instructions (e.g., *or*, *xor*) are more often to be found in decryption routines because shellcode needs to be decrypted before execution. These features are preserved after semantic analysis, which can be further used for statistical modeling. We believe that these features can help capture the category of shellcode, and they may exist in most of the self-contained exploit code.

It may be attempting to use (F_2, F_4) as the only feature for category analysis. Fortunately, we also have other useful instructions preserved except for the features (F_1, F_2, F_3, F_4). This motivates us to use the statistical model for capturing the differences across various exploit codes as much as possible.

For non-self-contained code, not all features (e.g., F_2, F_1) exist in the shellcode (e.g., code generated from Avoid UTF8/tolower [4]) because of the absence of GetPC and self-reference operations. In these cases, the remaining instruction sequences still can be taken as good indicators for shellcode category analysis because noisy bytes are filtered. The pruned bytes are more likely to mislead the state-of-the-art statistics-based learning approaches (e.g., *N*-gram based learning [23], Markov Chain [8], support vector machine [6]) for category analysis or detection. Note that the length of code sequence can be viewed as the number of dimension for the training code sequences. It is essential to prune useless instruction in order to reduce the dimension of training data. This makes the statistical module much easier and more accurate by alleviating the difficulty of "curse of dimensionality" [30].

6. MARKOV-MODEL TO CAPTURE THE CODE SIMILARITIES

In this section, we briefly introduce the statistical model used in SA^3. To be exact, we use a two-way of mixture Markov model to capture the code similarities. We first introduce the reason why we use this model. Then, we describe the construction of this model. Finally, we derive an algorithm to solve our model.

6.1. Why Markov model?

Let Y be the set of single bytes and Y^i denote the set of i-byte sequences. $X = Y \cup Y^2 \cup Y^3 \cup Y^4$ is the token set in our system because a token in a useful instruction contains at most 4 bytes (e.g., "*AAFFFFFF*"), which corresponds to the word size of 32-bit systems. A Markov chain [31] is a sequence of random variables $X_1, X_2, X_3, \ldots$, satisfying the Markov property: given the present state, the future and past states are independent. More formally, probability $\Pr(X_{n+1} = x | X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n) = \Pr(X_{n+1} = x | X_n = x_n)$, where x_i is the value for each state X_i, and $\Pr(X_{n+1} = x | X_n = x_n)$, is the conditional probability for transition from state X_n to X_{n+1}. The possible values of X_i form a countable set S, called the state space of the chain.

We observe that there are close relations among the code tokens in the refined instruction sequences. Markov Chain [31] is a good candidate to model uncertain dependencies in different contexts. In the context of code sequence analysis, each token in a sequence can be viewed as a state in a Markov Chain. We assume that a token in a sequence is dependent on the token in front and also the token next because of the great dependencies that existed in the code sequences of the nearest neighbors.

To be exact, the dependency of token x_j on x_i is the co-occurrences of token x_j and x_i . If x_i appeared in front of x_j in the same sequence, we can say that x_j is forward dependent on x_i. Otherwise, if x_i appeared after x_j in the same sequence, we can say that x_j is backward dependent on x_i. One-order Markov chain requires the nth token in a chain is only dependent on the $(n-1)$th token. However, in real code segment, the nth token can be dependent on the $(n-1)$th, $(n-2)$th, ..., $(n-p)$th tokens in a sequence, and also related to $(n+1)$th, $(n+3)$th, ..., $(n+q)$th tokens. We do not know what is the value of p and q beforehand.

In real exploit code, one instruction may be dependent on both tokens that appeared in the front or the back. In order to capture both forward and backward relationships, we define two kinds of relationships (forward dependency and backward dependency) in our model. We call them "bidirectional dependence." Our Markov-derived model is named as TWMM model.

First, we define the forward dependence; that is, the nth token is depended on consecutive p tokens in front. Next, we define the backward dependence; that is, the nth token is dependent on the next consecutive q tokens. Then, parameters $\pi_i (i=1,2)$ are used to make a balance between them, where $\pi_1 + \pi_2 = 1$. Parameter π plays a role to balance the two dependencies. Moreover, by utilizing the statistical optimization theory, the optimized π can be found automatically.

Figure 5 shows an example, where token 96 is forward dependent on p ($p=2$) tokens (83*C*6, 01) in front and also backward dependent on next q ($q=3$) tokens (40, 96, 46).

6.2. Model construction

First, we construct a TWMM model for each category of code sequences. Second, after a new code sequence is fed into the model, we attribute it to the class with the highest fitting value. However, if the highest fitting value is still less than a certain threshold, we will attribute it to the normal sequence. Here, the fitting value is the accumulation of probabilities, which reflects the matching score from a code sequence to the model.

Next, we show how to compute the probability for a code sequence. The probability of a code sequence can be decomposed into the product of the probability of each token in a sequence. For different tokens appeared, there is a transition matrix to label the probability from one token to another. Hence, p-forward tokens' transition probability to a specific token is the probability from front p tokens' transition probability to this token. Similarly, q-backward tokens' transition probability to a specific token is the probability from next q tokens' transition to this token.

In forward model, the ith token's probability is computed through product of the p-forward tokens' transition probability to this token. Similarly, in the backward model, the ith token's probability is computed through the product of the q-backward tokens' transition probability to this token. The probability is a product of $p(L_n - p)$ values in the forward model and a product of $q(L_n - q)$ values in the backward model, where L_n is the length for the nth code sequence. Therefore, the $p(L_n - p)$ root is needed for computing the sequence probability in forward model, and $q(L_n - q)$ root is needed in backward model.

More formally, let $x_{n,i}$ denote the ith token in the nth sequence, $A_1(x_{n,i}|x_{n,j};\theta_1)$ denote the transition probability from token $x_{n,j}$ to token $x_{n,i}$ in forward model θ_1, and $A_2(x_{n,i}|x_{n,j};\theta_2)$ denote the transition probability from token $x_{n,j}$ to token $x_{n,i}$ in backward model θ_2. Because the same token can be transferred to different tokens, the sum of such transition probability should be normalized to 1, that is,

$$\sum_{x_{n,i}} A_k\left(x_{n,i}\middle|x_{n,j};\theta_k\right) = 1(k=1,2) \tag{1}$$

Let $g(x_n|\theta_1)$ and $g(x_n|\theta_2)$ denote the probability for the nth sequence's matching scores in the forward model and backward model, respectively. Thus, we have

$$g(x_n|\theta_1) = \left(\prod_{i=p+1}^{L_n}\prod_{j=i-p}^{L_n-p} A_1\left(x_{n,i}\middle|x_{n,j};\theta_1\right)\right)^{\frac{1}{(L_n-p)p}} \tag{2}$$

$$g(x_n|\theta_2) = \left(\prod_{i=1}^{L_n-q}\prod_{j=i+1}^{i+q} A_2\left(x_{n,i}\middle|x_{n,j};\theta_2\right)\right)^{\frac{1}{(L_n-q)q}} \tag{3}$$

Next, by combing $g(x_n|\theta_1)$ and $g(x_n|\theta_2)$ in a balanced way, we have G_n to denote the matching score for nth sequence, that is,

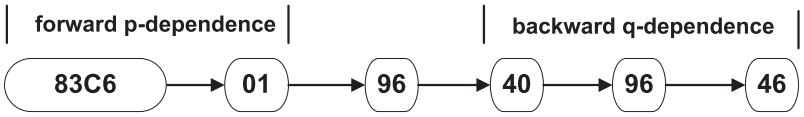

Figure 5. Explanation of dependence in Markov model.

$$G_n = \sum_{k=1}^{2} \pi_k g(x_n|\theta_k), \tag{4}$$

where $\pi_1 + \pi_2 = 1$. To obtain the solution for this model means to estimate the parameters in Eqation (4). Suppose that we have N sequences for each category; thus, the object function G to be optimized is the product of the likelihood for each sequence G_n, that is,

$$G = \prod_{n=1}^{N} \sum_{k=1}^{2} \pi_k g(x_n|\theta_k) \tag{5}$$

6.3. Model solution

Now, we show how to solve Equation (5). The object function G is to be maximized to fit the model according to the principle of maximum likelihood estimation [32]. From the point view of optimization techniques, the object function is not convex in terms of the mixture of two different Markov chains, thus directly setting the first order derivatives on the likelihood does not work. This model is also different from the standard mixture model, which requires the same format of sub-models in a mixture model. Thus, we use the Expectation Maximum (EM) algorithm [33] to iteratively maximize the likelihood function with a gradient descent algorithm.

The EM algorithm usually takes two steps, expectation step and maximization step. At each step, the model's likelihood function is updated in the direction of gradient ascent, and this process is iterated until the likelihood converges. The monotonic property makes this approach effective for the solution of many non-convex optimization problems. Next, we show how to train our model with the EM algorithm.

First, we construct the affiliated function [33]

$$W(\Theta, Q) = \sum_{n=1}^{N} \sum_{k=1}^{2} Q_{nk} log \frac{\pi_k g(x_n|\theta_k)}{Q_{nk}} \tag{6}$$

where Q_{nk} works as the hidden variable to denote the weight of data point n in terms of model k, and $\sum_{k=1}^{2} Q_{nk} = 1$. Because the log function is a concave function, according to the Jensen's inequality,[†] we have $log(\sum x) \geq \sum logx$. Thus, $logG \geq W(\Theta, Q)$. The maximization of the object function G in Equation (5) is equivalent to the maximization of Equation (6) because Equation (6) is the new lower bound of the likelihood function to be maximized.

Let Θ denote the parameters in the transition probability matrix $A_k(x_{n,i}|x_{n,j}; \theta_k)(1 \leq k \leq 2)$ and Q the hidden variable set Q_{nk}. Let Θ^t and Q^t denote each group of parameters used in the tth iteration in the parameter estimation process. During the maximization step, the object function of Equation (6) is required to be monotonically increased. With this, we obtain

$$W(\Theta^t, Q^t) \leq W\left(\Theta^{t+1}, Q^t\right) \leq W\left(\Theta^{t+1}, Q^{t+1}\right) \tag{7}$$

which can be solved by using Lagrange multipliers [34] to find the stationary points with $arg\ max_{\Theta}\ W(\Theta, Q^t)$ and $arg\ max_{Q}\ W\left(\Theta^{t+1}, Q\right)$ satisfied in each step.

Let $C(x_{n,i}|x_{n,j})$ denote the frequency of token transition from $x_{n,j}$ to $x_{n,i}$ in the nth sequence. Naturally, we use $C(\cdot|x_{n,j})$ to denote the frequency of the token transition from $x_{n,j}$ to any tokens in the nth sequence of the model. To solve Equation (7), we obtain solutions in Equations (8) and (9). The complete training algorithm is shown in Algorithm 1.

$$Q_{nk} = \frac{\pi_k g(x_n|\theta_k)}{\sum_{k=1}^{2} \pi_k g(x_n|\theta_k)}, \pi_k = \frac{\sum_{n=1}^{N} Q_{nk}}{N} \tag{8}$$

$$A_k(x_i|x_j; \theta_k) = \frac{\sum_{n=1}^{N} \frac{Q_{nk}}{L_n(L_n - \lambda_k)} C(x_{n,i}|x_{n,j})}{\sum_{n=1}^{N} \frac{Q_{nk}}{L_n(L_n - \lambda_k)} C(\cdot|x_{n,j})}, \quad \lambda_1 = p, \lambda_2 = q \tag{9}$$

Algorithm 1. EM training Algorithm

Input: Instruction sequences $I_0, I_1, I_2, \ldots, I_n$ of each category, ε is the parameter used for convergence decision.
Output: Parameters (Θ, Q) for each category.
Procedure:
1: Initialize π_k, $A_k(x_{n,i}|x_{n,j}; \theta_k)$, $Q_{nk}(1 \leq k \leq 2)$
2: Compute the probability for each sequence to obtain $W(\Theta^t, Q^t)$ with Equation (6)
3: Update Q_{nk}, π_k with Equation (8); update $A_k(x_{n,i}|x_{n,j})$ with Equation (9)
4: **if** $W(\Theta^{t+1}, Q^{t+1}) - W(\Theta^t, Q^t) < \varepsilon$ **then**
5: The algorithm converges, stop training
6: **else**
7: goto step 2
8: **end if**

The aforementioned Markov-derived model has a large state space (2^{32}), and thus, it seems impractical for code sequence recognition. Fortunately, lots of tokens never or seldom appear in the state space, and this gives us the opportunity to greatly reduce the state space. First, we ignore never appeared tokens and prune seldom appeared tokens by setting a threshold. It leads to sparse items in the whole state space and very sparse transition matrices. Second, we use the data structure of hash table for storage of state transition probabilities in order to reduce the memory and computation cost.

[†]For any concave function $f(x)$, if the balanced parameter t satisfies $0 < t < 1$, we have $f(tx_1 + (1-t)x_2) \geq tf(x_1) + (1-t)f(x_2)$.

7. EVALUATION

We test our system offline on massive polymorphic exploit code packets and on HTTP normal reply/request traces. First of all, we evaluate our approach on different kinds of exploit code in terms of false positives and false negatives, and then, we compare our approach with the approach free of any semantic analysis before attribution analysis. Next, we evaluate our approach in terms of computation time cost. Finally, we discuss the advantages and limitations of our approach.

The massive polymorphic exploit code packets are generated by the Metasploit [29] framework (e.g., PexFnstenvSub, Pex, ShikataGaNai) and also from polymorphic engines (e.g., CLET [28], ADMutate [35], JempiScodes [36]). CLET, ADMutate, JempiScodes, and ShikataGaNai are advanced polymorphic engines that obfuscate the decryption routines by metamorphism such as instruction replacement and garbage insertion. CLET uses spectrum analysis to counterattack the byte distribution analysis. Opcodes of the "*xor*" and "*fnstenv*" instruction are frequently found in the decryption routine of PexFnstenvSub and also in getting the values of the program counter register (GetPC). Pex uses *xor* decoders and relative call to get PC.

The exploit codes are sufficiently different from each other in terms of their semantics. The normal HTTP traffic contains 300 000 messages collected for 3 weeks at seven workstations owned by seven different individuals in our lab. To collect the traffic, a client-side proxy monitoring incoming and outgoing HTTP traffic is deployed underneath the web server. Those 300 000 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. We run our experiments on a 2.4-GHz Intel Quad-Core machine with 2-GB RAM, running Windows XP SP2.

7.1. Attribution analysis results

First, we evaluate our approach in different parameter settings in terms of different combinations of p and q. Second, we compare our approach with the approach free of making any semantic analysis beforehand. Here, we do not discuss much about data anomaly analysis because they have been well studied in previous researches [15,18].

7.1.1. Exploit code attribution.

For each category of exploit code, we generate a corresponding TWMM Model, and then, the new packets are fed into the model to evaluate the false positives and false negatives. We use fivefold cross validation to train the model and get the false negatives by matching the packet with the corresponding model. During the packet attribution phase, a threshold is set to decide the attribution for this packet. The threshold will both influence the false positives and false negatives in the receiver operating characteristic curve. As shown in Figure 6, for different combinations of p and q, we can get different results by setting different thresholds.

Another factor to influence the attribution analysis result is the setting of the parameters p and q. There are many choices of (p, q) combinations because p and q can be freely selected if we do not know any prior knowledge of the structures of code sequences. It is not realistic to brute-force search all possible (p, q) combinations.

From our observations, for each token, the tokens close in distance have much more influential power on it. That means p and q can be set to small numbers. We do not know exactly which is the best to achieve the optimal results. In our evaluation, we tentatively choose $p, q \in \{2, 4\}$.

From the results on different datasets in Figure 6, we can infer that token relevances are different on different datasets. Besides the parameters that influence the attribution results, the attribution analysis accuracy varies depending on the "nature" of the exploit code. On all six datasets, the detection accuracy can reach to above 90% in different parameter settings. This is a good indictor to show the effectiveness of our approach. The false positive rate is up to 4.5% at most. We may further bootstrap the misclassified packets to increase the analysis accuracy in our future work.

7.1.2. Comparison with approach *without* semantic analysis.

We compare our approach with the approach free of any semantic analysis beforehand. The same TWMM model is constructed for the original packets but *without* any semantic analysis before the attribution analysis. In the approach without any semantic analysis, the tokens used are all 1-byte tokens because we do not have any prior knowledge about the minimum semantic cell used in the whole code sequence.

Note that the changes of combinations of (p, q) do not make much difference for detection accuracy and false negative rate in our attribution analysis; thus, we set $p = 2$, $q = 2$ when making a comparison with the approach without semantic analysis. The results are also shown in Figure 6. Our semantic aware approach outperforms the approach without semantic analysis on all six data sets, and the detection accuracy can be boosted more than 10% on all six data sets with nearly the same false positives. The promising results show that our semantic aware attribution analysis is effective and much better than the approach with no semantic analysis.

7.1.3. Across category testing.

In the previous subsections, each exploit code is tested against its own category to obtain accuracy. In order to have a more complete assessment of our approach, we also carry out the experiments across categories and report the false negatives here. Similar to the previous experiments, we try different parameter settings, and we tentatively set $p, q \in \{2, 4\}$.

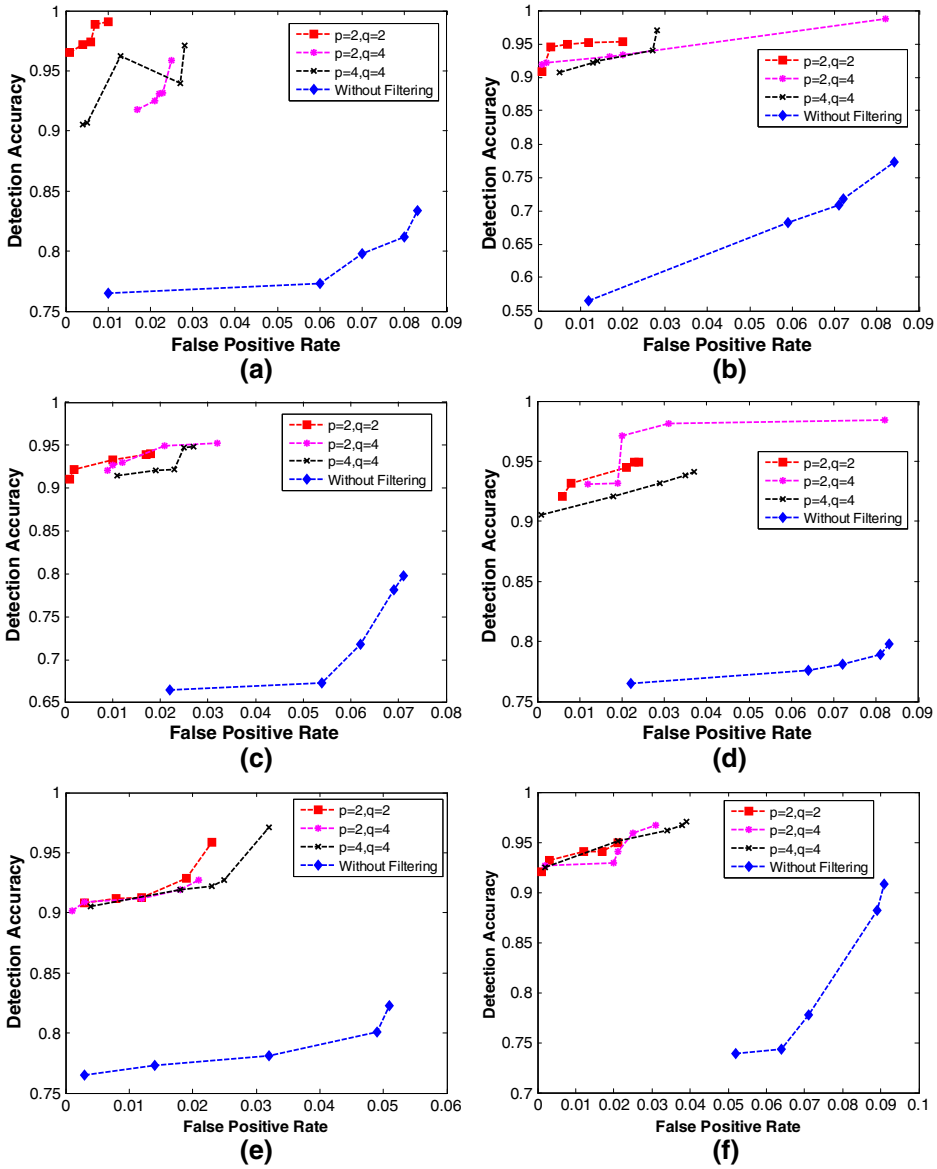

Figure 6. Comparisons of semantic aware approach with the approach without filtering noises on six data sets: (a) CLET; (b) ADMutate; (c) PexFnstenvSub; (d) JemipiScodes; (e) Pex; (f) ShikataGaNai.

Confusion matrix is a specific table layout that allows visualization of the performance of a classification algorithm. Each column of the matrix represents the instances in a predicted class, while each row represents the instances in an actual class. The (j, i) element of a confusion matrix shows the percentage of instances from category i is labeled as category j. Clearly, the diagonals of the confusion matrix (i.e., (i, i) element) indicate the percentages of the correctly labeled instances, whereas the off-diagonals indicate the percentages of incorrectly labeled instances.

Figure 7(a–d) shows the confusion matrices on different combinations of parameters of p and q. The color bar here shows the value scales (0–100%) of confusion matrices. For demonstration purpose, each color is corresponding to a value for an element of a confusion matrix. For example, the red color means that the value is large (approaching 100%), and the blue color means that the value is small (approaching 0%). We use Figure 7(a) to further illustrate the meanings of the confusion matrix. For the first column, the values of confusion matrices are [91.2 %, 1.8 %, 1.76 %, 1.74 %, 1.77 %, 1.73 %]. This is corresponding to the six categories used in our evaluation. This column is "CLET." According to our definition, in the predicted results of the actual exploits from category "CLET," 91.2% are labeled as "CLET," whereas 1.8% are labeled as ADMutate, 1.8% are labeled as PexFnStenvSub

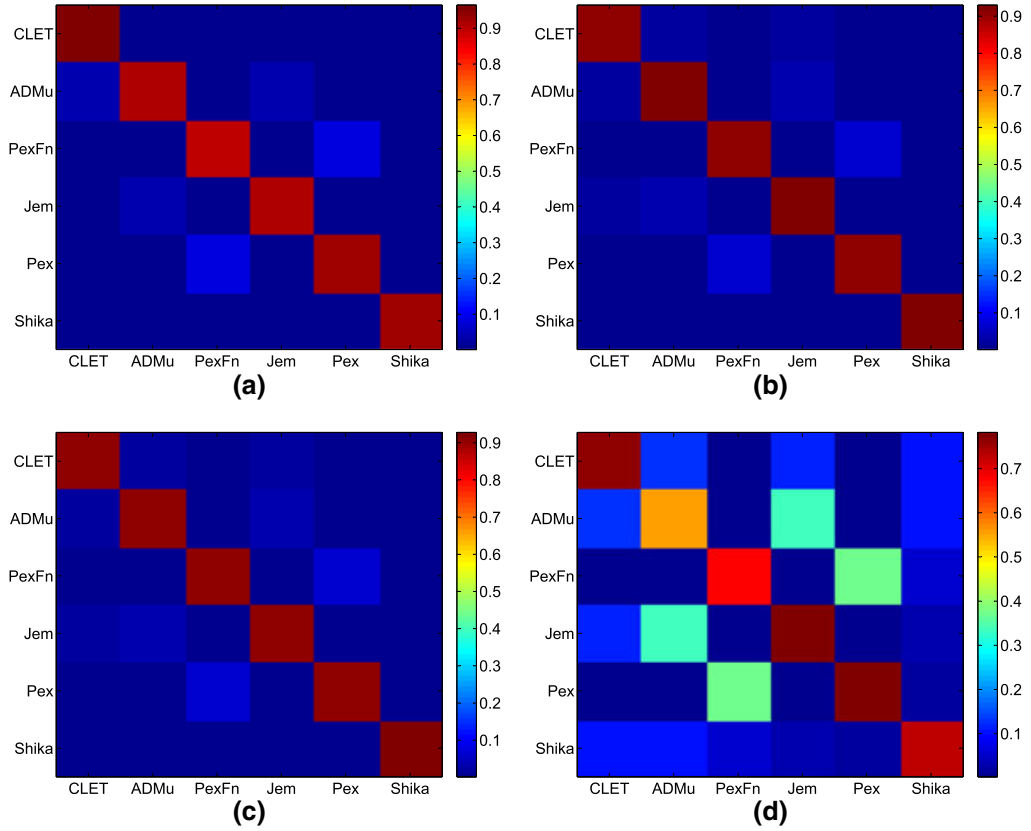

Figure 7. Comparisons of confusion matrices on different parameter settings. Six rows/columns are corresponding to the results obtained from CLET, ADMutate (ADMu), PexFnstenvSub (PexFn), JemipiScodes(Jem), Pex, ShikataGaNai(ShiKa). (a) $p = 2$, $q = 2$; (b) $p = 2$, $q = 4$; (c) $p = 4$, $q = 4$; (d) without filtering.

(PexFn), and so on. Moreover, 91.2% is the true positive for category "CLET," and [1.8 %, 1.76 %, 1.74 %, 1.77 %, 1.73 %] are the false negatives for the exploits from category "CLET" mislabeled as the other five categories.

Clearly, the diagonals of all confusion matrices are very strong. For the off-diagonal elements, the largest false negatives are produced between category PexFnstenvSub and Pex, which is 8.2% when $p=2$, $q=2$. There are also false negatives produced among categories CLET, ADMutate, and JemipiScodes, which is 4.3% at most when $p=2$, $q=2$. Similar to the previous results, different combinations of (p, q) also influence the false negatives. Note that there is a tradeoff between the false positives and false negatives. In our approach, we tune parameters to reduce the false positives as much as possible, and therefore, the false negatives are a little higher here.

7.1.4. Byte entropy-based approach for category analysis.

Statistical test (e.g., entropy test) is proposed for detecting packed or encrypted malware. Here, we also use the byte entropy test [37] to measure the randomness of the distribution of the bytes in different exploits. We use the entropy

$$Entropy(X) = -\sum_{X} P_r(x) log P_r(x), \qquad (10)$$

where X is the exploit sample and $P_r(x)$ is the probability $P_r(X=x)$.

In our experiments, we measure the byte entropy of different shellcode instances. We examine the entropy values to distinguish among the exploits from different categories. The mean and standard deviations of the byte entropy scores are listed in Table I. The mean scores for exploits from different categories are very similar. For example, the mean score (4.798) for exploits from category Pex can fall well within one standard deviation of those from the other category (PexFnstenvMov), which is 4.803 ± 0.015 (i.e., 4.788–4.818).

With these results, the byte entropy test is unable to distinguish the exploits from different categories. This indicates that simple statistical test does not work for the category analysis problem. In our approach, we use semantic aware statistical model to make a differentiation of them.

Table I. Byte entropy results.

Polymorphic engine	Mean	Variance
CLET	4.486	0.052
ADMutate	4.422	0.063
PexFnstenvMov	4.803	0.015
JempiScodes	4.723	0.014
Pex	4.798	0.126
ShiKataGaNai	4.982	0.272

7.2. Performance evaluation

Table II shows the time cost during the training phase and decision phase. The training time is the average time cost for each packet used in training, which includes the time of semantic analysis and also the time used for the training process of statistical model. The decision time is the average time cost for packet recognition, also including the time for semantic analysis.

The training time cost is high as a result of the use of the EM algorithm in the mixture Markov model. The EM algorithm usually needs hundreds of iterations before convergence especially when data do not fit a model very well (e.g., exploit code instances generated from ShiKataGaNai). It also takes time in the semantic analysis module, but the time cost for semantic analysis is negligible compared with the EM algorithm in the training process. Fortunately, in order to reduce the time cost, we can conduct the training process offline before the recognition phase. The recognition phase is very efficient.

7.3. Discussion

Here, we further discuss the strengths and limitations of our approach.

7.3.1. Strengths.

First of all, our approach can filter noises through semantic analysis in the code sequences, and thus, it has very good noise tolerance.

Second, our approach is very robust to many different kinds of attacks (e.g., coincidental-pattern attacks [38], the token-fit attacks [39], and allergy attacks [22]) due to the semantic analysis module applied. Moreover, our approach explores the semantic features to the classification process, which leverages the "semantics" to increase attribution analysis accuracy. This opens a door to combine the semantic analysis with statistical analysis for practical tasks.

Finally, during the training phase, it may be difficult to get many (e.g., 300, 400) training data for each category in a real deployment environment. The attribution results may decay because of the lack of training instances. Fortunately, compared with other models (e.g., support vector machine), Markov model has stronger recognition ability even with scare training data (e.g., 10, 20). That is why we use Markov-derived model in our statistical modeling module.

Table II. Average time cost for each packet (ms).

Polymorphic engine	Training time	Decision time
CLET	5213	7.2
ADMutate	4142	3.2
PexFnstenvMov	3829	4.5
JempiScodes	2487	6.8
Pex	3152	4.1
ShiKataGaNai	7650	4.3

7.3.2. Limitations.

First of all, because our semantic module is based on static analysis, we cannot handle some state-of-the-art code obfuscation techniques (e.g., branch-function obfuscation) in the semantic module, which may mislead the feature generations before statistical analysis. This can be solved by referring to more complicated semantic aware static/dynamic analysis techniques (e.g., symbolic execution, type inferences).

Secondly, for non-self-contained exploit code [4] and complicated obfuscated metamorphic exploit code, sometimes we fail to capture the features of such code before statistical analysis. The code may mislead the classifier to make the wrong decision results. This is also a problem that state-of-the-art statistical learning techniques cannot handle.

Finally, our semantic module based on static analysis is simplistic. It does not take branch into consideration while pruning exploit code. The define–define anomaly, for example, mentioned in Section 3 is very likely to be one variable being assigned differently in two branches. Simply pruning either one of the two definitions may lead to inaccuracy. In other words, a better static analytic module including control flow and data flow analysis can be used to strengthen the robustness of the semantic module.

8. CONCLUSION

In this paper, we present SA^3, an automatic exploit code attribution analysis system. The novelty of SA^3 is that it combines semantic analysis with statistical modeling for exploit code attribution analysis.

We use data anomaly analysis to extract the semantics from the exploit code sequences. We also derive a novel Markov model to modeling the pruned exploit code sequences. These two analysis help to capture the characteristics of the exploit code used for exploit code attribution analysis.

The comprehensive experiments show that our approach outperforms the pure statistics-based approach with much better accuracy. Our evaluation results also refute the conclusion of "impossibility of modeling polymorphic shellcode [12]" in some degree.

ACKNOWLEDGEMENTS

This work was supported by AFOSR FA9550-07-1-0527 (MURI), ARO W911NF-09-1-0525 (MURI), NSF CNS-0905131, NSF CNS-0916469, and ARO W911NF1210055.

REFERENCES

1. CRET. Computer emergency response team. http://www.cret.org/.
2. SecurityFocus. http://www.securityfocus.com/.
3. Baecher P, Koetter M. Getting around non-executable stack (and fix). http://libemu.carnivore.it/.
4. Krügel C, Lippmann R, Clark A. Emulation-based detection of non-self-contained polymorphic shellcode. *10th International Symposium on Recent Advances in Intrusion Detection*, 2007.
5. Bania P. Evading network-level emulation. http://packetstormsecurity.org/papers/bypass/.
6. Rieck K, Krueger T, Dewald A. Cujo: efficient detection and prevention of drive-by-download attacks. *Proc. of 26th Annual Computer Security Applications Conference (ACSAC)*, 2010.
7. Wang K, Cretu G, Stolfo S. Anomalous payload-based worm detection and signature generation. *Proceedings of the Recent Advances in Intrusion Detection*, 2006.
8. Song Y, Keromytis A, Stolfo S. Spectrogram: a mixture of Markov chains model for anomaly detection in web traffic. *Proceedings of the Network and Distributed System Security Symposium*, 2009.
9. AV-test. http://www.av-test.org/.
10. Hu X, Chiueh T, Shin K. Large-scale malware indexing using function-call graphs. *ACM Conference on Computer and Communications Security, pages 611–620*, 2009.
11. Ma J, Dunagan J, Wang HJ, Savage S, Voelker GM. Finding diversity in remote code injection exploits. *Internet Measurement Conference*, 2006; 53–64.
12. Song Y, Locasto M, Stavrou A, Keromytis A, Stolfo S. On the infeasibility of modeling polymorphic shellcode. In *Proceedings of the 14th ACM conference on Computer and communications security (CCS)*. 2007; 541–551.
13. Polychronakis M, Anagnostakis K, Markatos E. Network-level polymorphic shellcode detection using emulation. In *Proceedings of the GI/IEEE SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*. 2006; 54–73.
14. Gu B, Bai X, Yang Z, Champion A, Xuan D. Malicious shellcode detection with virtual memory snapshots. *INFOCOM*, 2010; 974–982.
15. Wang X, Pan C, Liu P, Zhu S. SigFree: a signature-free buffer overflow attack blocker. *15th Usenix Security Symposium*, 2006.
16. Christodorescu M, Kruegel C, Jha S. *Mining Specifications of Malicious Behavior*. ESEC/FSE'07, ACM Press: New York, NY, USA, 2007; 5–14.
17. Preda M, Christodorescu M, Jha S, Debray S. A semantics-based approach to malware detection. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*. 2007; 377–388.
18. Wang X, Jhi Y, Zhu S, Liu P. STILL: exploit code detection via static taint and initialization analyses. *Proceedings of Anual Computer Security Applications Conference (ACSAC)*, 2008.
19. Borders K, Prakash A, Zielinski M. Spector: automatically analyzing shell code. In *Proceedings of the 23rd Annual Computer Security Applications Conference*. 2007; 501–514.
20. Newsome J, Song D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. *Proceedings of Network and Distributed System Security Symposium*, 2005.
21. Krugel C, Kirda E. Polymorphic worm detection using structural information of executables. *2005 International Symposium on Recent Advances in Intrusion Detecion*, 2005.
22. Chung S, Mok A. Advanced allergy attacks: does a corpus really help. In *Recent Advances in Intrusion Detection (RAID)*, 2007.
23. Wang K, Parekh J, Stolfo S. Anagram: a content anomaly detector resistant to mimicry attack. *Proceedings of the Recent Advances in Intrusion Detection*, 2006.
24. Pedro N, Domingos P, Sumit M, Verma S. Adversarial classification. In *10th ACM SIGKDD Conference on Knowledge Discovery and Data mining*. 2004; 99–108.
25. Kong D, Jhi Y, Gong T, Zhu S, Liu P, Xi H. SAS: semantic aware signature generation for polymorphic worm detection. *Proceedings of International Conference on Security and Privacy in Communication Networks*, 2010.
26. Kong D, Tian D, Wu D, Liu P. SA3: Automatic semantic aware attribution analysis of remote exploits. *Proceedings of International Conference on Security and Privacy in Communication Networks*, 2011.
27. Collberg C, Thomborson C, Low D. A taxonomy of obfuscating transformations. *Technical Report 148*, University of Auckland, 1997.
28. 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.
29. Moore H. The Metasploit project. http://www.metasploit.com.
30. Bellman R. *Adaptive Control Processes: a Guided Tour*. Princeton University Press: Princeton, New Jersey, USA, 1961.
31. Meyn S, Tweedie R. *Markov Chains and Stochastic Stability*. Cambridge University Press: Cambridge, United Kingdom, 2005.
32. John A. R.A. Fisher and the making of maximum likelihood 1912–1922. *Statistical Science* 1997; **12** (3): 162–176.
33. Dempster A, Laird N, Rubin D. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society* 1977: 34–37.
34. Bertsekas D. *Nonlinear Programming*. Athena Scientific: Cambridge, MA, 1999.
35. Macaulay S. ADMmutate: Polymorphic shellcode engine. http://www.ktwo.ca/security.html.
36. Jemiscode. Jemiscodes—a polymorphic shellcode generator. http://www.shellcode.com.ar/en/proyectos.html.
37. Lyda R, Hamrock J. Using entropy analysis to find encrypted and packed malware. *IEEE Security and Privacy* 2007; **5**(2): 40–45.
38. Li Z, Sanghi M, Chen Y, Kao M, Chavez B. Hamsa: fast signature generation for zero-day polymorphic worms with provable attack resilience. *IEEE Symposium on Security and Privacy*, 2006.
39. Newsome J, Karp B, Song D. Polygraph: automatic signature generation for polymorphic worms. *IEEE Symposium on Security and Privacy*, 2005.