Signature Tree Generation for Polymorphic Worms

Yong Tang, Bin Xiao, *Member*, *IEEE*, and Xicheng Lu

Abstract—Network-based signature generation (NSG) has been proposed as a way to automatically and quickly generate accurate signatures for worms, especially polymorphic worms. In this paper, we propose a new NSG system—PolyTree, to defend against polymorphic worms. We observe that signatures from worms and their variants are relevant and a tree structure can properly reflect their familial resemblance. Hence, in contrast to an isolated view of generated signatures in previous approaches, PolyTree organizes signatures extracted from worm samples into a tree structure, called signature tree, based on the formally defined "more specific" relation of simplified regular expression signatures. PolyTree is composed of two components, signature tree generator and signature selector. The signature tree generator implements an incremental signature tree generation algorithm from worm sample clustering, up-to-date signature refinement to efficient tree construction. The incremental signature tree construction gives insight on how the worm variants evolve over time and allows signature refinement upon a new worm sample arrival. The signature selector chooses a set of signatures for worm detection from a benign traffic pool and the current signature tree constructed by the signature tree generator. Experiments show that PolyTree cannot only generate accurate signatures for polymorphic worms with noise, but these signatures are well organized in the signature tree to reflect the inherent relations of worms and their variants.

Index Terms—Signature tree, signature generation, polymorphic worm, sequence alignment.

1 Introduction

One of the most common and effective ways to detect worm attacks is to implement a signature-based intrusion detection system (IDS). An IDS samples suspicious data flows in a network with the goal of detecting the telltale byte sequences (or "signatures") of well-known or previously encountered worms. While these approaches are, in general, effective, they have two significant drawbacks. First, the signatures that they detect are produced manually by security experts only after a worm has already attacked a system and caused damage. Second, without accurate signatures, they cannot effectively detect polymorphic worms [1], that is, worms that change their infection attempts at what we call each "sample," each network flow to infect a computer on a network. Polymorphic worms do this by using one of the two classes of bytes, either *invariant* or *wildcard*. Invariant bytes have a fixed value and are present in every sample, and these are those bytes that carry the damaging infection. In contrast, wildcard bytes change their values in each sample. Since invariant bytes are composed of a number of invariant parts that are crucial to the exploitation of a vulnerable server [2], [3], we try to extract the invariant parts of polymorphic worms as their signatures. Fig. 1 presents one version of polymorphed Code Red II worm. This worm contains a sequence of seven *invariant parts*: "GET," ".ida?", "XX," "%u," "%u780," " = ", and "HTTP/1.0\r\n." Usually, there are some *distance restrictions* between adjacent invariant parts to ensure successful worm infections, like "'%u780' is 4 bytes after '%u'" in the CodeRed II worm.

To support the automatic and speedy generation of signatures of worms, especially polymorphic worms, the research community has proposed network-based signature generation (NSG), which uses a classifier or honeypot to collect worm samples, and extracts the invariant parts of a worm as its signature. Although a number of NSG approaches have been proposed in recent years, such as [4], [5], [6], [3], [7], the generated signatures by them may neglect two kinds of valuable information—the distance restrictions (like "'%u780' is 4 bytes after '%u'" in the CodeRed II worm) and one-byte invariant parts (like the symbol " = " in the Code Red II worm). According to the investigation by Kumar et al. [8], 28.57 percent of signatures in Snort encompass distance restrictions. After analyzing some Snort and Bro rules, we also found that a noticeable portion of signatures contains these two kinds of information, as Table 1 shows. Our recent investigations on Metasploit further show that distance restriction information is prevalently used in exploits and is important for accurate signature generation.[1] None of the previous NSG systems, however, extract distance restriction in signatures for polymorphic worms. Approaches in [4], [5], [6], [7] even ignored the order of invariant parts. Nonetheless, early NSG systems [4], [5], [9], [6] cannot extract one-byte invariant parts. Recent NSG systems [3], [7] tend to miss the one-byte invariant part as well.

- *Y. Tang and X. Lu are with the College of Computer, National University of Defense Technology, 615 research office, Changsha, Hunan 410073, China. E-mail: {ytang, xclu}@nudt.edu.cn.*
- *B. Xiao is with the Department of Computing, Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong. E-mail: csbxiao@comp.polyu.edu.hk.*

Manuscript received 28 Jan. 2008; revised 4 Dec. 2008; accepted 25 Mar. 2010; published online 8 June 2010.
Recommended for acceptance by S. Dolev.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2008-01-0037.
Digital Object Identifier no. 10.1109/TC.2010.130.

1. By searching functions of "rand_text, rand_text_alpha, rand_text_english, and rand_text_alphanumeric" in Metasploit that may create wildcard strings with fixed distance restrictions, we find that the fixed distance restriction appears 354 times in a total of 262 exploit source files.

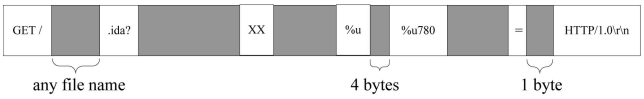

Fig. 1. One version of polymorphed Code Red II worm. Shaded content represents wildcard bytes and unshaded content represents invariant bytes.

TABLE 1
Analysis of Current IDS Rules

	Snort (exploit.rules file)	Bro (signatures.sig file)
total # of rules analyzed	155	2966
# (%) of rules with ordered multiple invariant parts	63 (40.6%)	365 (12.3%)
# (%) of rules with distance restriction	60 (38.7%)	312 (10.5%)
# (%) of rules with 1-byte invariant part	34 (22%)	86 (2.9%)

This paper presents a systematic approach—PolyTree, to automatic and accurate signature generation for polymorphic worms. The generated signatures are in the form of *Simplified Regular Expression (SRE)*, which extends Polygraph's token subsequence signature model with length constraints. We define a "*more specific than*" relation on this new signature model that allows two signatures to be compared to determine if one is a more specific form of the other. We use this relation to analytically define "*the most specific*" signature for a polymorphic worm, and hence, formalize the problem of signature generation for polymorphic worms. Based on the concept of the most specific signature, we proposed a signature generation algorithm using multiple sequence alignment. The generated signature is represented in SRE, which is effective and precise because of its successful one-byte invariant extraction and emphasis on the order and distance of extracted invariant parts.

The main novelty of PolyTree is that it organizes signatures extracted from worm samples into a tree structure, called *signature tree*. We observe that signatures from worms and their variants are relevant and a tree structure can properly reflect their familial resemblance. Rather than generating isolated signatures for multiple polymorphic worms, we propose to use the "more specific than" relation to organize generated signatures hierarchically into a well-defined signature tree. In this signature tree, each node is labeled with a signature and a signature of a child node must be *more specific than* the one of its parent node. The signature tree offers the following benefits: 1) the signature tree makes it simpler to balance the false positive rate and generalization ability of worm signatures and makes it easier to organize and maintain the generated signatures. For example, if a newly generated signature is added to the signature tree as a child node of an existing signature which has been submitted to IDS for detection, it is not necessary to submit this new signature to IDS since its parent signature can substitute for detection. 2) The signature tree gives insight on how the worm variants evolve over time. For example, if the signature of a new worm stays at the child node position to a known worm's signature in the signature tree, we can conjecture that this new worm might be a variant of the known worm. 3) The signature tree allows a quick and online signature refinement given a newly arrived worm sample. Previous approaches need to run signature generation algorithm again from a pool of all available worm samples whenever new samples arrive.

The PolyTree is composed of two components, signature tree generator and signature selector. The signature tree generator will construct a tree containing up-to-date signatures. We present an algorithm to incrementally generate signature tree without needing to repeat the signature generation process from the beginning. Upon encountering a new suspicious flow, the algorithm will first try to refine signatures generated using a Multiple Sequence Alignment (MSA) algorithm, and then the signature tree will be adjusted and updated if there are some signatures newly refined or inserted. More interesting, this process is also a worm sample clustering process. The majority samples from the same worm will be classified into the same node in the signature tree. We further show that given an adequate number of samples from a worm, its most specific signature will be generated in the signature tree. For worm detection, the signature selector will choose a set of signatures to IDSs based on a benign traffic pool and the current signature tree constructed by the signature tree generator.

We conducted experiments to evaluate the signature generation accuracy and worm traffic clustering of PolyTree based on five polymorphic worms and nine worm variants. Experimental results show that signatures of polymorphic worms can be incrementally refined in the signature tree and they were more accurate compared to previous NSG systems. The generated signatures were well organized in a signature tree to reflect the inherent relations of worms and their variants. In a sample pool of multiple worms, worm samples can be accurately clustered under the guidance of the signature tree. A further experiment demonstrates that our approach is more resilient against the Red Herring attack [10].

The rest of the paper is organized as follows: Section 2 describes related work. Section 3 illustrates the system architecture of PolyTree, consisting of the signature tree generator and signature selector. Section 4 presents how to generate accurate SRE signature for a single worm using multiple sequence alignment. Section 5 shows the implementation of signature tree generator, presenting a provable and incremental signature tree generation method. Section 6 shows the implementation of the signature selector. Section 7 evaluates PolyTree in the metric of signature accuracy and worm traffic clustering. Finally, Section 8 discusses the limitations and Section 9 draws the conclusion. Appendix analyzes the correctness of the signature tree generation algorithm and the runtime and attack resilience of PolyTree.

2 Related Work

In recent years, automatic signature generation for exploits, worms, and vulnerabilities has been an active subject and a number of automated worm signature generation systems have been proposed. These systems can be broadly classified as either host-based or network-based systems based on their detection mechanisms.

Host-based signature generation. Host-based signature generation systems reside in the protecting hosts and make use of host information to detect attempts by worms to infect systems and to generate signatures from these attempts. TaintCheck [11], Minos [12], Vigilante [13], and

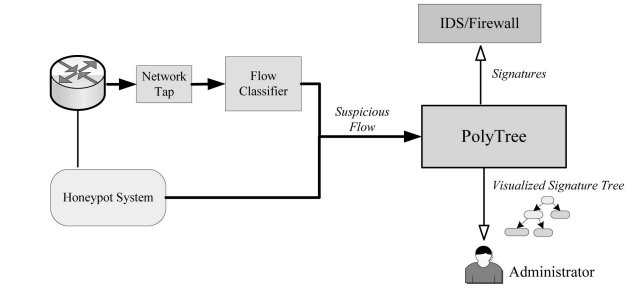

Fig. 2. Deployment of PolyTree.

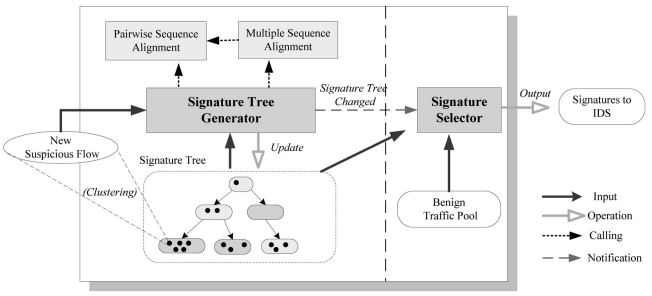

Fig. 3. Architecture of PolyTree.

DACODA [14] track dataflow derived from network to detect anomaly instruction execution. Some approaches [15], [16], [17] use address-space randomization (ASR) technique, and some [18] use instruction set randomization (ISR) technique to detect exploit attempts and then automatically generate signatures through forensic analysis between the related memory and input network message. Brumley et al. proposed a method [19] to identify a program's vulnerabilities, which uses static program analysis and only requires one single sample exploit. Wang et al. proposed packet vaccine [20], a method that randomizes address-like strings in packet payloads, and then carries out exploit detection, vulnerability diagnosis, and signature generation in one host. In general, host-based approaches can generate accurate signatures quickly but have a negative effect on the protecting host in terms of performance and configuration, e.g., the host must recompile the kernel and modify runtime libraries.

Network-based signature generation. Network-based signature generation systems solely analyze the suspicious traffic captured from network and output content-based signatures that reveal the residual similarity in the byte sequences of different polymorphic worm samples. Early network-based signature generation approaches including Honeycomb [4], Autograph [5], PAYL [9], and EarlyBird [6] produce single contiguous string signatures but these signatures often fail to match polymorphic worm payloads robustly [3]. Polygraph [3] and Hamsa [7] are two token-based approaches that try to select a set of tokens that has high coverage of the suspicious pool and low false positive reactions to the benign traffic pool. Polygraph may suffer from the fake anomalous flow attack discovered by Perdisci et al. [21]. While Hamsa improved over Polygraph in terms of speed and attack resilience; it may not work well if all tokens in the worm invariant are popular in benign traffic. Polygraph and Hamsa are constructed over token extraction (a token is a contiguous byte sequence with the minimum length and coverage in a suspicious pool). Although the minimum token length is a parameter for token extraction, it will not be set to 1. This is because if so, nearly every character will be extracted as a token for its frequent occurrence in a suspicious traffic pool. The token of a single character is useless (even harmful) for further signature construction (token selection). In fact, both systems request the length parameter to be at least two. Our signature generation approach belongs to network-based signature generation systems, which differs from others by successful one-byte invariant extraction, emphasis on the distance restriction of extracted invariant parts, the ability of incrementally refining signature, and signature tree construction to illustrate relations of worms and their variants.

3 Architecture of PolyTree

3.1 Deployment

Fig. 2 depicts a typical deployment of PolyTree that is similar to other network-based approaches, like Autograph [5], Polygraph [3], Hamsa [7], and LESG [22].

PolyTree first captures suspicious traffic from network via the following two means: 1) Through *flow classifier*. PolyTree sniffs traffic from networking devices (such as routers, switchers, and gateway) through a network tap. Network traffic will be reassembled, transforming packets into contiguous byte flows. The flows that are identified as suspicious or anomalous by a flow classifier will be sent to PolyTree for signature generation. Existing flow classifiers include [9], [5], [23], [24]. 2) Through *honeypot*. Since any traffic to the honeypot is suspicious, we can effectively collect suspicious traffic by Honeypot. So far, there are a number of Honeypot/Honeyfarm systems [4], [25], [26], [27], [28] that could be used to collect suspicious traffic. Note that our PolyTree is noise tolerant, which means that the flow classifier or honeypot system is allowed to be imperfect.

The captured suspicious samples are delivered to PolyTree system one by one for online signature generation. PolyTree incrementally constructs a signature tree that consists of all generated signatures. When a new suspicious flow is captured, PolyTree will classify it into a matching node (sample clustering), then create or refine signatures. When the signature tree is changed due to new sample arrivals, we select a set of signatures from the signature tree and submit them to IDSs for detection. The signature tree provides a more efficient way to organize and maintain the generated signatures. Besides, the signature tree gives insight on how the worm variants evolve in time for network administrators.

3.2 Architecture

Fig. 3 depicts the architecture of PolyTree, which consists of two main components—*signature tree generator* and *signature selector*.

Signature tree generator. This component is the core of PolyTree that implements the signature tree construction algorithm in Section 5. The algorithm will carry out operations from sample clustering, signature refinement to the final signature tree construction. When a new suspicious flow is captured, PolyTree classifies it into an appropriate node in the signature tree (sample clustering), tries to create

more specific signatures (signature refinement), and updates the signature tree (signature tree construction).

Signature selector. This component implements the signature selection algorithm in Section 6, which selects a set of signatures from the signature tree and submits them to IDSs for detection upon request or whenever the signature tree is changed.

4 Generating SRE Signature for Single Polymorphic Worm

In this section, we focus on generating accurate signatures for a single polymorphic worm. We will first propose a more precise signature type, the SRE signature, and define what is a more specific signature. We will then define the signature generation problem as calculating the most specific signature of a polymorphic worm. Finally, we will briefly introduce a signature generation method based on MSA.

4.1 Signature Type—SRE Signature

Motivated by the insufficiency of current signature types for expressing distance restriction, we extend the Polygraph's token-subsequence signature type with length constraints and propose a new signature type—SRE signature. We design the SRE signature type from regular expression. It is believed that regular expression has significant advantages for intrusion detection in terms of flexibility, accuracy, and efficiency [29], [8]. Regular expressions have been widely used in intrusion detection systems, for example, in Snort and Bro. However, the full regular expression is too complex and its numerous syntax rules are not needed for worm detection. Hence, we introduce the simplified regular expression as a way of representing worm signatures.

An SRE signature is a simplified form of a regular expression that contains only two qualifiers, ".*" and ".{k}." These can each be further abbreviated by replacing ".*" with "*", which represents an arbitrary string (including a zero-length string), and by replacing ".{k}" with "[k]," which represents a string consisting of k arbitrary characters. For example, "'one'[2]'two'*" is an SRE signature that is equivalent to the regular expression "one.{2}two.*". Suppose that $\Phi = \{*, [k]\}$ is the set of the two qualifiers and Σ^+ is the set of nonempty strings over a finite alphabet Σ. An SRE signature is defined as follows:

Definition 1 (SRE signature). *An SRE signature is a signature in the form of* $(q_0)s_1q_1s_2 \ldots q_{k-1}s_k(q_k)$, *where* $q_i \in \Phi$ *is a* qualifier, $s_i \in \Sigma^+$ *is a* substring $(i \in [0, k])$, *and* (q_0) *and* (q_k) *mean* p_0 *and* q_k *are optional.*

We define the length of an SRE signature as the total number of characters in substrings plus the number of qualifiers, and use $|\mathcal{X}|$ to denote the length of an SRE signature $\mathcal{X}$. For example, given an SRE signature $\mathcal{X} =$ "*'a'[2]'bbb'[1]'cccc'*", $|\mathcal{X}| = 11$ $(1 + 2 + 3 + 1 + 4)$.

Compared with the previous signature types to be used for worm detection, the SRE signature type is a more precise signature presentation because it can express distance restrictions of adjacent invariant parts using *qualifiers* (e.g., $[k]$ shows the distance of k bytes). In addition, SRE signatures, in the form of regular expressions, can be easily converted into existing IDS rules, and vice versa.

4.2 Problem Definition

We aim to generate more accurate signatures for polymorphic worms. It is natural to ask what is "a more accurate" and what is "the most accurate" signature for a polymorphic worm. Intuitively, "the most accurate" signature should be "the most specific," that is, in a balance of specific and general: specific—it contains as many features of the worm as possible so that it will not lead to false positives; general—it does not contain any useless or incorrect features of the worm so that it will not lead to false negatives. We try to answer these questions by formally defining the *more specific than relationship* (Definition 3) and *the most specific signature* (Definition 5) for a polymorphic worm.

Definition 2 (Containment, $\triangleleft$). *Let* $\mathcal{X}$ *and* $\mathcal{Y}$ *be two SRE signatures, we say that* $\mathcal{Y}$ contains $\mathcal{X}$, *denoted by* $\mathcal{X} \triangleleft \mathcal{Y}$, *if* $L(\mathcal{X}) \subseteq L(\mathcal{Y})$. *That is, the strings that match signature* $\mathcal{X}$ must *also match signature* $\mathcal{Y}$.

Definition 3 (More specific than, $\prec$). *Let* $\mathcal{X}$ *and* $\mathcal{Y}$ *be two SRE signatures. If* $\mathcal{X} \triangleleft \mathcal{Y}$ *and* $|\mathcal{X}| > |\mathcal{Y}|$, *we say that* $\mathcal{X}$ *is* more specific than $\mathcal{Y}$, *or* $\mathcal{Y}$ *is* more general than $\mathcal{X}$, *and this is denoted by* $\mathcal{X} \prec \mathcal{Y}$.

Here are some examples for Definition 2 and Definition 3: "'abc'*'bcd'" $\triangleleft$ "'ab'*'cd'", "'ab'[2]'cd'" $\triangleleft$ "'ab'*'cd'"; "'aaa'*'b'" $\prec$ "'aa'*'b'", "'aa'[3]'c'" $\prec$ "'aa'*'c'". Note that if a network flow f (as a special SRE signature with only one substring and without qualifiers) matches an SRE signature $\mathcal{X}$, we also use "$f \triangleleft \mathcal{X}$" to denote it. For instance, the flow "abcdef" matches "'ab'*'ef'" can be denoted by "abcdef" $\triangleleft$ "'ab'*'ef'".

Definition 4 (Signature of a polymorphic worm, $\lessdot$). *Given a polymorphic worm* $\mathbf{w}$, *if all possible samples of* $\mathbf{w}$ *match an SRE signature* $\mathcal{X}$, *then* $\mathcal{X}$ *is* a signature of $\mathbf{w}$, *and this is denoted by* $\mathbf{w} \lessdot \mathcal{X}$.

Definition 5 (The most specific signature of a polymorphic worm, MSSig). *Given a polymorphic worm* $\mathbf{w}$ *and its SRE signature* $\mathcal{X}$, *if* $\mathbf{w} \lessdot \mathcal{X}$ *and for any other SRE signature* $\mathcal{X}'$ *such that both* $\mathbf{w} \lessdot \mathcal{X}'$ *and* $\mathcal{X} \triangleleft \mathcal{X}'$ *hold, then* $\mathcal{X}$ *is* the most specific signature $(MSSig)$ *of* w, *and this is denoted by* $\mathcal{X} = MSSig(w)$.

From the above definitions, now we can answer "what is a more accurate signature" for a polymorphic worm and are able to compare the accuracy of two signatures. Suppose that both $\mathcal{X}$ and $\mathcal{Y}$ are signatures of a polymorphic worm w ($w \lessdot \mathcal{X}$ and $w \lessdot \mathcal{Y}$). Signature $\mathcal{X}$ is more accurate than $\mathcal{Y}$ if and only if $\mathcal{X} \prec \mathcal{Y}$. Given a number of samples of a polymorphic worm, the signature generation problem is formalized by Problem 1.

Problem 1 (The most specific signature generation for a single polymorphic worm).
INPUT: $s_1, \ldots, s_n$ are n samples of a polymorphic worm w.
OUTPUT: A signature $\mathcal{X}$ such that $\mathcal{X} = MSSig(w)$.

Given the polymorphic Code Red II worm in Fig. 1, we can convert the generated signatures by previous NSG methods into our defined SRE signature format. Honeycomb [4] outputs "*'.ida?'*". Polygraph [3] outputs "'GET / '*'.ida?'*'XX'*'%u'*'%u780'*'HTTP/1.0\r\n\r\n'". Hamsa [7] outputs "'GET/'*'.ida?'*'%u'*'%u780' *'HTTP/1.0\r\n'*". If these SRE signatures are denoted by $\mathcal{X}_1, \mathcal{X}_2$, and $\mathcal{X}_3$, all

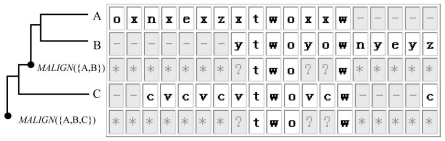

Fig. 4. Alignment result from multiple sequence alignment.

of them are signatures of the worm according to Definition 4. However, none of them is the most specific signature according to Definition 5 because given an SRE signature $\mathcal{Y} =$ "'GET /'*'.ida?'*'XX'*'%u'[4]'%u780'*'= '[1]'HTTP/1.0\r\n'", obviously, $\mathcal{Y} \prec \mathcal{X}_1$, $\mathcal{Y} \prec \mathcal{X}_2$, and $\mathcal{Y} \prec \mathcal{X}_3$. That is, $\mathcal{Y}$ is more specific than $\mathcal{X}_1, \mathcal{X}_2, \mathcal{X}_3$. This implies that the signatures generated by the previous methods are not "the most accurate." There is still room for improvement.

4.3 Signature Generation Using Multiple Sequence Alignment

The sequence alignment technique has been widely used to quantify and visualize similarity between sequences. Using this technique, we can arrange worm samples such that the same invariant bytes in worm samples are located in the same columns. For example, suppose that we have three worm samples where sample $A =$ "oxnxexzxtwoxxw," sample $B =$ "ytwoyownyeyz," and sample $C =$ "cvcvcvtwovcwc." From sequence alignment, Fig. 4 illustrates that the bytes in "two" and "w" are believed to be invariant bytes of the worm, and hence, are put in the same columns. The alignment will produce the result sequence "*******?'two'??'w'*****", which can be converted into an SRE signature "*'two'[2]'w'*".

In the rest of the paper, we use $Align(.,.)$ to denote the pairwise sequence alignment function that will output an SRE signature from two samples, and use $MAlign(.)$ to denote the multiple sequence alignment function that will output an SRE signature from a set of samples. We exploit multiple sequence alignment to generate signature for a single polymorphic worm. That is, if F is a set containing enough samples of a polymorphic worm w, we aim to obtain $MAlign(F) = MSSig(w)$. To achieve this, we proposed a new pairwise alignment algorithm and a new multiple sequence alignment algorithm [30]. Suppose that there are n samples, each having a length l, the time complexity of the new pairwise alignment algorithm and new multiple sequence alignment algorithm is, respectively, $O(l^2)$ and $O(nl^2)$. Though we are able to generate SRE signatures using MSA algorithm [30], [31] to support syntax like "at most k characters" or "the distance range $[k_1, k_2]$," we do not include such syntax in the SRE signatures because it could remarkably decrease the worm detection accuracy when worm samples are not adequate and it could make the SRE signature generation system be the victim of malicious attacks, which will be discussed in Section 8 and Appendix C.

To quantitatively evaluate the effectiveness of our multiple sequence alignment algorithm, suppose that w is a polymorphic worm and F is a sample set of w, we say that the multiple alignment algorithm $MAlign(.)$ outputs a "good alignment" if $MAlign(F) = MSSig(w)$ that is the most specific signature. In Section 7.2, we report that in the tested experiments, 96.8 percent of alignments are "good alignments" for the MSA algorithm.

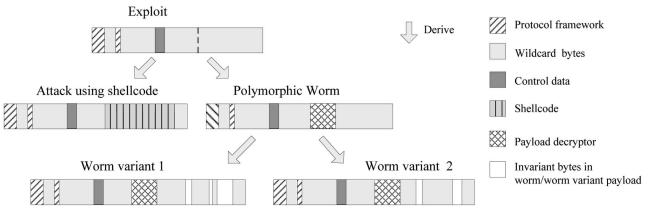

Fig. 5. The familial resemblance of the byte sequences of exploits, shellcode attacks, polymorphic worms, and worm variants—the motivation to generate the signature tree.

5 Generating Signature Tree for Multiple Polymorphic Worms

In this section, we consider the common situation in which the flows captured by an NSG system contain mixed samples of multiple polymorphic worms (perhaps accompanied by noise flows). We propose the idea of signature tree and describe an incremental signature tree generation algorithm for the signature tree construction. From the constructed signature tree, accurate signatures can be selected to IDS for worm detection.

5.1 Motivation and Problem Definition

Polymorphic worms and their variants can exhibit familiar resemblance in their infection attempts. A vulnerability can be abused by a number of different exploits [19], all of which can be manipulated to design polymorphic worms or shellcode attacks. A polymorphic worm can potentially mutate into variants. For example, the Sasser worm, Mocbot worm, and Agobot worm all propagate by exploiting the MS04-011 vulnerability. Sasser has seven variants and Mocbot has 10. Agobot has more than 480 variants. Considering these facts, we use a tree structure to represent the familial resemblance of the byte sequences of these attacks (exploits, shellcode attacks, polymorphic worms, and worm variants) in Fig. 5. According to the Epsilon-Gamma-Pi model [2], an exploit at least contains some invariant parts residing in protocol frameworks (typically in the ϵ part), such as "HTTP," and bogus control data (typically in the γ part), such as jump instructions. After an exploit succeeds, the victim host will execute a shellcode (for shellcode attacks) or a malicious code (for worms). As a result, there will be some invariant parts residing in every instance of a shellcode attack or a worm. Nevertheless, a polymorphic worm variant not only contains the same invariant parts as its parent worm, it also contains some additional invariant bytes to carry out specially designed malicious activities.

Given that we can generate signatures of these attacks by extracting their invariant parts and any two signatures can be compared in terms of specifics according to definitions in Section 4.2; can we also organize the generated signatures into a tree so as to represent the familial resemblance of the byte sequences of these attacks? One obvious benefit of the tree organization would be that the (signature) tree would reflect and illustrate logical relations between these attacks and shed light on how worm variants evolve over time. For example, if the signature of a new attack is a child node of a known worm's signature in the signature tree, we immediately know that this new attack should be a variant of the known worm. We propose the definition of a signature tree as follows:

TABLE 2
The Samples Used for the Demonstration in Fig. 7

w_1	*ONEaTWOaaTHREE*	w_5	*ONEdTWOddddddddd*	w_9	*ONEgFOUR*
w_2	*ONEbTWObbbTHREE*	w_6	*ONEeTWOeeeeeeeee*	w_{10}	*ONEhhhhFOUR*
w_3	*ONEcTWOccTHREE*	w_7	*ONEfTWOfffffffff*	w_{11}	*ONEiiiiiiFOUR*
w_4	*dddd*	w_8	*ONEggg*		

Definition 6 (Signature tree). *A* Signature Tree *(*STree*) is a rooted tree where each node comprises a set of network flows and is labeled with an SRE signature. The root is labeled with the most general signature "*". Each link to a child represents a* more specific than *relationship. Let $C.sig$ denote the signature of node C. A signature tree must have the following two properties:*

1. *that each sample $f \in$ node C implies $f \triangleleft C.sig$ and*
2. *node C_1 is the child of node C_2 implies $C_1.sig \prec C_2.sig$.*

From Definition 6, we know that the construction of a signature tree must satisfy two requirements. First, each child node signature is more specific than its parent node signature ($C_1.sig \prec C_2.sig$) in the signature tree. Second, the signature tree can classify samples into matching nodes. A sample f is classified into a node C ($f \in C$) if and only if it can match the signature of this node ($f \triangleleft C.sig$). For instance, the subfigure (j) in Fig. 7, as a classification result, shows a generated signature tree, where 11 samples ($w_1 - w_{11}$) given by Table 2 are clustered into five distinct nodes ($C0 - C4$).

Given a signature tree $STree$ and its n nodes (C_i, $i = 1, \ldots, n$), a node C_i is a two tuple $\{C_i.sig, C_i.flow\}$, where $C_i.sig$ is the extracted signature for all flows in the set $C_i.flow$ classified into node C_i. The $STree$ can be represented as a two tuple $\{STree.sig, STree.flow\}$, where $STree.sig$ is the signature set with $STree.sig = \cup_{i=1}^{n}\{C_i.sig\}$ and $STree.flow$ is a set containing all flows with $STree.flow = \cup_{i=1}^{n} C_i.flow$. We denote a flow set $F \subseteq STree$ if for each $f \in F$, $f \in STree.flow$. Given a signature $\mathcal{X}$, if $\mathcal{X} \in STree.sig$, we say that $\mathcal{X} \in STree$; a signature set $\mathcal{S} \subseteq STree$ if each signature in $\mathcal{S}$ is in the $STree$.

We define the signature tree generation problem for multiple polymorphic worms as below, where θ is the minimum required number of samples to generate an accurate signature for a single polymorphic worm. Note that: first, all samples in the suspicious traffic pool $\mathcal{M}$ should be clustered into signature tree nodes ($\mathcal{M} \subseteq STree$); second, *the most specific signature* of each polymorphic worm should be generated and included in the signature tree (for each $w_i (i = 1, \ldots, n)$, there exists a node C in $STree$ and $C.sig = MSSig(w_i)$). In other words, a signature tree generation method should incorporate the sample clustering process and the most specific signature generation process.

Problem 2 (Signature tree generation).
INPUT: *suspicious traffic pool* $\mathcal{M}$, where $W_1, W_2, \ldots, W_n \subset \mathcal{M}$ are the sample sets of worms $w_1, w_2, \ldots, w_n$, and $|W_i| > \theta (i = 1, \ldots, n)$.
OUTPUT: *A signature tree $STree$ such that $\mathcal{M} \subseteq STree$ and for each $w_i (i = 1, \ldots, n)$, there exists a node C in $STree$ and $C.sig = MSSig(w_i)$.*

5.2 Incremental Signature Tree Generation

Here, we describe the signature tree generation method that is implemented in the signature tree generator in the PolyTree system. This method seeks to generate a tree as a solution to Problem 2, by incrementally updating a fixed signature tree—initially, the signature tree contains only the root node with *the most general signature* "*" (any flow can match "*"). When a new suspicious flow (worm sample) is captured, the signature tree update algorithm in Algorithm 1 will be in effect. In this algorithm, f is the new captured sample, θ is the split parameter that denotes the minimum required number of samples from which we can generate an accurate new signature.

Algorithm 1. *SigTreeUpdate*(f, θ)
1. Find a node C_{belong} in the signature tree such that $f \triangleleft C_{belong}.sig$ and C_{belong} is in the highest level (has the biggest depth). If there are several such nodes, we choose the one added into this level earliest.
2. $C_{belong} \leftarrow C_{belong} \cup \{f\}$ /* step 1. sample clustering */
3. *if* C_{belong} has a subset $C_{new} = \{f_1, f_2, \ldots, f_\theta\}$ such that $MAlign(C_{new}) \prec C_{belong}.sig$, *then*
   /* step 2. signature refinement */
   /* step 3. signature tree construction*/
   3.1: $C_{belong} \leftarrow C_{belong} \setminus C_{new}$;
   add C_{new} as a child of C_{belong} in the signature tree;
   // split a new node C_{new} from C_{belong}
   $C_{new}.sig \leftarrow MAlign(C_{new})$;
   SinkSamples(C_{new}) ;
   3.2: *for all of* node C *such that* $C \neq C_{new}$, $C.level = C_{new}.level$ and $C.sig \prec C_{new}.sig$, *do*
   MoveSubtree(C, C_{new}); // move C to C_{new} as a child
   3.3: *if* C is a child of C_{belong} and $Align(C.sig, C_{new}.sig) \prec C_{belong}.sig$, *then*
   3.3.1: create a new node $C_{parent} = \{\}$ as a child of C_{belong};
   $C_{parent}.sig \leftarrow Align(C_{new}.sig, C.sig)$;
   3.3.2: *MoveSubtree*(C_{new}, C_{parent}) ;
   MoveSubtree(C, C_{parent});
   // put C_{new} and C as children of C_{parent}
   3.3.3: *SinkSamples*(C_{parent});

The signature tree update algorithm incorporates three steps to deal with the new sample f. The first step is *sample clustering*. A node C_{belong} is selected to contain this sample if $f \triangleleft C_{belong}.sig$. The second step, *signature refinement* (sentence 3), tries to generate a more specific signature than the old signature $C_{belong}.sig$ using the samples within the node C_{belong}. If so, we enter the third step, i.e., *signature tree construction*. In this step, a new node C_{new} labeled with the newly generated signature will be created (sentence 3.1), and then the signature tree will possibly be adjusted (sentences 3.2 and 3.3) by two operations, *SinkSamples* and *MoveSubtree*, to maintain certain desirable properties detailed in Appendix A.2, like "more specific signature should be on a higher level," "the samples in the signature tree tend to *sink* to nodes in a higher level." The definitions of *SinkSamples* and *MoveSubtree* operations are given below, where C_T and C_S are nodes in the signature tree $STree$.

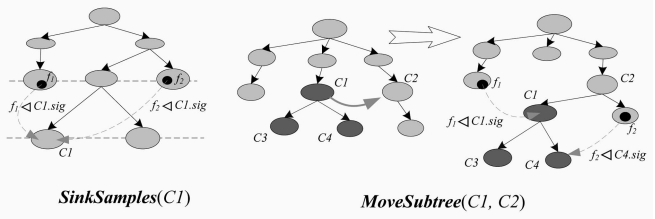

Fig. 6. The *SinkSamples* and *MoveSubtree* operations.

SinkSamples (C_T): A sample f is moved from C to C_T if $f \in C$, $C.level = C_T.level - 1$, and $f \triangleleft C_T.sig$,.

MoveSubtree (C_S, C_T): In the signature tree, the subtree rooted at C_S is moved to the node C_T (becoming a child of C_T). For each node C' in the subtree C_S, *SinkSamples(C′)* is executed.

Fig. 6 illustrates the above two operations. The illustration of *SinkSamples*($C1$) shows that the samples (f_1 and f_2) that can match $C1.sig$ in the level $C1.level - 1$ are moved (*sink*) to the node $C1$. The illustration of *MoveSubtree*($C1$, $C2$) shows that, first, the subtree rooted at $C1$ is moved to $C2$; then, the execution of SinkSamples(C1) results in the sample f_1 sinking to $C1$ since $f_1 \triangleleft C1.sig$, and the execution of SinkSamples(C4) results in f_2 sinking to $C4$ since $f_2 \triangleleft C4.sig$.

Appendix A.1 proves the correctness of the incremental signature tree generation algorithm. Appendix A.2 presents its two significant properties. Those are as follows: the majority samples from one polymorphic worm can be correctly clustered into the same node (Theorem 2), and if the performance of signature generation for a single polymorphic worm is good (Assumption 1), then the final generated signature tree will contain the most specific signature for each encountered polymorphic worm given an adequate number of worm samples were collected (Theorem 3).

5.3 Demonstration of the Execution of Incremental Signature Tree Generation

Fig. 7 shows an example to execute the signature generation with the parameter $\theta = 3$. Table 2 lists 11 samples that are delivered sequentially to Algorithm 1 as inputs. The main steps are as follows:

1. Initialize signature tree by creating the root $C0$, $C0.sig =$ "'*'".
2. Sequentially add samples w_1, w_2, w_3 to $C0$.
3. Split new node $C1$ from $C0$, since $MAlign(\{w_1, w_2, w_3\}) =$ "'ONE'[1]'TWO'*'THREE'" $\prec C0.sig$.
4. Since $w_4, w_5, w_6, w_7 \triangleleft C0.sig$, they are sequentially added to $C0$.
5. Since $MAlign(w_5, w_6, w_7) =$ "'ONE'[1]'TWO'*" $\prec$ "*", split a new node $C2$ from $C0$.
6. Because $C1.sig =$ "'ONE'[1]'TWO'*'THREE'" $\prec C2.sig =$ "'ONE'[1]'TWO'*", according to sentence 3.2, move the subtree rooted at $C1$ to $C2$ through the *MoveSubtree* operation.
7. Add w_8, w_9, w_{10}, w_{11} to $C0$, and then create new node C_3 containing samples w_9, w_{10}, w_{11}. $C3.sig =$ "'ONE'*'FOUR'".

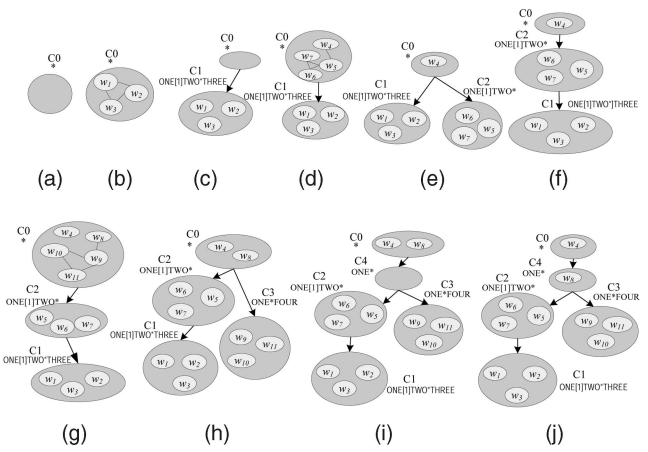

Fig. 7. Demonstration of the incremental signature tree generation process.

8. According to sentence 3.3, a new node $C4$ is created as the parent of $C2$ and $C3$. $C4.sig = MAlign(C2.sig, C3.sig)$="'ONE'*".
9. After the execution of $SinkSamples(C4)$ (sentence 3.3.3), w_8 is moved from $C0$ to $C4$ since $w_8 \triangleleft C4$ and $C0.level = C4.level - 1$.

From this example, we can witness the ability of the signature tree to cluster samples from a new worm, like w_9, w_{10}, w_{11}. The signature generated from the clustered worm samples can be selected to IDSs to detect future malicious traffic from the worm.

6 Signature Selection to IDS

Recall that the generated SRE signatures by PolyTree can be easily applied to current IDSs that support regular expressions, such as Snort and Bro. When the signature tree is constructed or dynamically updated, we need to select a set of signatures from the signature tree and submit them to IDSs for worm detection. Newly selected signatures are likely to be the signatures from new worms. We call this process as *signature selection* and the component that implements this process in PolyTree is the signature selector.

Suppose that $\mathcal{M}$ is the set of all suspicious flows captured by PolyTree, $\mathcal{N}$ is a benign traffic pool. Note that obtaining $\mathcal{N}$ is not easy, and in previous papers (like [3], [7]), $\mathcal{N}$ is collected from a "trusted" network. Strictly speaking, the traffic from a "trusted" network should be "normal traffic" or "near benign traffic" (contains some or no attack traffic) instead of "benign traffic" (contains no attack traffic). Signature selection is to select the smallest signature set $\mathcal{S}$ that covers most flows in $\mathcal{M}$ (low false negative) but very few in $\mathcal{N}$ (low false positive). Let $\mathrm{DR}_{\mathcal{S}}$ denote the detection rate (true positive) of $\mathcal{S}$ in $\mathcal{M}$ and $\mathrm{FP}_{\mathcal{X}}$ denote the false positive rate of a signature $\mathcal{X}$ in $\mathcal{N}$. The signature selection is formalized in Problem 3.

Problem 3 (Signature selection).
INPUT: Signature tree $STree$; benign traffic pool $\mathcal{N}$; upper limit of false positive $\rho < 1$.
OUTPUT: A set of signatures $\mathcal{S} \subseteq STree$ such that $\mathrm{DR_S}$ is maximized and $|S|$ is minimized subject to $\mathrm{FP}_{\mathcal{X}} \leq \rho$ for each $\mathcal{X} \in \mathcal{S}$.

Algorithm 2 describes how to select a signature set S from an $STree$. The selected signatures must satisfy a given FP upper limit ρ. The $STtree$ has the property that a parent signature has a larger detection rate than its descendent signatures (a parent signature is more general than its descendent signatures, and thus, has a higher detection rate). Thus, our strategy is to choose the parent signature instead of its descendent signatures if all of them are below the FP upper limit. The chosen parent signatures in the signature tree traversal, in turn, can maximize the overall detection rate for malicious flows.

Algorithm 2. SignatureSelection($STree, \mathcal{N}, \rho$)

1. *Initialization*
   for each node C in $STree$, calculate the false positive $\mathrm{FP}_{C.sig}$ using benign traffic pool $\mathcal{N}$;
   $S \leftarrow \{r.sig\}$, where r is the root of $STree$;
2. *do*
   for each $C.sig \in S$ such that $\mathrm{FP}_{C.sig} > \rho$, *do*
   $S \leftarrow S - \{C.sig\}$;
   if $C_1, C_2, \ldots, C_k$ are the children of C in $STree$, *then*
   $S \leftarrow S \cup \{C_1.sig, C_2.sig, \ldots, C_k.sig\}$
   until S does not change anymore;

By combining Algorithm 1 and Algorithm 2, PolyTree can achieve a better trade-off between false positive and false negative: on one hand, in the signature (and signature tree) generation process, we try to generate more specific signatures by MSA, which aims to reduce false positives. On the other hand, in the signature selection process, we prefer to select more general signatures to minimize false negatives.

7 Evaluation

In this section, we use synthetic polymorphic worm samples that are derived from real-world exploits to evaluate the effectiveness of our incremental signature tree generation algorithm and test the quality of output signatures in the PolyTree.

7.1 Methodology

Since there are no well-known polymorphic worm samples available on the Internet, most NSG methods (like Polygraph [3] and Hamsa [7]) evaluated their approaches using synthetically generated polymorphic worm samples based on real-world exploits. We follow the same evaluation principle. One benefit of using synthetic polymorphic worm samples is that we can know the most specific signature for each worm in advance. Hence, we can measure the accuracy of the generated signatures by comparing them with the most specific signatures.

We tested PolyTree by using two traffic pools, a suspicious flow pool containing samples of multiple polymorphic worms and a benign traffic pool. PolyTree sequentially and randomly fetched samples from the suspicious flow pool and updated the signature tree accordingly. A signature tree was constructed and a set of signatures was selected after all samples in the suspicious flow pool have been added into the signature tree. We then created another 1,000 worm samples to test the false negatives and used part of the benign traffic to test the false positives. All experiments were executed on a PC with a single 3.0 GHz Intel Pentium IV processor and 1 GB memory, running Windows XP SP2. Details of the polymorphic worm workload, the benign traffic data, and how we evaluated the clustering quality are as follows:

Polymorphic worm workload. Polygraph used synthetic worm samples from three pseudopolymorphic worms. Hamsa used synthetic worm samples from five to evaluate their methods. In our experiments, the synthetic worm samples are from 14 pseudopolymorphic worms or worm variants. Five pseudoworms include four (based on Apache-Knacker exploit, ATPhttpd exploit, BINDTSIG exploit, and Code Red II worm) developed by Polygraph [3] and one (based on IISprinter exploit) developed by us. Nine worm variants include three derived from ATPhttpd worm (ATPhttpd.A, ATPhttpd.B, and ATPhttpd.C) and six derived from CodeRed II (CodeRed.A, CodeRed.B, ..., CodeRed.F). The worm variants add one or several extra invariant parts into their worm samples comparing to their original worms. In practice, worm variants usually have extra payloads comparing to its original worm due to imperfect polymorphic engines.

Benign traffic data. Except BIND-TSIG worm, which uses the binary-based DNS protocol, all synthetically created worms in this paper use the HTTP protocol. For this reason, our focus is on the traffic data under the HTTP and DNS protocols. We collected the "benign" traffic data from three sources. First, we collected a 5-day HTTP trace (45,111 flows, 15 GB) from our campus network gateway. Second, we used a 2-day DNS trace, taken from a DNS server that served in an academic institution's domain. Third, we used the first week's network TCP dump data from the 1999 DARPA Intrusion Detection Evaluation Data Sets [32] (10 GB).

Evaluation of clustering quality. Note that our signature tree generation algorithm is able to do the worm traffic clustering to classify samples from the same worm or worm variant into the same cluster. To quantitatively evaluate this clustering quality, we applied a widely used metric: F-measure [33]. We manually tagged each sample with a conjecture indicating the worm name. A sample might have multiple conjectures. For example, because ATPhttpd.A is a variant of ATPhttpd worm, a sample of ATPhttpd.A will be labeled with "ATPhttpd.A" as well as "ATPhttpd."

Let C be the set of all clusters, J be the set of all possible conjectures, N_j be the number of samples labeled with conjecture j, and N be the number of total samples. Let c_j be the set of samples in the cluster c labeled with conjecture j, $|c|$ be the number of samples in the cluster c, and $|c_j|$ be the number of samples in the cluster c with the conjecture j. The F-measure cluster evaluation method combines two concepts from information retrieval, precision and recall. For cluster c and conjecture j, the Recall R and Precision P are calculated by: $R(j,c) = |c_j|/N_j$, $P(j,c) = |c_j|/|c|$. The F-Measure of cluster c and conjecture j , denoted by $F(j,c)$, is defined as

$$F(j,c) = \frac{2R(j,c)P(j,c)}{R(j,c) + P(j,c)}. \quad (1)$$

The overall F-Measure value is computed by taking the average of all $F(j,c)$ as (2) gives. The F-measure values are in the interval [0, 1] and a larger F-measure value indicates a higher clustering quality:

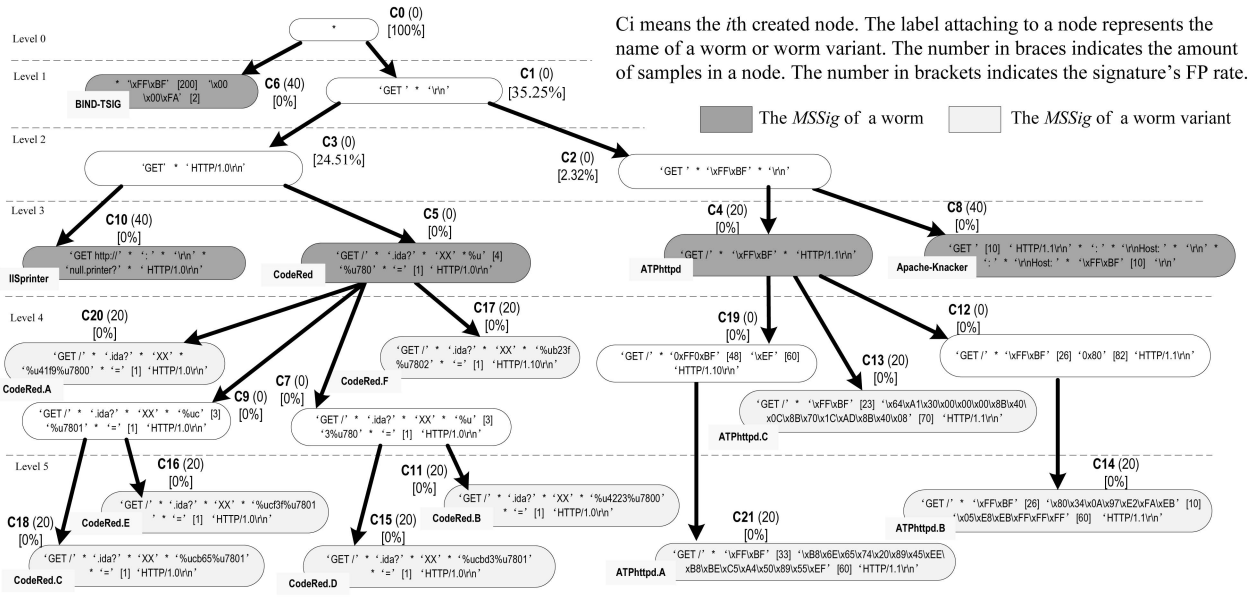

Fig. 8. The generated signature tree given five polymorphic worms and nine polymorphic worm variants ($\theta = 6$).

$$F(C) = \sum_{j \in J} \frac{N_j}{N} max_{c \in C} F(j, c). \quad (2)$$

7.2 Performance of Sequence Alignment Algorithm

To study the performance of our MSA algorithm [30], in each iteration, we randomly created θ samples of a polymorphic worm w (w is one of the five worms), and then we generated a signature by aligning these samples. If the generated signature is $MSSig$ (the most specific signature) of w (recall that we know the $MSSig$ for each worm in advance), then it is a "good alignment." We did this experiment 8,500 times and found that the rate that the algorithm outputs "good alignments" ($MAlign(.) = MSSig(w)$) is 96.8 percent (8,229/8,500) for $\theta = 6$. This suggests that the MSA algorithm has a good performance for accurate signature generation. The order of sequence selection for MSA has little influence on the signature generation accuracy.

7.3 Multiple Polymorphic Worms without Noise

We first consider a situation wherein the worm samples contain no noise samples. We used 320 samples from five worms and nine worm variants—40 from each Apache-Knacker, BIND-TSIG, and IISprinter; 20 from each ATPhttpd, ATPhttpd.A, ATPhttpd.B, and ATPhttpd.C; and 20 from each variant from CodeRed.A to CodeRed.F.

Clustering quality. Fig. 8 shows the generated signature tree under the split parameter $\theta = 6$. We can see that the signature tree is almost perfectly constructed. Since the worm samples are randomly created without any noise samples, the result is consistent to Theorems 2 and 3 in Appendix A.2, that is, the worm samples are well clustered and the signature tree contains all $MSSig$s for given worms and variants.

Table 3 presents how the split parameter θ could affect the F-Measure value and total number of nodes in the generated signature tree. The higher θ value means more samples are required to generate a new signature using MSA. Too low a θ value would lead to many overspecific signatures. For example, we observe that when $\theta = 2$, PolyTree generated a signature tree with 122 signatures, none of which are the most specific signatures of the 14 worms or worm variants. In contrast, too high a θ value would result in PolyTree being insensitive. For example, capturing five samples of a novel polymorphic worm will not be enough to trigger a new signature generation when we set $\theta = 6$.

From Table 3, we also notice that θ does not have a significant impact on the clustering quality since F-Measure values are always close or equal to 1. This means that for each worm w, there exists a node C in the signature tree in which $F(w, C)$ is close or equal to 1. In other words, almost all samples from w can be successfully added into the subtree rooted at node C.

Signature quality. After the signature tree has been generated, we employed Algorithm 2 to select a set of signatures to achieve a high detection rate against all tested worm samples and a low false positive rate to the benign traffic. From the signature tree shown in Fig. 8, five signatures (nodes C6, C4, C8, C5, and C10) were selected when the upper limit of false positive (ρ) in Algorithm 2 was set to 0.01. The quality of these signatures in terms of False Positive (FP) and False Negative (FN) on the evaluation benign traffic can be seen in columns 4 and 5 of Table 4.

Table 5 gives the signature comparisons between our approach and previous approaches with respect to the Code Red II worm. As we mentioned in Section 1, distance restrictions and one-byte invariant parts are valuable information for worm detection. Our approach

TABLE 3
Clustering Quality, Runtime, and Memory Usage for Different θ Value

θ	2	3	4	5	6	7	8	9
F-Measure	0.95	0.97	0.99	1	1	1	1	1
Number of nodes	122	93	45	23	22	22	22	22
Number of MSSig	0	3	14	14	14	14	14	14
Runtime (minutes)	68	78	52	72	80	95	125	135
Memory usage (MB)	110	97	84	72	67	68	70	75

TABLE 4
Selected Signatures from the Signature Tree to Detect Five Worms and Nine Worm Variants

Selected Node	Signature	Detecting which worms or worm variants	Evaluation FN	Evaluation FP
C6	*'\xFF\xBF'[200]'\x00\x00\xFA'[2]	BIND-TSIG	0	0
C4	'GET /'* '\xFF\xBF'*'HTTP/1.1\r\n'	ATPhttpd, ATPhttpd.A, ATPhttpd.B, ATPhttpd.C	0	0
C8	'GET '[10]' HTTP/1.1\r\n'* ': '* '\r\nHost: '* '\r\n'* ': '* '\r\nHost: '* '\xFF\xBF'[10]'\r\n'	Apache-Knacker	0	0
C5	'GET /'*'.ida?'*'XX'*'%u'[4]'%u780'* '='[1]'HTTP/1.0\r\n'	CodeRed.A, CodeRed.B, CodeRed.C, CodeRed.D, CodeRed.E, CodeRed.F	0	0
C10	'GET http://'* ': '* '\r\n'* 'null.printer?'* ' HTTP/1.0\r\n'	IISprinter	0	0

can generate a more accurate signature compared with previous approaches because these two types of information can be extracted and expressed in the SRE signature.

Performance study. We used 320 samples without noise to test the PolyTree performance. Table 3 shows the total runtime and memory consumption when we generate a signature tree to accommodate the given 320 worm samples under different θ values. Fig. 9 shows the runtime to add a new sample into the signature tree till 320 samples are added. The runtime to update the tree increases when there are more samples contained in the signature tree. Experimental results show that the average runtime to add a new sample is about 15 seconds in the tree dynamic update process.

7.4 Multiple Polymorphic Worms with Noise

In the first experiment, we used the same 320 samples as in the previous experiment but with injected various noise flows. We set $\theta = 6$. The noise flows were taken from the first week's data of the 1999 DARPA Intrusion Detection Evaluation Data Sets [32]. We find that PolyTree can generate "well-constructed" signature tree and output high-quality signatures even under a 100-percent noise rate (equal number of noise and worm samples). As shown in Table 6, PolyTree exhibits tight clustering of the same type worm samples in the presence of different noise ratios. Although injecting noise flows greatly increases the complexity of the signature tree structure (contains 121 nodes under a 100-percent noise ratio), the constructed signature tree can successfully contain all $MSSig$s of the testing worms and worm variants. Furthermore, PolyTree can still select the same five accurate signatures (under $\rho = 0.01$) as shown in Table 4 with different noise ratios, which is achieved by the function implemented in the signature selector component. This is because according to Theorems 2 and 3 in Appendix A, the majority samples from the same worm are always clustered into the same node during the signature tree construction, and hence, the $MSSig$ of each worm can be generated. The huge number of noise signatures derived from the noise samples will not be selected due to their high false positives.

TABLE 5
Comparison of Signature Accuracy for Different Approaches by Testing Polymorphic Code Red II Worm

Signature Type (Approach)	Generated Signature	Limitation in Precision
Single substring (earlier systems [4], [5], [6])	'.ida?' or '%u780'	lost of most invariant parts and the distance restrictions of invariant parts
Token-subsequence (Polygraph)	GET /.*.ida?.*XX.*%u.*%u780 .*HTTP /1.0\r\n	lost of "=" and the distance restrictions of invariant parts
Multi-sets of tokens (Hamsa)	{'.ida?': 1, '%u780': 1,'HTTP/1.0 \r\n':1,'GET/': 1,'%u': 2}	lost of "=" and the distance restrictions of invariant parts
SRE signature (Ours)	'GET /'*'.ida?'*'XX'*'%u'[4]'%u 780' * '='[1]'HTTP/1.0\r\n'	N/A

In the second experiment, we tested PolyTree by creating randomly generated strings as noise flows. The generated signature tree displays the worm traffic clustering property as well. The selected signature set is the same to the result as in Table 4, which contains five signatures coving all 14 worms and worm variants.

8 Limitation

Signature type. Essentially, there are two signature types for worm/attack detection—*exploit-based* and *vulnerability-based* signatures [19]. An exploit-based signature describes the characteristics of one or a few exploit(s). A vulnerability-based signature describes properties of a certain vulnerability and can detect all possible exploits utilizing this vulnerability. As Fig. 5 implies, our approach can only generate exploit-based signatures.

Runtime. Appendix B gives the runtime analysis of PolyTree. Given a suspicious flow set $\mathcal{M}$, where flows are delivered to PolyTree one by one for signature tree generation, the runtime of each update is $O(|\mathcal{M}_d|^\theta)$ at most and the overall runtime will not exceed $O(|\mathcal{M}||\mathcal{M}_d|^\theta)$, where $\mathcal{M}_d$ is the *maximal dissimilar subset* of $\mathcal{M}$.[2] PolyTree may suffer from the overhead of signature tree update when the number of worm samples is very large. Yet, there are two ways to reduce such an overhead. First, we can only refine PolyTree whenever necessary, e.g., ignoring signature refinement and the new signature creation (sentences 3.1 and 3.2 in Algorithm 1) if a new sample can be detected by a signature in the selected signature set S, which has been chosen by the signature selector in PolyTree. Even if we add the new sample to a node for refining the signature tree, it will not affect the selected signature set S to IDSs for worm traffic detection. Second, we can reduce the overall runtime by taking a small θ value and using a well-designed MSA algorithm. In this paper, we do not focus on the detailed MSA algorithm design. In addition, there is still room to reduce the PolyTree update time, e.g., adopting the fast pairwise sequence alignment technique in [34].

Attack resilience. Since MSA might not get the global optimal, adversaries may adopt some sophisticated attacks to mislead PolyTree. As a consequence, PolyTree may not be reliable for future sophisticated worms. But in practice, PolyTree can provide enough attack resilience for current worms since most of them do not adopt sophisticated polymorphic techniques or automatic signature generation (ASG) attacking techniques. Appendix C discusses the attack resilience of PolyTree for some state-of-the-art attacks

2. $\mathcal{M}_d$ is the maximal subset of $\mathcal{M}$ such that the signatures of any θ samples in $\mathcal{M}_d$ are not specific than the signature of $\mathcal{M}_d$. See Appendix B for more details.

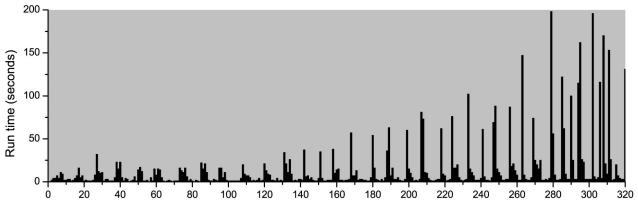

Fig. 9. The signature tree update time for each newly arrived sample ($\theta = 6$).

and the experiment results show that PolyTree is resilient to some attacks because the signature tree can be dynamically updated over time and signatures can be timely refined. Venkataraman et. al. generalized most attacks and proved lower bounds on the number of mistakes that any pattern extraction NSG algorithm must make under a certain assumption [35]. Our approach is a pattern extraction algorithm and subject to this fundamental limitation as well. An example to illustrate this limitation is given in Appendix C.

9 Conclusion

In this paper, we describe a new NSG system—PolyTree, to defend against polymorphic worms. PolyTree is able to incrementally cluster worm samples, generate and refine signatures for polymorphic worms by multiple sequence alignment, incrementally construct a signature tree, and select a set of accurate signatures to be used by current IDSs. In contrast to the isolated view of generated signatures in previous approaches, generating signature tree and revealing inherent relationship for multiple polymorphic worms are the main novelty of this paper. However, the preliminary work leaves large open space for further exploration, like in-depth resilience analysis and countermeasures against attacks, PolyTree update speed improvement and its online deployment.

Appendix A

Correctness of the Signature Tree Generation Algorithm

In this section, unless otherwise stated, a signature tree refers to a fixed one that has been incrementally updated by Algorithm 1.

A.1 Correctness of Signature Tree

Lemma 1. *$\triangleleft$ is a transitive relation.*

Proof. Suppose that $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ are three SRE signatures and $\mathcal{X} \triangleleft \mathcal{Y}$, $\mathcal{Y} \triangleleft \mathcal{Z}$. According to Definition 2, $L(\mathcal{X}) \subseteq L(\mathcal{Y})$ and $L(\mathcal{Y}) \subseteq L(\mathcal{Z})$, which implies that $L(\mathcal{X}) \subseteq L(\mathcal{Z})$. Thus, $\mathcal{X} \triangleleft \mathcal{Z}$. □

Lemma 2. *$\prec$ is an irreflexive and transitive relation.*

Proof. $\prec$ is an irreflexive relation since for any SRE signature $\mathcal{X}$, $|\mathcal{X}| < |\mathcal{X}|$ does not hold. Suppose that $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ are three SRE signatures and $\mathcal{X} \prec \mathcal{Y}$, $\mathcal{Y} \prec \mathcal{Z}$. Obviously, we have $\mathcal{X} \triangleleft \mathcal{Y}$, $\mathcal{Y} \triangleleft \mathcal{Z}$, and $|\mathcal{X}| > |\mathcal{Y}|$, $|\mathcal{Y}| > |\mathcal{Z}|$. From Lemma 1, $\mathcal{X} \triangleleft \mathcal{Z}$ and $|\mathcal{X}| > |\mathcal{Z}|$ imply that $\mathcal{X} \prec \mathcal{Z}$. The transitive property holds. □

TABLE 6
Clustering Quality for Multiple Polymorphic Worms

noise ratio	0%	20%	40%	60%	80%	100%
F-Measure	1	1	1	0.998	0.991	0.982
Number of nodes	22	27	40	69	92	121
Number of MSSig	14	14	14	14	14	14

Lemma 3. *$\triangleleft$ and $\prec$ are transitive for each other. That is, suppose $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ are three SRE signatures, if $\mathcal{X} \prec \mathcal{Y}$ and $\mathcal{Y} \triangleleft \mathcal{Z}$, or $\mathcal{X} \triangleleft \mathcal{Y}$ and $\mathcal{Y} \prec \mathcal{Z}$, then $\mathcal{X} \prec \mathcal{Z}$.*

Proof. The conclusion can be obtained immediately from Definition 3 and Lemmas 1 and 2. □

$Align(.)$ and $MAlign(.)$ are sequence alignment algorithms proposed in [30]. According to the pairwise alignment algorithm, all substrings in $Align(\mathcal{X}, \mathcal{Y})$ must be present in $\mathcal{X}$ with the same order, and the qualifiers between these substrings are progressively calculated during the alignment process to preserve the distance restriction. Given an arbitrary sequence f that can match $\mathcal{X}$, it must match the signature $Align(\mathcal{X}, \mathcal{Y})$. Hence, $\mathcal{X} \triangleleft Align(\mathcal{X}, \mathcal{Y})$. For each $\mathcal{Z} \in \mathcal{S}$, we can easily infer that $\mathcal{Z} \triangleleft MAlign(\mathcal{S})$, a set of signatures, because the $MAlign(.)$ algorithm is constructed by progressively employing the pairwise alignment algorithm. Thus, we have the following lemma:

Lemma 4. *If $\mathcal{X}$ and $\mathcal{Y}$ are two signatures, $\mathcal{S}$ is a set of signatures, then $\mathcal{X} \triangleleft Align(\mathcal{X}, \mathcal{Y})$, and for each $\mathcal{Z} \in \mathcal{S}$, we have $\mathcal{Z} \triangleleft MAlign(\mathcal{S})$.*

Lemma 5. *In a signature tree, if node C_i is the ancestor of node C_j, then $C_j.sig \prec C_i.sig$.*

Proof. We first prove that if node C_i is the parent of node C_j, then $C_j.sig \prec C_i.sig$. There are three cases in which C_j becomes a child of C_i: 1) C_j is split from C_i (sentence 3.1); 2) C_j is moved to C_i as a child (sentence 3.2); and 3) C_j is created as the parent of C_j and C'_j (sentence 3.3). In all cases, we have $C_j.sig \prec C_i.sig$.

If C_i is the ancestor of C_j, there must exist one path from C_i to C_j. Following above the same reasoning process recursively, we have $C_j.sig \prec C_i.sig$ from Lemma 3. □

Lemma 6. *In the signature tree, if a sample f is contained in node C, then $f \triangleleft C.sig$.*

Proof. There are three cases where f is added or moved to C. In each case, the lemma holds. 1) f is added to C (in sentence 2). Obviously, $f \triangleleft C.sig$. 2) C, containing f, is split from another node C_i (sentence 3.1). From Lemma 4, we have $f \triangleleft MAlign(C) = C.sig$. 3) f is moved to C in the *SinkSamples* operation. $f \triangleleft C.sig$ still holds. □

Theorem 1 (Correctness of signature tree). *The incremental signature tree generation algorithm will produce correct signature trees satisfying Definition 6.*

Proof. We can derive the theorem immediately from Lemmas 5 and 6. □

A.2 Properties of Signature Tree

Lemma 7. *Suppose that C_i and C_j are two nodes in the signature tree. If $C_i.sig \prec C_j.sig$, then $C_i.level > C_j.level$.*

Proof. We prove it by contradiction. Suppose that $C_i.level < C_j.level$. There must exist a node C_k who is the ancestor of C_j and $C_k.level = C_i.level$. From Lemma 5, we have $C_j.sig \prec C_k.sig$. Because $C_i.sig \prec C_j.sig$, we have $C_i.sig \prec C_k.sig$. This contradicts the sentence 3.2. Similarly, that $C_i.level = C_j.level$ will also lead to a contradiction. □

Lemma 8. *Given a polymorphic worm w, if C is a node in the signature tree such that $w \lessdot C.sig$, then there must not exist f as a sample of w and C' as a node in the signature tree such that $f \in C'$ and $C'.level < C.level$.*

Proof. We prove it by contradiction. Assume that there exist a worm sample f and a node C' in the signature tree such that $f \in C'$ and $C'.level < C.level$. We know that f must be added to C' first before node C is created. Otherwise, f must be added to C instead of C' because $f \triangleleft C.sig$ and $C.level > C'.level$. Hence, there are only the following two cases:

1. $C'.level = C.level - 1$. Obviously, this does not hold because when node C is added into the level $C.level$, f should be "sinked" to C from C' by the *SinkSamples* operation.
2. $C'.level < C.level - 1$. There must be a node C_j as the ancestor of C such that $C.sig \triangleleft C_j.sig$ and $C_j.level = C'.level - 1$. It is easy to know that $w \lessdot C_j.sig$ for $C.sig \triangleleft C_j.sig$. Thus, C' and C_j satisfy the conditions in case 1. It means that this case does not hold either. □

Lemma 7 shows an important property of the signature tree that the more specific signature, the higher level it should be contained in. Lemma 8 states that the captured worm samples in the signature tree tend to "sink" to nodes in a higher level. In other words, samples can be clustered in a node with the highest level, where nodes represent the most specific signatures.

Lemma 9. *Given a polymorphic worm w and its two samples f_1 and f_2, there must not exist two nodes C_1 and C_2 in the signature tree such that $C_1.level = C_2.level$, $w \lessdot C_1.sig$, $w \lessdot C_2.sig$, $f_1 \in C_1$ and $f_2 \in C_2$.*

Proof. We prove it by contradiction. Assume that there exist two nodes C_1 and C_2 satisfying the properties, i.e., $C_1.level = C_2.level = k$, $w \lessdot C_1.sig$, $w \lessdot C_2.sig$, $f_1 \in C_1$ and $f_2 \in C_2$. Without loss of generality, assume that C_1 is added (or moved) to level k earlier than C_2. Only the following two cases could be possible:

1. When f_2 is added to C_2, C_1 has been in level k. In this case, C_2 can be in level k or not. However, both locations of C_2 will lead to contradictions. If C_2 is in level k, since C_1 is added to level k earlier than C_2, obviously, f_2 should not be added to C_2 according to sentence 1 in Algorithm 1. If C_2 has not been added in level k, C_2 must be in a level less than k since a node can only sink during the incremental signature tree generation. Thus, f_2 must not be added to C_2 neither.
2. When f_2 is added to C_2, C_1 has not been in level k. Since at this moment C_2 is not in level k, $C_2.level$ must be less than k. After C_1 is added (or moved) to level k, f_2 should remain to be in C_2, which contradicts Lemma 8. □

Given a polymorphic worm w, Lemma 9 implies that there is only one node in the highest level k and labeled with one signature of w that contains the sample(s) of w. Moreover, if there are many nodes in level k that are labeled with a signature of w, the only node containing sample(s) of w is the one added to level k first.

Theorem 2 (Clustering quality guarantee). *If samples captured by PolyTree are not maliciously injected and some of them are from a polymorphic worm w, then the majority samples of w must be contained in one node in the signature tree.*

Proof. That the samples captured by PolyTree are not maliciously injected implies that the majority samples of w in the signature tree should be contained in some nodes labeled with a signature of w. Lemma 8 shows that all samples of w in the signature tree will "sink" to the nodes in the highest level k. Lemma 9 further states that, in fact, the samples of w at one level should be clustered into one node who is the first to be added at level k. □

Assumption 1. *The function $MAlign()$ used in the signature tree generation algorithm provides a solution to Problem 1. That is, if F is a set containing enough samples of a polymorphic worm w, then $MAlign(F) = MSSig(w)$.*

From the discussion in Section 4.3, we believe that this assumption is practical and can be achieved in most cases if we can collect enough number of samples from a specific worm.

Theorem 3 (Most specific signature generation guarantee). *If enough samples of a polymorphic worm w have been added to the signature tree, then the signature tree must contain a node C such that $C.sig = MSSig(w)$.*

Proof. According to Theorem 2, the majority samples of w will be grouped into one node. Let the node be C. From Assumption 1, we know that given enough samples captured by PolyTree, there should be θ (a majority number) samples grouped in C and the $MSSig$ of w will be generated in Algorithm 1. □

APPENDIX B
RUNTIME ANALYSIS

Runtime complexity of each signature tree update. Suppose that a sample to be inserted into a signature tree does not exceed l bytes. If the sample is added to a node that contains k samples, the complexity to update the signature tree is $O(C_k^\theta \theta l^2)$, where $O(\theta l^2)$ is the complexity of the $MAlign$ operation on a subset with θ samples and C_k^θ is the complexity to examine all possible subsets in the node. Since θ and l are constant, the runtime complexity of the signature tree update is $O(C_k^\theta)$ upon a new sample arrival.

Bounded runtime of each signature tree update. Suppose that C is a sample set and its signature is $sig(C)$, which means that for each sample $s \in C$, $s \triangleleft sig(C)$ holds. We say C is a *dissimilar set* when the signature of any θ samples in C is no more specific than $sig(C)$. For example, given $\theta = 2$, {"Aaaaa,""Abbb,""Accc"} is a dissimilar set, since the

signature of this set is "'A'*" and no signature of any two samples is more specific than "'A'*".

Given a signature tree containing all samples in a set $\mathcal{M}$, if a subset $\mathcal{M}_d$ is a dissimilar set that has the maximal number of samples, we call $\mathcal{M}_d$ the *maximal dissimilar subset* of $\mathcal{M}$. Since the sample size of any node in the signature tree will not exceed $|\mathcal{M}_d|$, the runtime of each signature tree update will not exceed $O(C^{\theta}_{|\mathcal{M}_d|}) = O(|\mathcal{M}_d|^{\theta})$. Fig. 9 illustrates the runtime of each update bounded by the size of the maximal dissimilar subset of a suspicious traffic pool.

Overall runtime under the online deployment. Based on the bounded runtime of each signature tree update, the overall runtime of PolyTree under the online deployment will not exceed $O(\sum_{k=1}^{|\mathcal{M}|} C^{\theta}_{|\mathcal{M}_d|}) = O(|\mathcal{M}||\mathcal{M}_d|^{\theta})$ (mainly for the signature tree update), where $\mathcal{M}_d$ is the maximal dissimilar subset of $\mathcal{M}$. In the worst-case scenario where the maximal dissimilar subset of $\mathcal{M}$ is $\mathcal{M}$ itself (ultimately, all samples of $\mathcal{M}$ will be added into a single node), the overall runtime is $O(|\mathcal{M}|^{\theta+1})$. In this scenario, the update performance will degrade significantly when $|\mathcal{M}|$ increases. Fortunately, the above worst-case scenario is not practical, since the captured real samples should exhibit great diversity and can be clustered into many nodes in the signature tree due to various network protocols and worm variants. Thus, $\mathcal{M}_d$ is the sample set of the most popular worm and its size should be much smaller than $|\mathcal{M}|$ in most cases.

Overall runtime comparison with Polygraph and Hamsa under the online deployment. Under the online deployment, the overall runtime of Polygraph is

$$O\left(\sum_{k=1}^{|\mathcal{M}|} k^2|\mathcal{N}|\right) = O\left(\frac{|\mathcal{M}|(|\mathcal{M}|+1)(2|\mathcal{M}|+1)}{6}\cdot|\mathcal{N}|\right) = O(|\mathcal{M}|^3|\mathcal{N}|),$$

where $\mathcal{N}$ is a benign traffic pool for the false positive testing. The overall runtime of Hamsa is $O(\sum_{k=1}^{|\mathcal{M}|} T_k|\mathcal{N}|)$, where T_k is the set of tokens for the first k samples. According to experiments of Hamsa [7], $T_k > k$. The overall runtime of Hamsa should be approximatively

$$O\left(\sum_{k=1}^{|\mathcal{M}|} k|\mathcal{N}|\right) = O(|\mathcal{M}|^2|\mathcal{N}|).$$

The asymptotic overall runtime for PolyTree, Polygraph, and Hamsa is $O(|\mathcal{M}_d|^{\theta})$, $O(|\mathcal{M}|^2|\mathcal{N}|)$, and $O(|\mathcal{M}||\mathcal{N}|)$, respectively.

It is sound to assume $|\mathcal{M}_d| \ll |\mathcal{M}| \ll |\mathcal{N}|$ in practice. As a result, PolyTree is possible to be faster than Polygraph and Hamsa, if $\mathcal{M}$ contains a great diversity of worms and θ can take a small value in a well-designed MSA algorithm.

APPENDIX C
ATTACK ANALYSIS

Red herring attack [10]. This attack is being considered as the biggest challenge to current NSG systems, which encompasses Randomized Red Herring attack and Dropped Red Herring attack. In the Randomized Red Herring attack, the attacker chooses a set of spurious invariant parts and constructs the target class samples such that each one, in a probability, contains a particular spurious invariant part. In the Dropped Red Herring attack, the attacker initially inserts a set of spurious invariant parts into worm samples. When time elapses, the attacker gradually drops spurious invariant parts one by one. Reported by Newsome et al. [10], this attack causes Polygraph [3] to generate signatures that are too specific, and thus, results in an unacceptable false negative rate. Similarly, Hamsa [7] is also vulnerable to this attack. We did an experiment to evaluate our approach for the Red herring attack. We chose the set of five spurious invariant parts as {"11111," "22222," "33333," "44444," "55555"}, and modified the BIND-TSIG worm by inserting these spurious invariant parts.

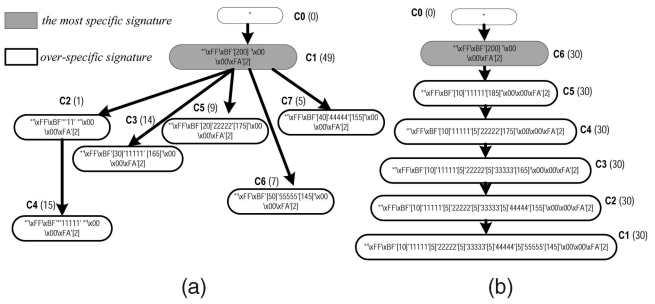

Fig. 10. The generated signature tree for BIND-TSIG worm under Red Herring Attacks. (a) Randomized Red Herring attack. (b) Dropped Red Herring attack.

For the Randomized Red Herring attack, we created 100 samples such that each one is with a probability 0.2 to contain a particular spurious invariant part, and then submitted these samples to PolyTree for signature generation in a random order. The generated signature is shown in Fig. 10a. We observe that the most specific signature of the BIND-TSIG worm is created earliest (C1 is the first created node), the overspecific signatures, like "'*'\xFF\xBF'[30]'11111'[165]'\x00\x00\xFA'[2]," were indeed created later. However, our signature selection algorithm will ensure no selection of this overspecific signature for detection because it is a child node of C1. This result illustrates that our approach is invulnerable to the Randomized Red Herring attack.

For the Dropped Red Herring attack, we first created a group of 30 samples that each one included all five spurious invariant parts. Then one spurious invariant part was dropped to create another group of 30 samples until the last 30 samples contained no spurious invariant parts. Totally, 180 samples in six groups were sequentially delivered to PolyTree group by group. The result is shown in Fig. 10b. We find that PolyTree is able to quickly generate a refined signature once there are θ up-to-date samples (in which a spurious invariant part was just dropped) available, and finally, six signatures are sequentially generated with the same order of the appearance of the six worm sample groups. According to our signature selection method, a refined (more general) signature will replace the old overspecific signature for worm detection. However, previous methods, like Hamsa, are much likely to fail to generate a refined signature because these θ samples could remain to be a small portion in the whole suspicious flow pool.

An adversary could carry out an "extreme Dropped Red Herring attack," where each group contains only θ samples. That is, a worm will drop one spurious invariant part after every last θ sample is released. Under this attack, PolyTree will make a number of FP mistakes due to the fundamental limitation of the Learning-based approach [35]. Fortunately, this attack is very difficult to implement since hosts infected by worms are distributed throughout the world and worms are difficult to synchronize their actions.

Coincidental-pattern attack [3]. Rather than filling in wildcard bytes with random characters, the attack only selects wildcard characters from a small range. For example, if the attacker fills each of the variant bytes in the worm with only binary bit "0" or "1," a polymorphic worm can be like "GET01001011HTTP1101011...." As a result, different worm samples share a large number of tokens consisting of "0" and "1," like "01," "11," "011," etc.

Through experiments, we find that filling in wildcard bytes from a small range (less than 5 characters) results in a poor clustering and inaccurate signature tree generation in PolyTree. In contrast, the use of a large range (more than 12) will have almost no influence on our approach. Fortunately, this kind of attacks adopting a small range of wildcard characters can be easily detected by a number of anomaly detection systems such as [36], [23], since the distribution of characters in worm payload is very anomaly. As a result, the attack is less likely to target at PolyTree by taking the risk of being detected by most anomaly detection systems.

Token-fit attack [7]. In this attack, a number of tokens extracted from benign traffic are intentionally encoded into the variant part of each worm sample. In order to evaluate the performance of PolyTree against this attack, we used 320 worm samples in the presence of 40 percent noise samples. This is similar to the experiment in Section 7.4, but for every worm sample, we injected another five tokens randomly selected from a set of 100 tokens. We conducted experiments three times. The results were similar to the experimental results in the Randomized Red Herring attack test that there was a slight decrease in the clustering quality (F-Measure) of the signature tree but PolyTree nonetheless was able to generate correct signatures.

Fake anomalous flows attack [21]. In this attack, a worm uses intentionally constructed fake suspicious flows, which contain fake invariants, to pollute the suspicious flow pool, and thus, mislead the signature generation system. To delude PolyTree, an attacker may inject some fake worm samples among real ones for each worm, hoping that PolyTree would classify the genuine and fake worm samples together into one cluster, thereby generating a wrong signature.

As has been previously observed [21], [7], if an accurate and robust flow classifier is in place, the attack can be circumvented by PolyTree. Yet PolyTree will not be influenced by the attack even without such a flow classifier when the number of fake samples for a worm is less than θ. In addition, our approach is resilient to this attack because an attacker cannot manage the samples encountered by PolyTree when injecting suspicious flows.

Acknowledgments

The authors would like to thank Newsome of Carnegie Mellon University for helping them to generate worm samples. This work was partly supported by China 973 under Grant nos. 2009AA01A403, and 2009CB320503; China 863 under Grant no. 2008AA01A325; China NSFC under Grant no. 60803161, and 61003303; and HK RGC PolyU 5311/06E.

References

[1] Y. Song, M.E. Locasto, A. Stavrou, A.D. Keromytis, and S.J. Stolfo, "On the Infeasibility of Modeling Polymorphic Shellcode," *Proc. ACM Conf. Computer and Comm. Security (CCS)*, 2007.

[2] J.R. Crandall, S.F. Wu, and F.T. Chong, "Experiences Using Minos as a Tool for Capturing and Analyzing Novel Worms for Unknown Vulnerabilities," *Proc. GI SIG SIDAR Conf. Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2005.

[3] J. Newsome, B. Karp, and D. Song, "Polygraph: Automatically Generating Signatures for Polymorphic Worms," *Proc. 2005 IEEE Symp. Security and Privacy*, pp. 226-241, 2005.

[4] C. Kreibich and J. Crowcroft, "Honeycomb—Creating Intrusion Detection Signatures Using Honeypots," *Proc. Second Workshop Hot Topics in Networks (Hotnets II)*, 2003.

[5] H.A. Kim and B. Karp, "Autograph: Toward Automated, Distributed Worm Signature Detection," *Proc. USENIX Security Symp.*, pp. 271-286, 2004.

[6] S. Singh, C. Estan, G. Varghese, and S. Savage, "Automated Worm Fingerprinting," *Proc. Sixth USENIX Symp. Operating Systems Design and Implementation (OSDI)*, 2004.

[7] Z. Li, M. Sanghi, Y. Chen, M.Y. Kao, and B. Chavez, "Hamsa: Fast Signature Generation for Zero-Day Polymorphic Worms with Provable Attack Resilience," *Proc. 2006 IEEE Symp. Security and Privacy*, 2006.

[8] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection," *Proc. ACM SIGCOMM*, vol. 36, pp. 339-350, 2006.

[9] K. Wang, G. Cretu, and S.J. Stolfo, "Anomalous Payload-Based Worm Detection and Signature Generation," *Proc. Int'l Symp. Recent Advances in Intrusion Detection (RAID)*, 2003.

[10] J. Newsome, B. Karp, and D. Song, "Paragraph: Thwarting Signature Learning by Training Maliciously," *Proc. Int'l Symp. Recent Advances in Intrusion Detection (RAID)*, pp. 81-105, 2006.

[11] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," *Proc. 12th Ann. Network and Distributed System Security Symp.*, 2005.

[12] J.R. Crandall and F.T. Chong, "Minos: Control Data Attack Prevention Orthogonal to Memory Model," *Proc. 37th Ann. IEEE/ACM Int'l Symp. Microarchitecture*, pp. 221-232, 2004.

[13] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-End Containment of Internet Worms," *Proc. ACM Symp. Operating Systems Principles*, pp. 133-147, 2005.

[14] J.R. Crandall, Z. Su, S.F. Wu, and F.T. Chong, "On Deriving Unknown Vulnerabilities from Zero-Day Polymorphic and Metamorphic Worm Exploits," *Proc. 12th ACM Conf. Computer and Comm. Security*, pp. 235-248, 2005.

[15] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt, "Automatic Diagnosis and Response to Memory Corruption Vulnerabilities," *Proc. 12th ACM Conf. Computer and Comm. Security*, pp. 223-234, 2005.

[16] Z. Liang and R. Sekar, "Automatic Generation of Buffer Overflow Attack Signatures: An Approach Based on Program Behavior Models," *Proc. 21st Ann. Computer Security Applications Conf.*, pp. 215-224, 2005.

[17] Z. Liang and R. Sekar, "Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers," *Proc. 12th ACM Conf. Computer and Comm. Security*, pp. 213-222, 2005.

[18] M.E. Locasto, K. Wang, D. Angelos, and J. Salvatore, "Flips: Hybrid Adaptive Intrusion Prevention," *Proc. Eighth Int'l Symp. Recent Advances in Intrusion Detection*, pp. 82-101, 2005.

[19] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Towards Automatic Generation of Vulnerability-Based Signatures," *Proc. 2006 IEEE Symp. Security and Privacy*, pp. 2-16, 2006.

[20] X.F. Wang, Z. Li, J. Xu, M.K. Reiter, C. Kil, and J.Y. Choi, "Packet Vaccine: Black-Box Exploit Detection and Signature Generation," *Proc. 13th ACM Conf. Computer and Comm. Security*, pp. 37-46, 2006.

[21] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif, "Misleading Worm Signature Generators Using Deliberate Noise Injection," *Proc. IEEE Symp. Security and Privacy,* 2006.
[22] Z. Li, L. Wang, Y. Chen, and Z. Fu, "Network-Based and Attack-Resilient Length Signature Generation for Zero-Day Polymorphic Worms," *Proc. 15th IEEE Int'l Conf. Network Protocols (ICNP '07),* 2007.
[23] K. Wang and J. Salvatore, "Anomalous Payload-Based Network Intrusion Detection," *Proc. Int'l Symp. Recent Advances in Intrusion Detection (RAID),* pp. 203-222, 2004.
[24] R. Vargiya and P. Chan, "Boundary Detection in Tokenizing Network Application Payload for Anomaly Detection," *Proc. ICDM Workshop Data Mining for Computer Security (DMSEC),* 2003.
[25] Y. Tang and S. Chen, "Defending against Internet Worms: A Signature-Based Approach," *Proc. IEEE INFOCOM,* 2005.
[26] V. Yegneswaran, P. Barford, and D. Plonka, "On the Design and Use of Internet Sinks for Network Abuse Monitoring," *Proc. Int'l Symp. Recent Advances in Intrusion Detection (RAID),* pp. 146-165, 2004.
[27] M. Bailey, E. Cooke, F. Jahanian, and J. Nazario, "The Internet Motion Sensor—a Distributed Blackhole Monitoring System," *Proc. Network and Distributed System Security Symp. (NDSS),* 2005.
[28] Y. Tang, H.P. Hu, X.C. Lu, and J. Wang, "Honids: Enhancing Honeypot System with Intrusion Detection Models," *Proc. Fourth IEEE Int'l Workshop Information Assurance (IWIA '06),* pp. 135-143, 2006.
[29] R. Sommer and V. Paxson, "Enhancing Byte-Level Network Intrusion Detection Signatures with Context," *Proc. 10th ACM Conf. Computer and Comm. Security (CCS '03),* pp. 262-271, 2003.
[30] Y. Tang, X. Lu, and B. Xiao, "Generating Simplified Regular Expression Signatures for Polymorphic Worms," *Proc. Fourth Int'l Conf. Autonomic and Trusted Computing (ATC '07),* 2007.
[31] Y. Tang, B. Xiao, and X. Lu, "Using a Bioinformatics Approach to Generate Accurate Exploit-Based Signatures for Polymorphic Worms," *Computers & Security,* vol. 28, pp. 827-842, 2009.
[32] R. Lippmann, J.W. Haines, D.J. Fried, J. Korba, and K. Das, "The 1999 DARPA Off-Line Intrusion Detection Evaluation," *Computer Networks,* vol. 34, no. 4, pp. 579-595, 2000.
[33] M. Steinbach, G. Karypis, and V. Kumar, "A Comparison of Document Clustering Techniques," *Proc. KDD Workshop Text Mining,* 2000.
[34] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney, "High-Throughput Sequence Alignment Using Graphics Processing Units," *BMC Bioinformatics,* vol. 8, no. 1, 2007.
[35] S. Venkataraman, A. Blum, and D. Song, "Limits of Learning-Based Signature Generation with Adversaries," *Proc. Network and Distributed System Security Symp. (NDSS),* 2008.
[36] C. Kruegel, T. Toth, and E. Kirda, "Service Specific Anomaly Detection for Network Intrusion Detection," *Proc. ACM Symp. Applied Computing (SAC '02),* pp. 201-208, 2002.

Yong Tang received the BSc, MSc, and PhD degrees in computer science from the College of Computer, National University of Defense Technology, China, in 1998, 2002, and 2008, respectively. He is currently a lecturer in the College of Computer, National University of Defense Technology, China. His research interests include network security, computer networks, data mining, and machine learning.

Bin Xiao received the BSc and MSc degrees in electronics engineering from Fudan University, China, in 1997 and 2000, respectively, and the PhD degree in computer science from the University of Texas at Dallas in 2003. He is currently an associate professor in the Department of Computing at the Hong Kong Polytechnic University. His research interests include distributed computing systems, data storage, and security in wired and wireless computer networks. He is a member of the IEEE and the IEEE Computer Society.

Xicheng Lu received the BSc degree in computer science from Harbin Military Engineering Institute, China, in 1970. He is currently a professor in the College of Computer, National University of Defense Technology, Changsha, China. He was a visiting scholar at the University of Massachusetts between 1982 and 1984. His research interests include distributed computing, computer networks, parallel computing, network security, etc. He has served as a member of editorial boards of several journals and cochaired many professional conferences. He is the joint recipient of more than a dozen academic awards, including four First Class National Scientific and Technological Progress Prizes of China. He is an academician of the Chinese Academy of Engineering.