Misleading Worm Signature Generators Using Deliberate Noise Injection

Roberto Perdisci[⋆,†], David Dagon[†], Wenke Lee[†], Prahlad Fogla[†] and Monirul Sharif[†]

[†]Georgia Institute of Technology, Atlanta, GA 30332, USA
[⋆]University of Cagliari, 09123 Cagliari, ITALY
{rperdisc,dagon,wenke,prahlad,msharif}@cc.gatech.edu

Abstract

Several syntactic-based automatic worm signature generators, e.g., Polygraph, have recently been proposed. These systems typically assume that a set of suspicious flows are provided by a flow classifier, e.g., a honeynet or an intrusion detection system, that often introduces "noise" due to difficulties and imprecision in flow classification. The algorithms for extracting the worm signatures from the flow data are designed to cope with the noise. It has been reported that these systems can handle a fairly high noise level, e.g., 80% for Polygraph. In this paper, we show that if noise is introduced deliberately to mislead a worm signature generator, a much lower noise level, e.g., 50% or below, can already prevent the system from reliably generating useful worm signatures. Using Polygraph as a case study, we describe a new and general class of attacks whereby a worm can combine polymorphism and misleading behavior to intentionally pollute the dataset of suspicious flows during its propagation and successfully mislead the automatic signature generation process. This study suggests that unless an accurate and robust flow classification process is in place, automatic syntactic-based signature generators are vulnerable to such noise injection attacks.

1 Introduction

Signature generation is a key step in the defense against worm propagation. Most of the signatures used by firewalls or signature-based intrusion detection systems (IDS) are created using a manual analysis of worm traffic flows. This is usually a time-consuming process, and thus cannot keep pace with rapidly spreading worms. Manual analysis becomes even harder and more time-consuming if the worms use metamorphism and polymorphism techniques. Automatic signature generation is a promising alternative. The goal is to automatically, and thus very quickly, extract the invariant parts of a worm as its signature. Early approaches [8, 6, 21] are based on syntactic analysis of suspicious traffic flows. These approaches have limited abilities to extract reliable signatures from truly polymorphic worms. Newsome et al. recently proposed two approaches to address this problem [14, 15]. Polygraph [14] is based on syntactic analysis of suspicious traffic flows, and implements three different types of signature generation algorithms. Taint analysis [15] is a semantic analysis approach based on the execution of possible vulnerable applications inside a protected environment.

In this paper, we will focus on signature generation systems that aim at automatically extracting and deploying signatures that could be used by firewalls or signature-based network IDS. Other automatic signature generators are based on the extraction of *host-based* signatures that need to access the execution or application environment they are trying to protect in order to be effective, as proposed for example in [10]. We do not discuss these systems here. We will examine the abilities of syntactic-based automatic signature generators in the face of advanced polymorphic worms that not only spread using a high level of polymorphism but also deliberately *mislead* the signature generation process. Using Polygraph [14] as a case study, we introduce a new and general class of attacks whereby a worm can combine polymorphism and misleading behavior in order to prevent the generation of reliable signatures. We will show that this result can be achieved by intentionally injecting noise into the dataset of suspicious flows used by syntactic-based signature generators to extract the signatures. We will present a specific attack that can mislead Polygraph, and then we will discuss how such noise injection attacks are general in that different attacks can be devised to mislead other recently proposed automatic signature generators.

The system architecture of Polygraph includes a flow classifier module and a signature generator module [14]. The flow classifier collects the suspicious flows that will be used to extract the signatures. The authors assumed that the flow classifier can be imperfect and that it can introduce

some noise into the pool of suspicious flows, regardless of the classification technique used by the flow classifier. The authors then proposed some techniques to cope with the noise during the signature generation process. This design characteristic is common to most of the syntactic-based automatic signature generators. That is, little or no attention is paid to filtering the noise during the suspicious flow gathering process. We believe this is a serious shortcoming that can be exploited by combining polymorphism and misleading behavior. We will show how a *misleading polymorphic worm* can create and send *fake anomalous* flows during its propagation to deliberately pollute the set of flows used to extract the signatures. Polygraph's authors state that their system is resilient to (at least) 80% of noise into the set of suspicious flows [14]. We will show that by constructing well-crafted *fake anomalous* flows, a worm can mislead the signature generation algorithms by injecting much less than 80% of noise into the set of suspicious flows, thus preventing the generation of useful signatures. We would like to emphasize that although we demonstrate the effects of the noise injection attack on Polygraph, which is used as a case study here, it is a general attack on all the syntactic-based signature generation systems proposed in the literature because they do not addresses directly the problem of intentional pollution of the dataset of suspicious flows. In particular, we will discuss how the attack can be generalized to defeat other recent automatic signature generation systems, and why it cannot always be prevented by even *semantic-based* approaches similar to [15].

Our work is structured as follows. In Section 2 we present a technique the worm could apply to pollute the suspicious flow pool. We show the effect of the attack using Polygraph as a case study and then discuss the effects on other signature generators. We present results obtained by simulating the noise injection attack against Polygraph in Section 3, and then we discuss them in Section 4. Section 5 summarizes the related work and then we briefly conclude in section 6.

2 Noise Injection Attack

Noise injection attack works by polluting the set of traffic flows, or *suspicious flow pool* [6, 14], used by automatic signature generators in the signature extraction process. The attack aims to mislead the signature generation algorithms by injecting *well-crafted* noise to prevent the generation of useful signatures. In the following sections we briefly survey the most common techniques used by a "flow classifier" to collect the suspicious flows. We then show how the worm can inject noise without *a priori* knowledge about the classification technique in use. To accomplish the task of misleading the signature generation algorithms, the noise has to be crafted in a suitable manner. Different noise injection attacks can be implemented by crafting the noise in different manners. We first demonstrate how this attack can be implemented against Polygraph [14], and then analyze the possible effects of noise injection attack on Nemean [28], another recently proposed automatic signature generator. Different implementations of the attack can be devised to mislead other signature generators.

2.1 Collecting Suspicious Flows

A few techniques have been proposed to accomplish the task of collecting the suspicious flows. Honeycomb [8] uses a simulated honeynet. Any flow sent towards the honeynet is inserted into the suspicious flow pool. Nemean [28] uses a similar approach combining real and simulated hosts. In [25] a double honeynet is proposed. In this case a first-layer honeynet is made of real hosts. Whenever a first-layer honeypot is infected by a worm, its outgoing traffic is redirected to a second-layer simulated honeynet and inserted into the suspicious flow pool. Autograph [6] implements a classification approach based on port-scanning detection. Each valid flow sent by a scanner to a valid IP address is inserted into the suspicious flow pool. Anomaly-based IDS can also be used as flow classifiers. For example, PAYL [27] uses the byte frequency distribution of the normal packets to detect anomalies, and can be used as a flow classifier.

There are other techniques that are not considered in our study. Earlybird [21] extracts all the possible substrings of a given fixed length β from each packet to compute the content prevalence. β cannot be reduced to just a few bytes due to computational complexity and memory consumption problems. As shown in [14], a polymorphic worm can contain invariants that are just two or three bytes long, potentially evading Earlybird. Since our study focuses on *misleading* polymorphic worms that try to mislead signature generators, we must assume that the flow classifier can detect polymorphic worm instances as suspicious flows. Approaches for run-time detection of injected code, e.g., [15, 10, 5, 1] are not considered because they are largely limited to *application-based* worms (e.g., CodeRed[12], Slammer[11], etc.) and are not effective against *OS-based* worms (e.g., Sasser[17], Zotob[18], etc.). We are concerned with general-purpose worms. More importantly, these approaches are "host-based" while almost all the automatic signature generators presented in literature use "traffic-based" flow classifiers.

2.2 Injecting Noise into The Suspicious Flow Pool

Suppose a worm has infected a host in network A and is now trying to infect some hosts in network B. Suppose also that each time the worm sends a polymorphic instance

to a host in B, it also sends a *fake anomalous* flow to the same host, as shown in Figure 1. Section 2.3.5 provides details on the creation of *fake anomalous* flows. For now consider that the *fake anomalous* flow does not need to exploit the vulnerability and thus can be crafted in a very flexible manner to appear like the real worm in all but the invariant parts (which are necessary to exploit the vulnerability). For example a *fake anomalous* flow can be crafted so that it contains the same protocol framework as the worm (e.g., a GET request) and the same byte frequency distribution, and at the same time not containing the real worm's invariants.

Suppose the network B is monitored by a "traffic-based" flow classifier. The worm and its *fake anomalous* flow must both be stored in the suspicious flow pool in order to mislead the signature generation algorithm. This is possible with the flow classifiers we consider (see Section 2.1). We describe how this can be accomplished with each of the flow classifiers below:

- **Honeynet**. In this case the vulnerable host that the worm is trying to infect can be a real or simulated honeypot. Since both the real worm and the *fake anomalous* flow are sent to the same destination at (roughly) the same time, they will both be considered suspicious by the honeypot and stored into the suspicious flow pool.

- **Double honeynet**. In this case the real worm will infect a first-layer honeypot, whereas the *fake anomalous* flow will not, and will be disregarded. However, only the outgoing traffic will be redirected to the second-layer simulated honeypot and stored into the suspicious flow pool. Given that the outgoing traffic generated by the worm instance at the first-layer honeypot will again contain both a real worm flow and another fake anomalous flow, they will be stored into the suspicious flow pool together.

- **Port-scanning detection**. If the worm scans more than s unused IP addresses, the source of the scanning (i.e., the infected host in A) will be considered a scanner. Therefore, each flow sent by the infected host in A towards B after the scanning phase will be considered suspicious. Given that the real worm and the *fake anomalous* flow originate from the same source host, they will be both inserted into the suspicious flow pool.

- **Byte frequency-based classifier**. The *fake anomalous* flow can be easily crafted to match the byte frequency distribution of the real worm flow (as discussed in Section 2.3.5). This means that if the real worm flow is flagged as anomalous, its *fake anomalous* flow will very likely be flagged as anomalous, too. Thus, both the worm and the *fake anomalous* flow will be stored into the suspicious flow pool.

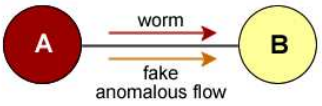

Figure 1: Worm propagation

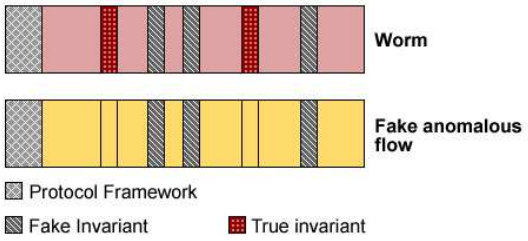

Figure 2: Structure of the fbws (simplifi ed)

Note that each copy of the worm could craft and send more than one *fake anomalous* flow at the same time. In this case the real worm flow and all its *fake anomalous* flows will be inserted into the suspicious flow pool together. The discussion above suggests that without a semantic-based analysis it is not possible to distinguish between the real worm flow and its fake anomalous flows.

2.3 Crafting the Noise: A Case Study Using Polygraph

In this section we present a noise injection attack devised to mislead Polygraph [14]. In order to explain how the noise can be crafted to mislead Polygraph we first describe the high level structure of a polymorphic worm and how Polygraph extracts worm signatures.

2.3.1 High Level Structure of A Polymorphic Worm

As discussed in [7] and in [14], a polymorphic worm is made of the following components:

- **Protocol framework**. In many cases the vulnerability is associated with a particular execution path in the application code. In turn, this execution path can be activated by one (or just a few) particular request type(s). Therefore, the protocol framework is usually common to all the worm variants.

- **Exploit's invariant bytes**. These bytes have a fixed value that cannot be changed because they are absolutely necessary for the exploit to work.

- **Wildcard bytes**. These bytes can assume any value without affecting the exploit.

- **Worm's body**. It contains the instructions the worm executes once the vulnerability has been exploited. If the worm uses a good polymorphic engine, these bytes can assume different values in each worm copy.

- **Polymorphic decryptor**. It contains the first instructions to be executed after the vulnerability has been exploited. The polymorphic decryptor decodes the worm's body and then jumps to it. The polymorphic decryptor itself can change.

Note that this is a simplified view. Depending on the particular vulnerability exploited by the worm, the protocol framework and the exploit's supposedly invariant bytes may assume many different values. This means that there can actually be no invariants, given that each worm instance could use one out of many different values.

2.3.2 Polygraph's Signature Generation Module

Polygraph consists of several modules [14]. A flow classifier performs flow reconstruction and classification on packets received from the network. The flows deemed suspicious are stored into a *suspicious flow* pool, whereas the flows deemed innocuous are stored into an *innocuous flow* pool. The signature generator module uses both pools during the signature generation process. The objective of Polygraph [14] is to extract the invariant parts of a polymorphic worm using three different signature generation algorithms. We briefly summarize how these algorithms work.

- **Conjunction signatures**. During the preprocessing phase the substrings common to all the flows in the suspicious flow pool are extracted. These substrings are called *tokens*. A conjunction signature is made of an unordered set of tokens. A flow matches the signature if it contains all the tokens in the signature.

- **Token-Subsequence signature**. As with the conjunction signatures, the set of tokens in common among all the suspicious flows are extracted. Then, each suspicious flow is rewritten as a sequence of tokens separated by a special character γ. A string alignment algorithm creates an ordered list of tokens that is present in all the suspicious flows. A token-subsequence signature consists of the obtained ordered list of tokens. A flow matches the signature if the ordered sequence of tokens is in the flow.

- **Bayes signatures**. All the tokens of a minimum length α that are common to at least K out of the total number N of suspicious flows are extracted. Then, for each token t_i, $p(t_i|Suspicious\ flow)$ and $p(t_i|Innocuous\ flow)$, the probabilities of finding the token in a suspicious flow and in an innocuous flow, respectively, are computed. A score

$$\lambda_i = log\left[\frac{p(t_i|Suspicious\ flow)}{p(t_i|Innocuous\ flow)}\right]$$

is then assigned to each token t_i. The probability $p(t_i|Suspicious\ flow)$ is estimated over the suspicious flow pool, whereas $p(t_i|Innocuous\ flow)$ is estimated over the innocuous flow pool. During the match process, the scores λ_i for the tokens t_i contained in the flow under test are summed. The flow matches the signature if the obtained total score Λ exceeds a precomputed threshold θ. This threshold is computed during the signature generation process. Given a predetermined acceptable percentage of false positives r, θ is chosen so that the signature produces less than r false positives and minimizes the number of false negatives at the same time.

The conjunction and token-subsequence signatures are not resilient to noise in the suspicious flow pool. For example, if just one noise flow that does not contain the worm's invariants appears in the suspicious flow pool, the worm's invariants will not be extracted during the preprocessing phase because they are not present in *all* of the flows. For this reason Polygraph [14] applies a hierarchical clustering algorithm during the generation of conjunction and token-subsequence signatures in an attempt to isolate the worm flows from the noise. Each cluster consists of a set of suspicious flows $\{a_1, a_2, .., a_n\}$, and the signature s_a extracted from the set. That is, each cluster can be represented as a pair $(\{a_1, a_2, .., a_n\}, s_a)$. The similarity between two clusters is based on the *specificity* of the signatures, namely, the number of false positives (measured over the innocuous flow pool) produced by the new signature obtained by merging the two clusters. For example, the similarity between two clusters $(\{a_1, a_2, .., a_n\}, s_a)$ and $(\{b_1, b_2, .., b_m\}, s_b)$ is computed as the number of false positives produced by the signature $s_{a,b}$ extracted from the merged set of flows $\{a_1, a_2, .., a_n, b_1, b_2, .., b_m\}$. The algorithm starts with N clusters, one for each suspicious flow, and then proceeds iteratively to merge pairs of the (remaining) clusters. At each step, only the one pair of clusters that upon merging produce the signature with the lowest false positive rate are actually merged. The algorithm proceeds until all the "merged" signatures produce an unacceptable number of false positives or there is only one cluster left.

2.3.3 Misleading Conjunction and Token-Subsequences Signatures

A signature is useful if it contains at least a subset of the invariant substrings of the worm. The hierarchical clustering

algorithm implemented by Polygraph is greedy [14]. This choice is motivated by the fact that a non-greedy clustering algorithm would be computationally expensive. This property can be exploited by injecting well-crafted noise to prevent the generation of a useful signature. Below, we describe how to craft the noise to mislead Polygraph.

Suppose that a polymorphic worm propagates using the scheme described in Section 2.2 (see Figure 1). Suppose also that the *fake anomalous* flow is crafted so that it has some substrings in common with the real worm, but does not contain the *true* invariant parts of the worm, as shown in Figure 2. We call TI (True Invariants) the set of *true* invariant substrings, and FI (Fake Invariants) the set of substrings in common between the worm and its fake anomalous flow. Suppose now that the suspicious flow pool contains three copies of the worm, and then also three corresponding *fake anomalous* flows. We call w_i the i-th copy of the worm in the suspicious flow pool and f_i its *fake anomalous* flow. Note that FI_i is different for different pairs of w_i and f_i because each fake anomalous flow is crafted specifically according to a worm flow, and each worm flow is different due to polymorphism.

The clustering algorithm starts (at step 0) by constructing one signature for each (single) flow in the suspicious flow pool. During the first step of the clustering process, whenever a worm flow w_i and the corresponding fake anomalous flow f_i are considered together, a signature containing the common substrings FI_i will be generated. It is worth noting that the generated signature in this case will not contain TI. Whenever two worm flows w_i and w_j are considered together, a signature containing TI will be generated. Whereas, whenever two fake anomalous flows f_i and f_j or a worm flow w_i and a fake anomalous flow f_j ($j \neq i$, i.e, it is from a different worm flow) are considered together, the generated signature will contain just substrings extracted from the protocol framework PF (and possibly other substrings that are in common just by chance). Obviously, a signature containing mostly tokens extracted from the protocol framework would produce a high number of false positives because the normal/innocuous flows will also need to use the protocol/application and thus can also contain substrings of the protocol framework. Therefore, pairs of w_i and f_j and pairs of f_i and f_j ($i \neq j$) will not be merged. Now, the question is whether a pair of w_i and f_i (resulting in a signature containing FI_i) or a pair of w_i and w_j (resulting in a signature containing TI) will be merged.

Let $p(false\ positive|FI_i)$ and $p(false\ positive|TI)$ be the probabilities that a signature containing FI_i and a signature containing TI will produce a false positive, respectively. If the fake invariants FI_i had been "well-crafted" by the worm during propagation so that $p(false\ positive|FI_i) < p(false\ positive|TI)$, the "merged" signature s_1, produced by the first step of the clustering algorithm (see above) will contain FI_i but will not contain TI. That is, a worm flow and its corresponding fake anomalous flow, say, w_1 and f_1 will be merged. Of course, the question is how to obtain $p(false\ positive|FI_i) < p(false\ positive|TI)$. In Section 2.3.5, we will describe how to produce, in practice, a fake anomalous flow that corresponds to a true worm flow. For now, we state that the FI_i tokens are made of random bytes and that the total number and the lengths of tokens in FI_i are greater than the number and the lengths of tokens in TI. As a result, $p(false\ positive|FI_i) < p(false\ positive|TI)$ will be very likely to hold. To show this, let $p_f(b)$ be the probability of a byte b, contained in a fake invariant token, to appear in an innocuous flow, and $p_t(b)$ the probability of a byte b, contained in a true invariant token, to appear in an innocuous flow. Let the cardinalities of the sets FI_i and TI be $x = |FI_i|$ and $y = |TI|$, respectively, and the lengths of a token $t_{f_k} \in FI_i$ and a token $t_{t_k} \in TI$ be l_k and h_k, respectively. Assuming the bytes of a token to be extracted from a uniform random distribution and assuming the tokens to be statistically independent, we can write:

$$\begin{aligned} p(false\ positive|FI_i) &= \prod_{k=1}^{x} \prod_{j=1}^{l_k} p_f(b_{k,j}) \\ p(false\ positive|TI) &= \prod_{k=1}^{y} \prod_{j=1}^{h_k} p_t(b_{k,j}) \end{aligned} \quad (1)$$

where $b_{k,j}$ is the j-th byte of the k-th token. Now, if we assume that the bytes $b_{k,j}$ have the same probability, p, to be present in an innocuous flow, so that $p_f(b_{k,j}) = p_t(b_{k,j}) = p$, $\forall_{k,j}$, it is easy to see that if $x \cdot avg_k(l_k) > y \cdot avg_k(h_k)$ we can obtain $p(false\ positive|FI_i) < p(false\ positive|TI)$.

Now, returning to the clustering process. At this point, there is one cluster, say, $(\{w_1, f_1\}, s_1)$, and two worm flows and two fake anomalous flows. Consider all the candidates for merging. We already know from the above discussion that if we only consider the four clusters containing a single flow, the only acceptable merging will be between a worm flow and its corresponding fake anomalous flow, say w_2 and f_2, resulting in a signature containing FI_2. But w_2 (or f_2) can also merge with the existing cluster, resulting in a set $\{w_1, f_1, w_2\}$ (or $\{w_1, f_1, f_2\}$). By extracting the substrings common to all the three flows the algorithm would obtain only tokens belonging to the protocol framework (and possibly other small substrings that are common to all three flows just by chance). We call CS_{ij} the signature extracted from $\{w_i, f_i, w_j\}$ (or $\{w_i, f_i, f_j\}$). Note that $TI \not\subseteq CS_{ij}$. Again, $p(false\ positive|FI_j) < p(false\ positive|CS_{ij})$ will very likely hold given that CS_{ij} will mostly contain just tokens from the protocol framework. Therefore, the only acceptable cluster is $\{w_2, f_2\}$.

The algorithm continues and finally there will be three clusters, namely $\{w_1, f_1\}$, $\{w_2, f_2\}$ and $\{w_3, f_3\}$, and three

corresponding signatures. At this point, the clustering algorithm will consider merging the clusters, say, to form $\{\{w_1, f_1\}, \{w_2, f_2\}\}$. But the set of substrings in common among all the four flows will not contain TI. Once again, the signature will mostly contain invariants related to the protocol framework, and as a result will likely produce a high number of false positives. Thus, this cluster is not acceptable, and the clustering algorithm has to terminate.

In conclusion, the noise injection attack misleads Polygraph to generate signatures containing the fake invariant strings (FI_i), rather than a useful signature containing the true invariants (TI).

2.3.4 Misleading Bayes Signatures

To generate Bayes signatures, Polygraph first extracts the tokens of a minimum length α that are common to at least K out of a total number of N suspicious flows. If $K = 0.2 \times N$, as suggested in [14], an attacker can mislead the Bayes signatures by simply programming the worm so that it sends five fake anomalous flows per worm variant because in this case the true invariants (TI) occur in less than 20% of the suspicious flows and will not be extracted/used. It seems then that for a low value of K the worm needs to flood the suspicious flow pool with a large number of fake anomalous flows. However, we show how the worm can craft the fake anomalous flows so that just a few (one or two) of them per worm variant will be sufficient to mislead the generation of Bayes signatures.

If a worm crafts the fake anomalous flows as described in Section 2.3.3, the Bayes signature generation algorithm will very likely generate a *useful* worm signature containing tokens related to the protocol framework PF and the true invariant tokens TI. The tokens PF will be present in 100% of the suspicious flows, whereas the tokens TI will be present in 50% of the suspicious flows if one fake anomalous flow per worm variant is used. The fake invariants FI are specific for each worm variant and its fake anomalous flow. This means each FI_i will, in general, be present less than K times in the suspicious flow pool (unless K is very small) and will not be used to generate the Bayes signatures. In short, the technique described in Section 2.3.3 cannot mislead Bayes signatures.

As described in Section 2.3.2, during the generation of a Bayes signature a score λ_i is computed for each token t_i in the signature. During the matching process, the scores of matched tokens are summed. The technique we develop here is to insert a set of strings in the fake anomalous flows in such a way that the generated signatures contains tokens that will score an innocuous flow higher than a true worm flow, thus making it very hard to set a proper threshold value (θ) to obtain both low false positive and false negative rates.

Consider now a length n string of bytes $v = (v_1, v_2, ..v_n)$ that appears in the innocuous flow pool (but does not appear in the worm flows) with a probability p that is neither too low nor too high, for example $p_1 = 0.05 < p(v|Innocuous\ flow) < 0.20 = p_2$. If v is injected into the fake anomalous flows generated by each variant of the worm, this string will appear in at least 50% of the suspicious flows. This means that the string v will be considered as a token in the Bayes signature. We have $p(v|suspicious\ flow) \geq 0.5$ and $p_1 < p(v|Innocuous\ flow) < p_2$, thus the token v would receive a score λ_v between $log(0.5/p2)$ and $log(0.5/p_1)$. If we split the string v to all the possible substrings of length $m < n$, we will obtain $n - m + 1$ different substrings $v_{1,m} = (v_1, v_2, ..v_m)$, $v_{2,m+1} = (v_2, v_3, ..v_{m+1})$, ..., $v_{n-m+1,n} = (v_{n-m+1}, v_{n-m+2}, ..v_n)$. Suppose now the worm injects all of the $n - m + 1$ substrings randomly (with respect to the position for each substring) in each fake anomalous flow, instead of injecting the entire string v. All of the substrings of v will be present in at least 50% of the suspicious flows in the suspicious flow pool and will therefore be added as tokens into the Bayes signature.

If m is not much lower than n, we can expect that $p(v_{j,j+m-1}|Innocuous\ flow)$ will be not much higher than $p(v|Innocuous\ flow)$. In turn, we expect the score $\lambda_{v_{j,j+m-1}}$ associated with each of the $n - m + 1$ substrings of v to be not much lower than the score λ_v. This results in a multiplying effect on the score of v because a flow that contains v also contains all of its substrings. We will refer to the strings $v_{j,j+m-1},\ j = 1..(n - m + 1)$ as *score multiplier strings*.

The Bayes signatures now include PF, TI and the score multiplier strings. During the matching phase, the total score for a real worm flow is:

$$S = \sum_l \lambda_{PF_l} + \sum_h \lambda_{TI_h} \tag{2}$$

Here λ_{TI_h} is the score of a worm's true invariant token TI_h and λ_{PF_l} is the score of a protocol framework token PF_l (note that the worm will not contain v).

On the other hand, the total score for an innocuous flow containing v is *at least*:

$$\Lambda = \sum_{j=1}^{n-m+1} \lambda_{v_{j,j+m-1}} \tag{3}$$

The innocuous flow contains v and thus all of its substrings, which are tokens in the Bayes signatures (the flow can also contain PF tokens etc.) If the attacker chooses v and m such that $\Lambda > S$, it will be impossible to set a threshold θ for the Bayes signatures that will produce a low false positive rate and low false negative rate at the same time. This is because if $\theta < S$ (and then also $\theta < \Lambda$)

the signature will generate a high number of false positives (from around 5% to 20% for the proposed example $0.05 < p(v|Innocuous\ flow) < 0.20$), due to the presence of v, and then of all its substrings, into a non-negligible percentage of normal traffic. On the other hand, if $\theta > \Lambda$ (and then also $\theta > S$) the Bayes signature will produce around 100% false negatives.

In conclusion, the attacking technique described here prevents the generation of a useful signature. We will discuss in Section 3 and in Appendix A how the attacker can automatically extract a set of *candidate* strings v (and therefore its *score multiplier* substrings) from network traffic traces. The obtained candidate strings can be used to obtain the multiplying effect explained above.

2.3.5 Crafting The Noise

Before propagating to the next victim the worm must first create a polymorphic copy of itself w_i. Then it can create the associated fake anomalous flow f_i using the following algorithm:

a) $f_i^{(0)}$ = **clone**(w_i): Create a copy of w_i.

b) $f_i^{(1)}$ = **randomlyPermuteBytes**($f_i^{(0)}$): Permute the bytes of $f_i^{(0)}$ but leaving the protocol framework bytes unchanged.

c) $a[\ ]$ = **extractFakeInvariants**(w_i,k,l): Copy k substrings of length l from w_i into an array a, choosing them at random, but do not copy substrings that contain protocol framework or true invariant bytes.

d) $f_i^{(2)}$ = **injectFakeInvariants**($f_i^{(1)}$,$a[\ ]$): Copy the fake invariant substrings into $f_i^{(1)}$ but do not overwrite bytes belonging to the protocol framework (see Figure 2).

e) $f_i^{(3)}$ = **injectScoreMultiplierStrings**($f_i^{(2)}$,v): Inject *score multiplier strings* in $f_i^{(2)}$ by splitting a string v as explained in Section 2.3.4. The string v can be chosen from a set of candidate strings obtained by means of an analysis of normal network traffic traces performed using the algorithm explained in Appendix A. The attacker could embed a subset of the candidate strings into the worm's code. The decision on which string v to use can be based on time. For example, the worm could embed the time of its first infection into its code and then use a different string v periodically (e.g., every 10 minutes for a fast-propagating worm). This is necessary because the worm and its fake anomalous flows can arrive at the flow classifiers from multiple infected hosts. Given that the *score multiplier strings* have to be present in a high fraction of the total number of fake anomalous flows into the suspicious pool, the worm cannot just pick v at random each time it propagates to a new victim. Instead, each v has to be used for a period of time.

f) $f_i^{(4)}$ = **obfuscateTrueInvariants**($f_i^{(3)}$): This is necessary because $f_i^{(3)}$ could still contain some true invariant strings, even though just by chance. The obfuscation process assures that $f_i^{(4)}$ will not contain the worm's true invariants.

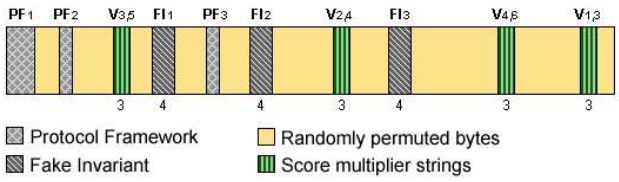

Figure 3: An example of fake anomalous flow

Here $f_i^{(h)}$ represents an "update" of $f_i^{(h-1)}$. The final fake anomalous flow $f_i^{(4)}$ and the worm variant w_i are sent together to the next victim. An example of the application of the above algorithm is reported in Figure 3. The fake anomalous flow has been crafted using $k = 3$ fake invariants of length $l = 4$. The string v is 6 bytes long and the length of the *score multiplier substrings* is $m = 3$. It is worth noting that the resulting *fake anomalous* flow does not contain the true invariant tokens. If the byte frequency distribution of w_i and f_i are not very close (due to the injection of the score multiplier strings) a simple padding technique could be applied to make the two byte frequency distribution closer.

2.3.6 Combining Noise Injection and Red Herring Attacks

In Section 2.3.3 we presented how the fake anomalous flows can be crafted to mislead the generation of Conjunction and Token-subsequences signatures. For such attack to be successful, fake anomalous flows generated by different worm variants should not contain common substrings. The attacking method presented in Section 2.3.4 to mislead the generation of Bayes signatures violates this constraint because all the fake anomalous flows in the suspicious flow pool have to contain the same *score multiplier strings*. However, this turns out not to be a problem. During the application of the hierarchical clustering algorithm, whenever two fake anomalous flows f_i and f_j are involved in a merge, the extracted tokens will be either part of the protocol framework or *score multiplier* substrings. Therefore, the generated signature will very likely produce a high number of false positives and the flows will not be kept in the same

cluster. It is then very likely to see (following the analysis in Section 2.3.3) that the only acceptable clusters are $\{w_i, f_i\}$. Thus, the attack against Bayes signatures described in Section 2.3.4 does not interfere with the attack against Conjunction or Token-subsequence signatures. It follows that crafting the fake anomalous flows as described in Section 2.3.5, the attack is effective against the three different types of Polygraph signature generation algorithms.

However, the results of the attack are not deterministically predictable. As mentioned in Section 2.3.3 it is possible that a set of flows contains some substrings that are common just by chance to all the flows in the set. For example it could happen that two worm variants w_i and w_j present (by chance) a common substring $c_{i,j}$, besides the protocol framework and true invariant tokens. This means that to avoid w_i and w_j being kept in the same cluster, the constraint $p(false\,positive|FI) < p(false\,positive|TI, c_{i,j})$ needs to be verified. Given that $c_{i,j}$ is unknown, it is not easy to craft the set of fake invariants FI to assure that this constraint is satisfied. Besides, even if the worm crafts FI so that $p(false\ positive|FI)$ is close to zero, it can also happen that $p(false\ positive|TI, c_{i,j}) = 0$. In this case there is no way to determine which signature is more specific than the other, and we assume the merged cluster to be kept is chosen at random.

We will show in Section 3 that in practice the probability of success for the noise injection attack is fairly high. To further increase the success chance of the noise injection attack, it is possible to combine it with the *red herring* attack discussed by Polygraph's authors in [14]. The worm variants could include some *temporary true invariants* that change over time. If the Conjunction and Token-subsequence signature generation algorithms produce (by chance) a useful signature, this signature would become useless over a certain period of time. After this period of time Polygraph could try to generate again new Conjunction and Token-subsequence signatures to detect the worm. Nevertheless, this time Polygraph may not be as "fortunate" as the first time in generating a useful signature. Besides, if the *temporary true invariants* were chosen among high frequency strings (e.g., extracted from network traces using the algorithm presented in Appendix A setting the probability between 0.8 and 1), the related tokens would receive a low score during the generation of the Bayes signature and therefore would not interfere with the noise injection attack against Bayes signatures. The final result is that the attacker has a very high probability to succeed in misleading all the three types of signatures at the same time.

2.4 Effects of the Noise on Other Automatic Signature Generators

We have performed experiments only on Polygraph. However, it is possible to evaluate the effects of different noise injection attacks on other systems basing the analysis on the description of the signature generation algorithms. We present an analysis of the possible effects of noise injection attack on Nemean [28].

Nemean is a recently proposed automatic signature generator that uses a semantic analysis of the network protocols and two types of signatures, namely connection and session signatures [28]. It uses a honeynet to collect the suspicious flow pool. Then it applies a clustering algorithm to group similar connections in *connection clusters* and similar sessions in *session clusters*. Each cluster contains the observed variants of the same worm. Even though Nemean is suitable for generating signatures for worms that use limited polymorphism [28], it introduces interesting features such as semantic protocol analysis and connection and session clustering. For this reason, it is interesting to discuss how it could be misled using the noise injection attack.

Nemean represents a connection by a vector containing the distribution of bytes, the request type and the response codes that occurred in the network data [28]. The *fake anomalous* flows can be injected into the suspicious flow pool as explained in Section 2.2. Given that the *fake anomalous* flows can be crafted to have the same protocol framework and (almost perfectly) the same distribution of bytes as the worm variant they derive from, the *fake anomalous* flows and the worm variants will be very likely considered in the same connection cluster. If the *fake anomalous* flows are crafted by applying a random permutation of the worm's bytes (see Section 2.3.5), the signature generation algorithm will not be able to discover significant invariant parts common to the flows in a cluster, and the extracted connection signatures will be useless because they will likely produce a high number of false positives. This noise injection attack will affect the session signatures as well, given that they are constructed based on the results produced by the connection clustering process [28].

3 Experiments

In our experiments we tried to have an experimental setup similar to the one reported in [14] in order to make the results comparable. Polygraph software is not publicly available, therefore we implemented our own version following the description of the algorithms in [14].

3.1 Experimental Setup

Polygraph setup. We performed all the experiments setting the minimum token length $\alpha = 2$ and the token-extraction threshold for Bayes signature generation to be 20% of the total size of the suspicious flow pool. We also set the minimum cluster size to 3 and the maximum acceptable false positive rate for a signature to be 0.01 during the application of the hierarchical clustering algorithm for Conjunction and Token-subsequences signatures.

Polymorphic worm. We considered the Apache-Knacker exploit reported in [14] as the attack vector for the worm. We simulated an ideal polymorphic engine following the same idea used by Polygraph's authors, keeping the protocol framework of the attack and the first two byte of the return address fixed and filling the wildcard and code bytes uniformly at random. Each worm variant matches the regular expression:

```
GET .* HTTP/1.1\r\n.*\r\nHost: .*\r\n
.*\r\nHost: .*\xFF\xBF.*\r\n
```

Datasets. We collected several days of HTTP requests taken from the backbone of a busy aggregated /16 and /17 academic network (i.e., CIDR [3] blocks of the form `a.b.0.0/16` and `c.d.e.0/17`) hosting thousands of machines. The collected traffic contains requests towards thousands of different public web-servers, both internal and external with respect to our network. The network traffic traces were collected between October and November 2004. We split the traffic traces to obtain three different datasets which are described below.

Innocuous flow pool. The innocuous flow pool was made of 100,459 flows related to HTTP requests towards 898 different web-servers[1]. Among these, 7 flows matched the same regular expression as the polymorphic worm. Thus, in absence of noise in the suspicious flow pool, a generated signature that matched the worm invariants would result in around 0.007% of false positives on the innocuous flow pool. These 7 flows were the only ones to contain the \xFF\xBF string. Very similar to our traffic data, the \xFF\xBF string was present in 0.008% of the evaluation flows used by Polygraph's authors to perform their experiments [14]. In [14] the \xFF\xBF token caused the Bayes signature to produce 0.008% of false positives.

Test flow pools. We used two sets of test flows in our experiments. The first set was made of 217,164 innocuous flows[1] extracted from the traffic traces. We inspected this test set to ensure that it did not include any flow containing the \xFF\xBF string. The second test set was made of 100 simulated worm variants. We used the first test set to measure the false positive rate and the second to measure the false negative rate produced by the signatures. Note that we obtained the innocuous flow pool and the test set made of innocuous flows from two different slices of the network traces.

Score multiplier strings. We used a dataset made of 5,000 flows to extract the score multiplier strings. We analyzed the flows using the algorithm presented in Appendix A. We extracted all the substrings of length from 6 to 15 bytes having an occurrence frequency between 0.05 and 0.2, obtaining around 300 different strings. Many of them were strings related to HTTP-header fields introduced by certain browsers, such as "Cache-Control', "Modified-Since", "Firefox/0.10.1", "Downloader/6.3", etc. The extracted strings are the candidate strings that can be used to obtain a score multiplying effect to force Bayes signatures to generate a high number of false positives, as explained in Section 2.3.4. It is worth noting that the flows used to extract the score multiplier strings contained both inbound and outbound HTTP requests taken from the perimeter of our network. The flows were related to requests among a large number of different web-servers and clients. For these reasons we expect the obtained strings and occurrence frequencies to be general and not specific just to our network[2].

Fake anomalous flows. We crafted the fake anomalous flows using the algorithm presented in Section 2.3.5. We used $k = 2$ fake invariants of length $l = 5$ for all the fake anomalous flows. We used several combinations of score multiplier strings v by splitting them in different ways to obtain a different number of substrings for each test. For each fake anomalous flow, we chose $\frac{2}{3}$ of the obtained substrings at random and injected them into the flow[3].

3.2 Misleading Bayes Signatures

In [15] Polygraph's authors state that Bayes signatures are resilient to the presence of noise into the suspicious flow pool until the noise level reaches at least 80% of the total number of flows. In our experiments we found that if the fake anomalous flows are properly crafted, just 50% of noise in the suspicious flow pool (i.e., 1 fake anomalous flow per worm variant) can make the generated signature useless. We performed several experiments using 10 worm variants and 1 or 2 fake anomalous flows per variant in the suspicious flow pool. The fake anomalous flows were crafted as explained in Section 2.3.5 and 3.1. We report the results of two group of tests below.

Case 1. We obtained the best result using "Firefox/0.10.1" (12.2%) and "shockwave-flash" (11.9%) as

[1]The fbws were "innocuous" in the sense that they did not contain the considered worm.

[2]The extracted strings could obviously present different occurrence frequencies over time. Nevertheless it is reasonable to assume that the attacker could perform a similar analysis on traffi c traces collected just a few weeks or even days before launching the attack.

[3]Thus, the fake anomalous fbws did not always contain the same set of substrings.

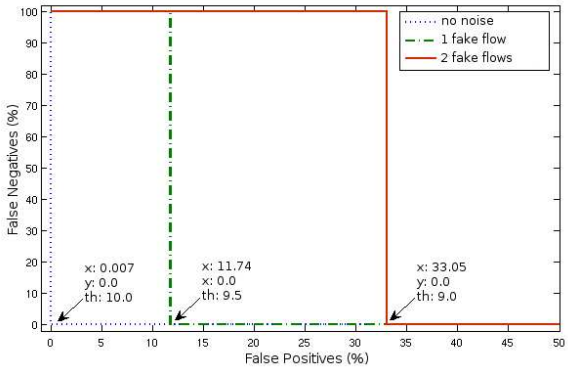

Figure 4: Case 1. The false positives are measured over the innocuous fbw pool

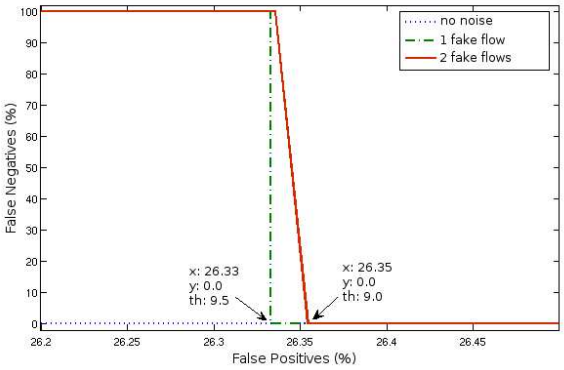

Figure 5: Case 1. The false positives are measured over the test fbw pool

score multiplier strings. The percentages between parenthesis represent the occurrence frequencies of the strings (see Section 3.1). We split the two score multiplier strings to obtain all the possible substrings of size $m = 9$ (e.g., "Firefox/0", "irefox/0.", "refox/0.1", etc.).

As described above, we simulated two attack scenarios using 1 and 2 fake anomalous flows per worm variant, respectively. Therefore, the suspicious flow pool was made of 20 flows during the first attack scenario and of 30 flows during the second one. We generated the Bayes signature on the suspicious flow pool and measured the false positive rates on the innocuous flow pool and the test flow pool made of innocuous traffic. The results are shown in Figure 4 and Figure 5. Please note that the graphs are represented on different ranges of false positives to highlight the difference between the two attack scenarios. The plots represent the false positives and false negatives produced by the signature while varying the threshold θ starting from 0.0 and incrementing it using a 0.5 increment step. A threshold equal to 0.0 obviously produces 100% of false positives and 0% of false negatives. By incrementing the threshold, the percentage of false positives decreases. The arrows indicate the coordinates related to the maximum value of the threshold that produces no false negatives. The Bayes signature generated during the second scenario is reported in Appendix B.

In Section 2.3.2 we discussed how Polygraph optimizes the threshold θ for Bayes signatures. It is easy to see from Figure 4 that the noise injection attack prevents the threshold θ to be optimized. Consider for example the graph related to the injection of 1 fake anomalous flow per worm variant. If $\theta = 9.5$, the signature generates 11.74% of false positives and 0% of false negatives. In order to decrease the number of false positives the threshold would need to be incremented further. However, as soon as the threshold exceeds 9.5 the signature produces 100% of false negatives.

Case 2. In this case "Pragma: no-cache" (9.4%) and "-powerpoint" (7.0%) were used as score multiplier strings. We split these two strings to obtain all the substrings of length $m = 4$. Again, the suspicious flow pool contained 10 worm variants and 1 or 2 fake anomalous flows per variant. The results are reported in Figures 6 and 7. Please note that, again, the graphs are represented on different ranges of false positives to highlight the difference between the two attack scenarios. The Bayes signature generated during the second scenario (2 fake anomalous flows per worm variant) is reported in Appendix B.

3.3 Misleading All The Three Signatures at The Same Time

The objective of the noise injection attack is to prevent the generation of useful signatures. In order to achieve this result the attack needs to prevent the generation of useful conjunction, token-subsequences, and Bayes signatures at the same time. As discussed in Section 2.3.6, the results of the attack are not deterministically predictable. In order to estimate the probability of success we simulated the noise injection attack multiple times. We considered an attack successful if Polygraph did not generate a conjunction or token-subsequence signature that would match the worm and if the Bayes signature produced more than 1% of false positives measured over the innocuous flow pool. Even though a false positive rate around 1% is seemingly low, we consider it intolerable for a *blocking* signature. We report the results with fake anomalous flows crafted using two different combinations of score multiplier strings. We divided the tests into two groups. The first group of tests were performed using "Forwarded-For" (11.3%) and "Modified-Since" (15.2%) as score multiplier strings, splitting them into substrings of length $m = 5$. The second group of test were performed using "Cache-Control" (15.1%) and "Range: bytes" (11.9%), splitting them in substrings of length $m = 4$. For each group of tests we simulated two

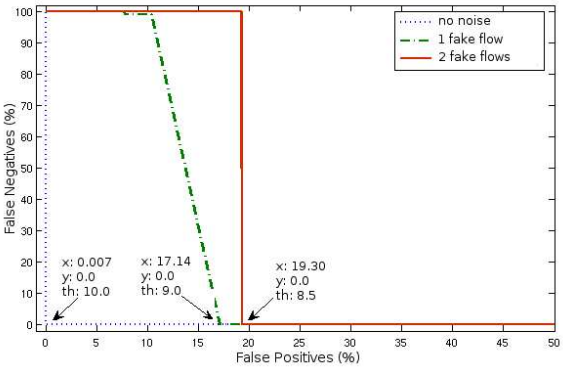

Figure 6: Case 2. The false positives are measured over the innocuous fbw pool

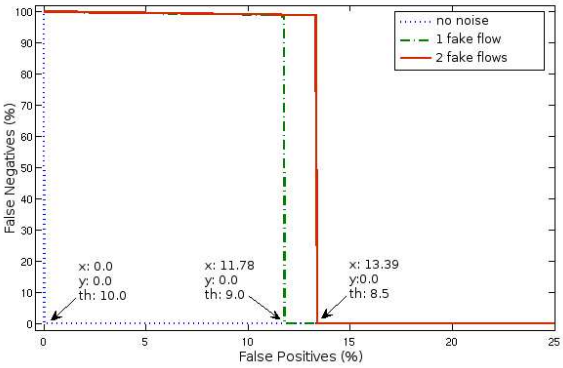

Figure 7: Case 2. The false positives are measured over the test fbw pool

noise injection attack scenarios using 1 and 2 fake anomalous flows per worm variant, respectively. We used 5 worm variants in the suspicious flow pool for both the first and the second scenario. We generated the signatures 45 times for the first group of tests and 20 times for the second group. The results are shown in Table 1 and Table 2. The reported percentages represent how many times the attack was successful in avoiding the generation of useful signatures. The first three rows report the percentage of success computed for each type of signatures, individually. The last row represents the percentage of attacks that succeeded in misleading Polygraph so that it could not generate any useful signature, regardless of the signature type. It is worth noting that in both experiments, when using 2 fake anomalous flows per worm variant, the attack has a higher probability to succeed, and further, it prevents Polygraph from generating a useful Bayes signature 100% of the time.

	1 fake anomalous flow	2 fake anomalous flows
Conjunction	73.3%	88.9%
Token-subsequences	60.0%	73.3%
Bayes	100%	100%
All three signatures	44.4%	**62.2%**

Table 1: Percentage of successful attacks (using "Forwarded-For" and "Modifi ed-Since")

	1 fake anomalous flow	2 fake anomalous flows
Conjunction	65%	95%
Token-subsequences	40%	90%
Bayes	90%	100%
All three signatures	20%	**85%**

Table 2: Percentage of successful attacks (using "Cache-Control" and "Range: bytes")

4 Discussion and Future Work

Polygraph's authors showed that their system is resilient to the presence of as much as 80% of "normal" noise in the suspicious flow pool [14]. However, we showed that if the noise is properly crafted, just 50% of noise could prevent Polygraph from generating useful signatures a majority of the times. In addition, as explained in Section 2.3.6, the noise injection attack can be easily combined with the red herring attack discussed in [14]. The combination of the two attacks increases the probability that the worm will prevent the generation of a useful signature.

We also conducted experiments on NETBIOS traffic to extract score multiplier strings that can be used by a worm that uses this protocol as attack vector. We chose NETBIOS because it is an attack vector for most of the *OS-based* worms. We analyzed more than 5,000 NETBIOS flows, searching for strings of length from 6 to 15 bytes and an occurrence frequency between 0.05 and 0.2. We found 29 candidate strings in "TCP-based" NETBIOS traffic and 58 candidate strings in "UDP-based" requests. This experiment suggests that our noise injection technique using "score multiplier" strings can work for a variety of protocols. In our future work we plan to perform further experiments using worms that are based on other protocols, e.g., NETBIOS, as attack vectors.

A possible defense against our implementation of the noise injection attack is to use a white list to attempt to filter out flows that contain the score multiplier substrings. However, this is not straightforward and may not even be possible. As shown in Section 3.1, there are a very large number of strings that a worm can potentially use. The set of candidate strings extracted from the traffic are determined by the occurrence frequency ranges, and the sets of substrings are determined by the string length value. These are chosen by the attacker and are not known *a priori* to the signature generator. Further, the strings actually used by a worm in-

stance to create fake anomalous flows can change over time. As a result, a reliable way to filter out the fake anomalous flows is to look for occurrences of all possible substrings of a very large set of strings. This can be very expensive. Further, such aggressive filtering may prevent the system from producing useful worm signatures that happen to contain such substrings.

Another possible countermeasure against the score multiplier strings technique is to modify the detection algorithm for Bayes signatures. For example, every time a test flow matches a token, the related bytes in the flow should be marked to prevent them from "partecipating" in matching another token of the same signature. This means that the score mutiplier effect described in Section 2.3.4 cannot be achieved anymore. However, the attack may still work if multiple candidate strings v (see Section 2.3.4) are carefully chosen and if they are split without overlap, although now the induced false positive rate may be much less than the one obtaind during the experiments reported in Section 3.2.

Even if the above countermeasures happen to work in some cases, the fundamental problem still exists: without an accurate and robust flow classifier that can prevent the injection of fake anomalous flows, syntactic-based automated signature generators are vulnerable. The noise injection attack we have described in this paper is proof-of-concept. We suspect there are many other similar attacks. We believe that in order to accurately and reliably filter out fake anomalous flows, we must use multiple semantic-aware sensors (on hosts and networks) and correlate their observations. This will be part of our future work.

5 Related Work

In addition to the relevant research discussed throughout the paper, there are several other related works. Our work fits into the larger field of IDS evasion. Researchers in this area have used TCP/IP transformations to demonstrate IDS evasions [16], and address weaknesses created by ambiguities in network protocols [4]. Numerous tools have been created, including `fragroute` [23], `snot` [22], and `mucus` [13]. Some authors have investigated techniques to automate the generation of evasive attacks. For example, in [26], the authors identified mutation operations to generate variations on known exploits. Similarly, the authors in [19] modeled attack transformations to derive new variations on known attacks. None of these evasions used polymorphism, let alone the misleading or evasive polymorphism considered in our work.

Polymorphic packers and unpackers are of course common in virus toolkits [9, 20]. A few worms also contain polymorphic engines [24]. For the most part, however, these polymorphic engines are simple, and do not produce evasive polymorphic code. Noted exceptions are [2], and [7], which are closest to our work. In [2] the authors used spectral analysis to create blended shellcode that evaded an anomaly detection system that used data mining. In [7], the authors presented a technique to generate *blending polymorphic worms* (instead of just simple polymorphic worms) which evade PAYL [27]. These works addressed *evasive polymorphism* to evade anomaly-based IDS, whereas our work addresses *misleading polymorphism* that aims at misleading the signature extraction algorithms used by syntactic-based automatic signature generators.

6 Conclusion

Syntactic-based automatic worm signature generators, e.g., Polygraph, typically assume that a set of suspicious flows are provided by a flow classifier that often introduces "noise". The algorithms for extracting the worm signatures from the flow data are designed to cope with the noise. It has been reported that these systems can handle a fairly high noise level, e.g., 80% for Polygraph. In this paper, using Polgraph as a case study, we described a new and general class of attacks whereby a worm can combine polymorphism and misleading behavior to generate *fake anomalous flows* to intentionally pollute the dataset of suspicious flows. In order to mislead Polygraph, we presented a noise injection attack whereby the fake anomalous flows are crafted to contain *fake invariants* that mislead the signature extraction algorithms. We presented several techniques, in particular, *injection of score multiplier* substrings. Our experiments showed that just injecting one fake anomalous flow for each actual worm flow, i.e., a 50% noise level, the worm can already prevent Polygraph from reliably generating useful signatures. Our study suggests that unless an accurate and robust flow classification process is in place, automatic syntactic-based signature generators can fail in face of deliberate noise injection attacks.

References

[1] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[2] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. Underduk. Polymorphic shellcode engine using spectrum analysis. *Phrack*, 0x3d, 2003.

[3] V. Fuller, T. Li, J. Yu, and K. Varadhan. Classless inter-domain routing (cidr): an address assignment and aggregation strategy,. `http://www.ietf.org/rfc/rfc1519.txt`, 1993.

[4] M. Handley and V. Paxson. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[5] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, October 2003.

[6] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[7] O. Kolesnikov, D. Dagon, and W. Lee. Advanced polymorphic worms: Evading ids by blending in with normal traffi c. Technical report, Georgia Institute of Technology, 2004. `http://www.cc.gatech.edu/~ok/w/ok_pw.pdf`.

[8] C. Kreibich and J. Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. In *Proceedings of the 2nd ACM Workshop on Hot Topics in Networks*, November 2003.

[9] Ktwo. ADMmutate Shellcode mutation Engine. `http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz`, 2001.

[10] Z. Liang and R. Sekar. Automatic generation of buffer overfbw attack signatures: An approach based on program behavior models. In *Proceedings of the 21st Annual Computer Security Applications Conference*, December 2005.

[11] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. *IEEE Security & Privacy*, pages 33–39, July-August 2003.

[12] D. Moore, C. Shannon, and J. Brown. Code-red: a case study on the spread and victims of an internet worm. In *Proceedings of the Internet Measurement Workshop (IMW)*, November 2002.

[13] D. Mutz, G. Vigna, and R. Kemmerer. An experience developing an ids stimulator for the black-box testing of network intrusion detection systems. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2003.

[14] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.

[15] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *Proceedings of the 12th Network and Distributed System Security Symposium*, February 2005.

[16] T. H. Ptacek and T. N. Newsham. Denial of service: Eluding network intrusion detection. Technical Report T2R-0Y6, Secure Networks, Inc., 1998.

[17] S. S. Response. W32.sasser.worm. `http://securityresponse.symantec.com/avcenter/venc/data/w32.sasser.worm%.html`, 2004.

[18] S. S. Response. W32.zotob.a. `http://securityresponse.symantec.com/avcenter/venc/data/w32.zotob.a.htm%l`, 2005.

[19] S. Rubin, S. Jha, and B. P. Miller. Automatic generation and analysis of nids attacks. In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC)*, 2004.

[20] M. Sedalo. Jempi Scodes: Polymorphic shellcode generator. `http://www.shellcode.com.ar/en/proyectos.html`, 2003.

[21] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fi ngerprinting. In *Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation*, December 2004.

[22] Sniphs. Snot. `http://www.stolenshoes.net/sniph/index.html`, 2003.

[23] D. Song. Fragroute. `http://www.monkey.org/~dugsong/fragroute/`, 2005.

[24] J. Stewart. Phatbot trojan analysis. `http://www.lurhq.com/phatbot.html`, 2004.

[25] Y. Tang and S. Chen. Defending against internet worms: A signature-based approach. In *Proceedings of the 24th Annual Conference IEEE INFOCOM 2005*, March 2005.

[26] G. Vigna, W. Robertson, and D. Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, 2004.

[27] K. Wang and S. J. Stolfo. Anomalous payload-based network intrusion detection. In *Proceedings of the 7th Symposium on Recent Advances in Intrusion Detection*, September 2004.

[28] V. Yegneswaran, J. T. Giffi n, P. Barford, and S. Jha. An architecture for generating semantics-aware signatures. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.

A Extracting score multiplier strings from network traces

The objective of the algorithm presented below is to extract a set of strings of length from l_1 to l_2 that appear in a set of flows F with a probability p between p_1 and p_2. The probability p of a string to appear in F is computed as the ratio of the number of flows that contain the string and the total number of flows in F. These strings can be extracted using a *tree* structure. We present the main idea of the algorithm first, and then we go into the details.

Each flow in the set of flows is identified by a ID number. All the strings of a given length l are stored in a tree structure along with the IDs of the flows that contain the string. Each node at depth l in the tree represents the l-th character of a substring in the flows. The algorithm starts considering strings made of one character and then construct the tree iteratively. For a given string $s^{(l)} = (s_1 s_2 \cdots s_l)$ in a flow, the algorithm traverses the tree to its prefix $s^{(l-1)} = (s_1 s_2 \cdots s_{l-1})$ and a child node corresponding to s_l is created/updated. After processing all the strings of length l, the "depth-l" nodes (and edges) that has a probability $p < p_1$ are pruned. Intuition is that if the probability of a string s is smaller than p_1, then the probabilities of all the strings with prefix equal to s are smaller than p_1, too. For a given node n, the probability p is computed based on the list of its flow IDs as described below.

The details of the algorithm are as follows. It starts with an empty root node r. During each step the algorithm ex-

plores all the flows in F one at a time. The first step considers all the strings of length $l = 1$ in the flows. Therefore, for every character in the flows, the algorithm checks whether a node corresponding to the considered character is present in the tree. If a node already exists, the algorithm checks if the current flow (the one from which the character has been extracted) is present in the list of flow IDs associated to the node. If not, the flow ID is added in its flow list and the algorithm goes on considering the next character in the flow. If the current flow ID is present, instead, the algorithm does nothing and continues with the next character. If the node is not present, a node corresponding to the character is added as a child of the root node and its flow IDs list is initialized with the current flow ID. After all the flows have been processed, the probability p of each node n in the tree is computed as the ratio of the number of different flow IDs associated to n and the total number of flows in F. Now, the "depth-1" nodes which have probability $p < p_1$ to appear in the set of flows F are pruned.

This procedure is iteratively repeated for increasing string lengths. At any step l, for each strings $s^{(l)}$ of length l in the flows, the algorithm traverses the tree to the node at depth $(l-1)$ corresponding to the prefix of $s^{(l)}$, namely $s^{(l-1)}$. If a node in the path towards $s^{(l-1)}$ was marked as pruned, it means that probability of a prefix of $s^{(l-1)}$ to appear in F is less than p_1. Thus, the tree do not need to be "updated" with a new node and the string $s^{(l)}$ is disregarded. If the node n at depth $(l-1)$ that represents $s^{(l-1)}$ is reached, then the algorithm checks whether a child of n corresponding to the last character of $s^{(l)}$ is present in the tree. If this child exists, the current flow ID is included into the flow IDs list of the child (if not yet present) and then the algorithm continue considering the next string of length l in the flows. Otherwise, a node corresponding to the last character of $s^{(l)}$ is added as a child of n and its IDs list is initialized. The algorithm continues until all the flows have been processed. Afterwords, it traverses the tree to prune all the nodes at depth l which have a probability p less the p_1 to appear in the flows. The algorithms iteratively repeat the explained steps until it reaches the desired maximum length l_2. Once l_2 has been reached, all the strings in the tree whose probability p is greater than p_2 are removed and the remaining strings that have a length greater than l_1 are shown.

B Bayes signatures

The two signatures reported below were generated over a suspicious flow pool containing 30 flows, 10 worm variants and 2 fake anomalous flow per variant, as described in Section 3.2. In the first case (Signature 1) the fake anomalous flows were crafted as explained in Section 2.3 using "Firefox/0.10.1" (0.122) and "shockwave-flash" (0.119) as score multiplier strings, splitting them in all the substrings of length $m = 9$. It is worth noting that the signature contains tokens that are less long than 9 characters. For example "ave-fla" is 7 characters long. In [15] Polygraph's authors described an algorithm to filter the non distinct tokens. If a token t_1 is a substring of another token t_2, they are considered distinct if t_1 appears at least in $K = 20\%$ (considering our experimental setup) of the suspicious flows not as a substring of t_2. "ave-fla" appears in the flows as substring of "kwave-fla", but also as substring of "wave-flas" and "ave-flash". "ave-fla" and "kwave-fla" will then be deemed distinct given that "wave-flas" and "ave-flash" appear in more than 20% of the suspicious flows, which means that "ave-fla" appears in more than 20% of suspicious flows not as a substring of "kwave-fla". The same holds if "ave-fla" is considered as a substring of either "wave-flas" or "ave-flash". Therefore, "ave-fla" will be considered a distinct token with respect to all the three considered strings and will be used in the signature.

In the second case (Signature 2) the fake anomalous flows were crafted using "Pragma: no-cache" (0.094) and "-powerpoint" (0.07) as score multiplier strings.

Please note that both signatures are very long and some tokens have been omitted here for the sake of brevity.

Signature 1

```
<BayesSignature>
  <token>ckwave-fl</token><score>1.226376887312</score>
  <token>ave-fla</token><score>1.574683581580</score>
  <token>ockwave</token><score>1.73669364837</score>
  <token>kwave-fl</token><score>1.514058959763</score>
  <token>GET </token><score>0.001653782157429</score>
  <token>\r\nHost: </token><score>0.01088936346</score>
  <token>ox/0.10.1</token><score>1.00568567265</score>
  <token>Firefox/0</token><score>0.0</score>
  <token>refox/0</token><score>0.901665207652</score>
  <token>ave-f</token><score>1.737202511078</score>
  <token>shockwave</token><score>1.226376887312</score>
  <token>ockwave-f</token><score>1.226376887312</score>
  <token>\r\n</token><score>2.190187966704E-4</score>
  <token>kwave-f</token><score>1.68590921669</score>
  <token>efox/0.10</token><score>0.581901113127</score>
  <token>HTTP/1.1\r\n</token><score>0.456821797</score>
  <token>ckwave-f</token><score>1.631841995420</score>
  <token>kwave-fla</token><score>1.226376887312</score>
  <token>efox/0</token><score>1.073515464579</score>
  <token>irefox/0</token><score>0.431661578406</score>
  <token>\xFF\xBF</token><score>8.47298252532</score>
  <token>efox/0.1</token><score>1.217889879847</score>
  <token>ave-flash</token><score>0.820911779203</score>
  <token>refox/0.1</token><score>0.869583185579</score>
  <token>fox/0.10.</token><score>0.687231941540</score>
  ...
</BayesSignature>
```

Signature 2

```
<BayesSignature>
<token>cac</token><score>1.4623066797265616</score>
<token>\r\n</token><score>2.1901879667042l8E-4</score>
<token>-cac</token><score>1.1849171593021348</score>
<token>agma</token><score>1.4572109198109477</score>
<token>pow</token><score>2.0022106645559026</score>
<token>oint</token><score>1.4303079284389304</score>
<token>poin</token><score>1.3140517165922387</score>
<token>GET </token><score>0.001653782157429088</score>
<token>-pow</token><score>1.441044976439887</score>
<token>erpo</token><score>1.4413254026896216</score>
<token>ower</token><score>1.5247585609776242</score>
<token>powe</token><score>1.630008221107803</score>
<token>no-c</token><score>1.203481845586463</score>
<token>gm</token><score>1.7951796395818411</score>
<token>poi</token><score>1.9422437382750952</score>
<token>ragm</token><score>1.2902554787976765</score>
<token>\r\nHost: </token><score>0.010889363469</score>
<token>rag</token><score>1.8074424891398477</score>
<token>Prag</token><score>1.6005090604829408</score>
<token>owe</token><score>1.6980386101305638</score>
<token>we</token><score>1.4305200746247666</score>
<token>ache</token><score>0.501173539592899</score>
<token>HTTP/1.1\r\n</token><score>0.4568532348</score>
<token>o-c</token><score>1.6041495913093828</score>
<token>werp</token><score>1.4371272437109852</score>
<token>rpo</token><score>1.9473944422216436</score>
<token>ca</token><score>0.06312339137098681</score>
<token>\xFF\xBF</token><score>8.47298252532043</score>
<token>erp</token><score>1.9934036721698547</score>
<token>wer</token><score>1.8309304120713068</score>
<token>-ca</token><score>1.5050574981045213</score>
<token>gma:</token><score>1.1950439523747765</score>
<token>cach</token><score>0.934411500514455</score>
<token>gma</token><score>1.7331763923440477</score>
<token>che</token><score>0.7498172832696891</score>
...
</BayesSignature>
```