Hamsa*: Fast Signature Generation for Zero-day Polymorphic Worms with Provable Attack Resilience

Zhichun Li   Manan Sanghi   Yan Chen   Ming-Yang Kao   Brian Chavez

Northwestern University
Evanston, IL 60208, USA
{lizc,manan,ychen,kao,cowboy}@cs.northwestern.edu

Abstract

Zero-day polymorphic worms pose a serious threat to the security of Internet infrastructures. Given their rapid propagation, it is crucial to detect them at edge networks and automatically generate signatures in the early stages of infection. Most existing approaches for automatic signature generation need host information and are thus not applicable for deployment on high-speed network links. In this paper, we propose Hamsa, a network-based automated signature generation system for polymorphic worms which is fast, noise-tolerant and attack-resilient. Essentially, we propose a realistic model to analyze the invariant content of polymorphic worms which allows us to make analytical attack-resilience guarantees for the signature generation algorithm. Evaluation based on a range of polymorphic worms and polymorphic engines demonstrates that Hamsa significantly outperforms Polygraph [16] in terms of efficiency, accuracy, and attack resilience.

1 Introduction

The networking and security community has proposed intrusion detection systems (IDSes) [19, 22] to defend against malicious activity by searching the network traffic for known patterns, or *signatures*. So far these signatures for the IDSes are usually generated manually or semi-manually, a process too slow for defending against self-propagating malicious codes, or *worms*, which can compromise all the vulnerable hosts in a matter of a few hours, or even a few minutes [25]. Thus, it is critical to automate the process of worm detection, signature generation and signature dispersion.

*Hamsa (pronounced 'hum-sa') is the sanskrit word for the swan bird which has the mystical potency of separating out the milk from a mixture of milk and water.

To evade detection by signatures, attackers could employ *polymorphic* worms which change their byte sequence at every successive infection. Our goal is to design an automatic signature generation system for polymorphic worms which could be deployed at the network level (gateways and routers) and hence thwart a zero-day worm attack.

Such a signature generation system should satisfy the following requirements.

Network-based. Most of the existing approaches [4, 14, 26, 31] work at the host level and usually have access to information that is not available at the network routers/gateways level (*e.g.*, the system calls made after receiving the worm packets). According to [25], the propagation speed of worms in their early stage is close to exponential. So in the early stage of infection only a very limited number of worm samples are active on the Internet and the number of machines compromised is also limited. Hence, it is unlikely that a host will see the early worm packets and be able to respond in the critical early period of attack. Therefore, the signature generation system should be network-based and deployed at high-speed border routers or gateways that sees the majority of traffic. The requirement of network-based deployment severely limits the design space for detection and signature generation systems and motivates the need for high-speed signature generation.

Noise-tolerant. Signature generation systems typically need a flow classifier to separate potential worm traffic from normal traffic. However, network-level flow classification techniques [10, 18, 28–30] invariably suffer from false positives which lead to noise in the worm traffic pool. Noise is also an issue for honeynet sensors [12, 26, 31]. For example, attackers may send some legitimate traffic to a honeynet sensor to pollute the worm traffic pool and to evade noise-intolerant signature generation.

Attack-resilient. Since the adversary for the algorithm is a human hacker, he may readily adapt his attacks to evade the system. Therefore, the system should not only work for known attacks, but also be resilient against any possible evasion tactics.

Efficient Signature Matching. Since the signatures generated are to be matched against every flow encountered by the NIDS/firewall, it is critical to have high-speed signature matching algorithms. Moreover, for the network-level signature matching, the signatures must solely be based on the network flows. Though it is possible to incorporate host-based information such as system calls in the signatures, it is generally very difficult to get efficient matching for these signatures on the network level.

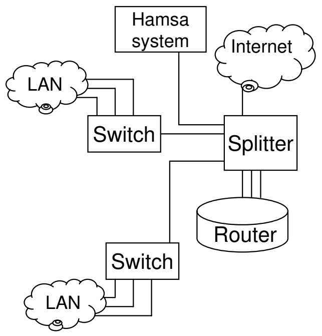

Figure 1. Attaching Hamsa to high-speed routers

Towards meeting these requirements, we propose Hamsa, a network-based automatic signature generation system designed to meet the aforementioned requirements. Hamsa can be connected to routers via a span (mirror) port or an optical splitter as shown in Figure 1. Most modern switches are equipped with a span port to which copies of the traffic from a list of ports can be directed. Hamsa can use such a span port for monitoring all the traffic flows. Alternatively, we can use a splitter such as a Critical Tap [6] to connect Hamsa to routers. Such splitters are fully passive and used in various NIDS systems to avoid affecting the traffic flow.

Hamsa is based on the assumption that a worm must exploit one or more server specific vulnerabilities. This constrains the worm author to include some *invariant* bytes that are crucial for exploiting the vulnerabilities [16].

We formally capture this idea by means of an adversary model Γ which allows the worm author to include any byte strings for the worm flows as long as each flow contains tokens present in the invariant set $\mathcal{I}$ in any arbitrary order. Under certain uniqueness assumptions on the tokens in $\mathcal{I}$ we can analytically guarantee signatures with bounded false positives and false negatives.

Since the model allows the worm author to choose any bytes whatsoever for the variant part of the worm, Hamsa is provably robust to any polymorphism attack. Such analytical guarantees are especially critical when designing algorithms against a human adversary who is expected to adapt his attacks to evade our system. However, to the best of our knowledge, we are the *first* to provide such analytical guarantees for polymorphic worm signature generation systems. To give a concrete example, we design an attack in Section 3.2 which could be readily employed by an attacker to evade state-of-the-art techniques like Polygraph [16] while Hamsa successfully finds the correct signature.

The signature generation is achieved by simple greedy algorithms driven by appropriately chosen values for the model parameters that capture our uniqueness assumptions and are fast in practice. Compared with Polygraph, Hamsa is tens or even hundreds of times faster, as verified both analytically and experimentally. Our C++ implementation can generate signatures for a suspicious pool of 500 samples of a single worm with 20% noise and a 100MB normal pool within 6 seconds with 500MB of memory [1]. Thus Hamsa can respond to worm attacks in its crucial early stage. We also provide techniques for a variety of memory and speed trade-offs to further improve the memory requirements. For instance, using MMAP we can reduce the memory usage for the same setup to about 112MB and increase the runtime only by around 5-10 seconds which is the time required to page fault 100MB from disk to memory. All the experiments were executed on a 3.2GHz Pentium IV machine.

In the absence of noise, the problem of generating conjunction signatures, as discussed by Polygraph, is easily solved in polynomial time. Presence of noise drastically affects the computational complexity. We show that finding multi-set signatures, which are similar to Polygraph's conjunction signatures, in the presence of noise is NP-Hard.

In terms of noise tolerance, can bound the false positive by a small constant while the bound on false negative depends on the noise in the suspicious traffic pool. The more accurate is the worm flow classifier in distinguishing worm flows from the normal flows, the better is the bound on false negatives that we achieve. We also provide a generalization for measuring the goodness of signature using any reasonable scoring function and extend our analytical guarantees to that case.

We validate our model of worm flows experimentally and also propose values for parameters characterizing the uniqueness condition using our experimental results. Evaluations on a range of polymorphic worms and polymorphic engines demonstrate that Hamsa is highly efficient, accurate, and attack resilient, thereby significantly outperforming Polygraph [16].

Paper Layout We discuss the problem space and a high level design of Hamsa in Section 2. We formulate the signature generation problem in Section 3 and present our algorithm in Section 4. In Section 5 we generalize our problem formulation to better capture the notion of a "good" signature. We discuss some implementation details in Section 6

[1] if the data is pre-loaded in memory

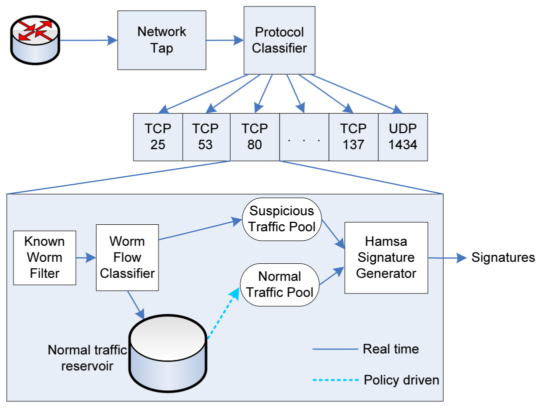

Figure 2. Architecture of Hamsa Monitor

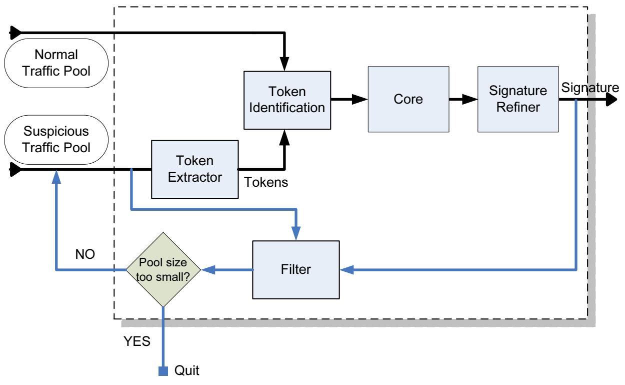

Figure 3. Hamsa Signature Generator

and evaluate Hamsa in Section 7. Finally we compare with related work in Section 8 and conclude in Section 9.

2 Problem Space and Hamsa System Design

2.1 Two Classes for Polymorphic Signatures

Signatures for polymorphic worms can be broadly classified into two categories - content-based and behavior-based. Content-based signatures aim to exploit the residual similarity in the byte sequences of different instances of polymorphic worms. There are two sources of such residual similarity. One is that some byte patterns may be unavoidable for the successful execution of the worm. The other is due to the limitations of the polymorphism inducing code. In contrast, behavior based signatures ignore the byte sequences and instead focus on the actual dynamics of the worm execution to detect them.

Hamsa focuses on content-based signatures. An advantage of content-based signatures is that they allows us to treat the worms as strings of bytes and does not depend upon any protocol or server information. They also have fast signature matching algorithms [27] and can easily be incorporated into firewalls or NIDSes. Next we discuss the likelihood for different parts of a worm (ϵ, γ, π) [5] to contain invariant content.

- ϵ is the protocol frame part, which makes a vulnerable server branch down the code path to the part where a software vulnerability exists.
- γ represents the control data leading to control flow hijacking, such as the value used to overwrite a jump target or a function call.
- π represents the true worm payload, the executable code for the worm.

The ϵ part cannot be freely manipulated by the attackers because the worm needs it to lead the server to a specific vulnerability. For Codered II, the worm samples should necessarily contain the tokens “ida” or “idq”, and “%u”. Therefore, ϵ is a prime source for invariant content. Moreover, since most vulnerabilities are discovered in code that is not frequently used [5], it is arguable that the invariant in ϵ is usually sufficiently unique.

For the γ part, many buffer overflow vulnerabilities need to hard code the return address into the worm, which is a 32-bit integer of which at least the first 23-bit should arguably be the same across all the worm samples. For instance, the register springs can potentially have hundreds of way to make the return address different, but use of register springs increases the worm size as it needs to store all the different address. It also requires considerable effort to look for all the feasible instructions in libc address space for register springing.

For the π part, a perfect worm using sophisticated encryption/decryption (SED) may not contain any invariant content. However, it is not trivial to implement such perfect worms.

As mentioned in [5], it is possible to have a perfect worm which leverages a vulnerability by using advanced register springs and SED techniques does not contain any invariance. This kind of a worm can evade not only our system, but any content-based systems. But in practice such worms are not very likely to occur.

2.2 Hamsa System Design

Figure 2 depicts the architecture of Hamsa which is similar to the basic frameworks of Autograph [10] and Polygraph [16]. We first need to sniff the traffic from networks, assemble the packets to flows, and classify the flows based on different protocols (TCP/UDP/ICMP) and port numbers. Then for each protocol and port pair, we need to filter out the known worm samples and then separate the flows into the suspicious pool ($\mathcal{M}$) and the normal traffic reservoir using a *worm flow classifier*. Then based on a normal traffic selection policy we select some part of the normal traffic reservoir to be the normal traffic pool ($\mathcal{N}$). Since it is usually easy to collect a large amount of normal traffic and we found experimentally that Hamsa is not sensitive to the normal traffic pool, we can selectively choose the amount and the period of normal traffic we use for the normal traffic pool. This strategy prevents attackers from controlling which normal traffic is used by Hamsa and also allows preprocessing of the normal traffic pool. The suspicious and normal traffic pools are given as input to the signature generator (Figure 3) which outputs signatures as described in Sections 4 and 5.

Token Extractor For the token extraction we use a suffix array based algorithm [15] to find all byte sequences that occur in at least λ fraction of all the flows in suspicious pool. The idea is that the worm flows will constitute at least λ fraction of the pool and thus we can reduce the number of tokens that we need to consider.

Core This part implements the algorithms described in Sections 4 and 5.

Token Identification The goal of token identification is to test the tokens' specificity in the normal traffic pool.

Signature Refiner This module finds all common tokens from the flows in the suspicious pool that match the signature outputted by the core. This way we can make our signature more specific (lower false positive) without affecting its sensitivity (the coverage of the suspicious pool).

3 Problem Definition and Computational Challenge

A token is a byte sequence that occurs in a significant number of flows in the suspicious pool. In particular we consider a byte sequence t to be a token if it occurs in at least λ fraction of suspicious flows.

Multiset Signature Model We consider signatures that are *multi-sets* of tokens. For example a signature could be {'host', 'host', 'http://1.1', '0xDDAF', '0xDDA', '0xDDA', '0xDDA'} or equivalently denoted as {('host',2), ('http://1.1',1), ('0xDDAF',1), ('0xDDA',3)}. A flow matches this signature if it contains at least one occurrence each of 'http://1.1' and '0xDDAF', two occurrences of token 'host', and three occurrences of token '0xDDA', where overlapping occurrences are counted separately.

A flow W is said to *match* a multi-set of tokens $\{(t_1, n_1), \ldots, (t_k, n_k)\}$ if it contains at least n_j copies of t_j as a substring. For a set of flows $\mathcal{A}$ and a multi-set of tokens $\mathcal{T}$, let $\mathcal{A}_\mathcal{T}$ denote the largest subset of $\mathcal{A}$ such that every flow in $\mathcal{A}_\mathcal{T}$ matches $\mathcal{T}$.

Note that a multiset signature does not capture any ordering information of tokens. While a worm author may be constrained to include the invariant bytes in a specific order, the ordering constraint makes the signature easy to evade by inserting spurious instances of the invariant tokens in the variant part. An example of such an attack called the coincidental-pattern attack is discussed in [16].

Matching of Multiset Signatures Counting the number of overlapping occurrences of a set of tokens in a flow of length ℓ can be done in time $O(\ell + z)$ where z is the total number of occurrences. This is achieved by using a keyword tree as proposed by [2]. The keyword tree can be constructed in time $O(\tau)$ as a preprocessing step where τ is the total length of all the distinct tokens in all the signatures. Therefore, a set of signatures is first preprocessed to construct a keyword tree of all the distinct tokens. Then for each incoming flow, all the overlapping occurrences of the tokens are counted in linear time. Given these counts, we can check if the flow matches any of the signatures. This check can be done in time linear to the number of tokens in all the signatures and can thus be used for high-speed filtering of network flows. Currently, the improved hardware-based approach [27] can archive 6 – 8Gb/s.

Architecture The worm flow classifier labels a flow as either worm or normal. The flows labeled worms constitute the *suspicious* traffic pool while those labeled normal constitute the *normal* traffic pool. If the flow classifier is perfect, all the flows in the suspicious pool will be worm samples. Then finding a multi-set signature amounts to simply finding the tokens common to all the flows in the suspicious pool which can be done in linear time. However, in practice flow classifiers at the network level will have some false positives and therefore the suspicious pool may have some normal flows as noise. Finding a signature from a noisy suspicious pool makes the problem NP-Hard (Theorem 1).

3.1 Problem Formulation

Given a suspicious traffic pool $\mathcal{M}$ and a normal traffic pool $\mathcal{N}$, our goal is to find a signature $\mathcal{S}$ that covers most of the flows in $\mathcal{M}$ (low false negative) but not many in $\mathcal{N}$ (low false positive). Let $\text{FP}_\mathcal{S}$ denote the false positive of signature $\mathcal{S}$ as determined by the given normal pool and $\text{COV}_\mathcal{S}$ denote the true positive of $\mathcal{S}$ or the fraction of suspicious flows covered by $\mathcal{S}$. That is, $\text{FP}_\mathcal{S} = \frac{|\mathcal{N}_\mathcal{S}|}{|\mathcal{N}|}$ and $\text{COV}_\mathcal{S} = \frac{|\mathcal{M}_\mathcal{S}|}{|\mathcal{M}|}$.

Problem 1 (Noisy Token Multiset Signature Generation (NTMSG)).
INPUT*: Suspicious traffic pool* $\mathcal{M} = \{M_1, M_2, \ldots\}$ *and normal traffic pool* $\mathcal{N} = \{N_1, N_2, \ldots\}$*; value* $\rho < 1$.

$\mathcal{M}$	: suspicious traffic pool	$\mathcal{N}$	: normal traffic pool
λ	: parameter for token extraction	$\mathcal{A}_{\mathcal{T}}$	: largest subset of traffic pool $\mathcal{A}$ that matches multiset of tokens $\mathcal{T}$
$\text{FP}_{\mathcal{S}}$	: $\frac{\lvert\mathcal{N}_{\mathcal{S}}\rvert}{\lvert\mathcal{N}\rvert}$ for a multiset of tokens $\mathcal{S}$	$\text{COV}_{\mathcal{S}}$	: $\frac{\lvert\mathcal{M}_{\mathcal{S}}\rvert}{\lvert\mathcal{M}\rvert}$ for a multiset of tokens $\mathcal{S}$
$u(i), k^*$	: model parameters	σ_i	: $\sum_{j=1\ldots i} u(j)$
ℓ	: maximum token length	T	: total number of tokens
$\lvert\mathcal{M}\rvert$	: number of suspicious flows in $\mathcal{M}$	$\lvert\mathcal{N}\rvert$	: number of normal flows in $\mathcal{N}$
$\mathcal{M}^1$	: set of true worm flows in $\mathcal{M}$	$\mathcal{M}^2$	: set of normal flows in $\mathcal{M}$
α	: coverage of true worms	β	: false positive of invariant content of the true worm

Table 1. Notations used

OUTPUT: *A multi-set of tokens signature* $\mathcal{S} = \{\tilde{t_1}, \ldots, \tilde{t_k}\}$ *such that* $\text{COV}_{\mathcal{S}}$ *is maximized subject to* $\text{FP}_{\mathcal{S}} \leq \rho$.

3.2 Hardness

In the absence of noise, generation of multiset signatures is a polynomial time problem, since it is equivalent to extract all the tokens which cover all the samples. According to Theorem 1, introduction of noise drastically alters the computational complexity of the problem.

Theorem 1. *NTMSG is NP-Hard*

An Illustration The multiset signatures we consider are very similar to conjunction signatures proposed by Polygraph. In absence of noise, these signatures are generated by simply finding the set of tokens common to all the flows in the suspicious pool. In presence of noise, Polygraph proposes to use hierarchical clustering to separate out noise and worm traffic and then uses the noise-free signature generation algorithm for each cluster. We now discuss how this technique fails in presence of arbitrarily small noise.

Since the worms are generated by an attacker who will try to evade the system by exploiting its weakness, it is imperative to examine the worst case scenario. Hierarchical clustering begins with considering each sample to be in a cluster of its own. It then iteratively merges two clusters with the optimum score. For signature generation, the clusters whose union gives a signature with the least false positive is merged at every iteration. The process is continued till either there is only one cluster or merging any two clusters results in high false positive. Now if the variant part of worm flows contain common tokens in normal pool, hierarchical clustering will tend to cluster a worm sample with the normal sample and hence fail to separate out the suspicious samples. In their experiments, they find that this works well if the variant part of the worm flows is randomly generated. However, on using a smaller distribution the algorithm suffers from false negatives [16].

For our example, suppose the invariant content consists of three tokens t_a, t_b and t_c each of which have the same false positive (i.e. coverage in the normal traffic pool) and occur independently. Let $t_{11}, t_{12}, t_{13}, t_{14}, t_{21}, t_{22}, t_{23}, t_{24}, t_{31}, t_{32}, t_{33}$ and t_{34} be 12 other tokens such that the false positive of each of them is the same as that of the invariant tokens and they occur independent of each other. Let W_1, W_2 and W_3 be three worm flows such that W_i consists of all the invariant tokens and the tokens t_{ij} for all j. Let N_1, N_2 and N_3 be three normal flows where N_i consists of tokens t_{ij} for all j. Let the suspicious pool $\mathcal{M}$ contain 99 copies of each of the worm samples W_i and 1 copy of each of the normal sample N_i. Therefore there is 1% noise in the traffic pool. During the first few iterations hierarchical clustering will cluster all the copies of W_i in one cluster to have six clusters - 3 with 99 samples corresponding to W_i's and 3 with a single copy of all the N_i's. Since the false positive of $\{t_{i1}, t_{i2}, t_{i3}, t_{i4}\}$ is smaller than that of $\{t_a, t_b, t_c\}$, the hierarchical clustering will merge the cluster of W_i with that of N_i and terminate with three clusters whose signature is $\{t_{i1}, t_{i2}, t_{i3}, t_{i4}\}$. Hence, it fails to find the invariant content as the signature.

Therefore, presence of noise makes the problem of signature generation for polymorphic worms considerably harder. In Section 4 we present an algorithm which is able to bound the false positive and false negative of the output signature while allowing the attacker full flexibility in including any content in the variant part of different instance of the polymorphic worms. We note that the proposed algorithm outputs the right signature $\{t_a, t_b, t_c\}$ in the above example.

4 Model-based Greedy Signature Generation Algorithm

Though the problem NTMSG is NP-hard in general, under some special conditions it becomes tractable. To capture these conditions, we make the assumption that the input set of flows is generated by a model as follows.

Majority of flows in $\mathcal{M}$ are worms while the remaining are normal flows which have the same token distribution as the flows in $\mathcal{N}$. The worm flows are generated by an adversary who is constrained to include a multiset of tokens $\mathcal{I}$, called the *invariant*, in any order in each of the worm flow. Other than this the adversary has full flexibility over the rest of worm flow.

We now define an ordering of the tokens in $\mathcal{I}$ as follows. Let $\mathcal{I} = \{\hat{t_1}, \hat{t_2}, \ldots, \hat{t_k}\}$ such that

$$\begin{aligned} \mathrm{FP}_{\{\hat{t_1}\}} &\leq \mathrm{FP}_{\{t_i\}} \quad \forall\, i \\ \mathrm{FP}_{\{\hat{t_1},\hat{t_2}\}} &\leq \mathrm{FP}_{\{\hat{t_1},t_i\}} \quad \forall\, i > 1 \\ \mathrm{FP}_{\{\hat{t_1},\ldots,\hat{t_j}\}} &\leq \mathrm{FP}_{\{\hat{t_1},\ldots,\hat{t_{j-1}},t_i\}} \quad \forall\, j\, \forall\, i > j-1 \end{aligned}$$

In words, $\hat{t_1}$ is the token with the smallest false positive rate. The token which has the least false positive value in conjunction with $\hat{t_1}$ is $\hat{t_2}$. Similarly for the rest of $\hat{t_i}$'s.

We propose a model Γ with parameters k^*, $u(1), u(2), \ldots, u(k^*)$ to characterize the invariant $\mathcal{I}$. The constraint imposed by $\Gamma(k^*, u(1), \ldots, u(k^*))$ on $\mathcal{I} = \{\hat{t_1}, \hat{t_2}, \ldots, \hat{t_k}\}$ is that $\mathrm{FP}_{\{\hat{t_1},\ldots,\hat{t_j}\}} \leq u(j)$ for $j \leq k^*$; $k \leq k^*$ and the occurrence of $\hat{t_i}$ in the normal traffic pool is independent of any other token.

In other words, the model makes the assumption that there exists an ordering of tokens such that for each i, the first i tokens appear in only $u(i)$ fraction of the normal traffic pool. Consider the following values for the u-parameters for $k^* = 5$: $u(1) = 0.2$, $u(2) = 0.08$, $u(3) = 0.04$, $u(4) = 0.02$ and $u(5) = 0.01$. The constraint imposed by the model is that the invariant for the worm contains at least one token $\hat{t_1}$ which occurs in at most 20% ($u(1)$) of normal flows. There also exists at least another token $\hat{t_2}$ such that at most 8% ($u(2)$) of the normal flows contain both $\hat{t_1}$ and $\hat{t_2}$. Similarly for the rest of $u(j)$.

Note that the assumption is only on the invariant part over which the attacker has no control. Such invariant bytes can include protocol framing bytes, which must be present for the vulnerable server to branch down the code path where a software vulnerability exists; and the value used to overwrite a jump target (such as a return address or function pointer) to redirect the servers execution. *The attacker is allowed full control over how to construct the worm flow as long as they contain the tokens in $\mathcal{I}$.* We also allow the attacker to order the tokens in the invariant in any manner though in some cases he may not enjoy such flexibility.

In essence, the model imposes some uniqueness constraint on the tokens that comprise the worm invariant. This uniqueness constraint is captured by the u-values as discussed above. If all the tokens in the worm invariant are very popular in normal traffic, then the proposed greedy algorithm cannot be guaranteed to find a good signature. However, since the invariant content is not under the control of the worm author, such an assumption is reasonable. We use experimental valuations to validate this and propose some reasonable values for the u-values.

Algorithm 1 NTMSG($\mathcal{M}$,$\mathcal{N}$)

```
1. $\mathcal{S} \leftarrow \Phi$
2. **For** $i = 1$ to $k^*$
    (a) Find the token $t$ such that $\mathrm{FP}_{\mathcal{S}\cup\{t\}} \leq u(i)$ and $|\mathcal{M}_{\mathcal{S}\cup\{t\}}|$ is maximized. **If** no such token exists, **then output** "No Signature Found".
    (b) $\mathcal{S} \leftarrow \mathcal{S} \cup \{t\}$
    (c) **if** $\mathrm{FP}_{\mathcal{S}} < \rho$, **then output** $\mathcal{S}$.
3. **Output** $\mathcal{S}$.
```

4.1 Runtime Analysis

We first execute a preprocessing stage which consists of token extraction and labeling each token with the flows in $\mathcal{M}$ and $\mathcal{N}$ that it occurs in. If ℓ is the maximum token length, T the total number of tokens and m and n the total byte size of suspicious pool and the normal pool respectively, then this can be achieved in time $O(m + n + T\ell + T(|\mathcal{M}| + |\mathcal{N}|))$ by making use of suffix arrays [15].

Given T tokens, Algorithm 1 goes through at most k^* iterations where k^* is a model parameter. In each iterations, for each token t we need to determine the false positive and coverage of the signature obtained by including the token in the current set. Using the labels attached to each token, this can be achieved in time $O(|\mathcal{M}| + |\mathcal{N}|)$. Therefore the running time of the algorithm is $O(T(|\mathcal{M}| + |\mathcal{N}|))$. Since $|\mathcal{N}|$ is usually greater than $|\mathcal{M}|$, we get a runtime of $O(T \cdot |\mathcal{N}|)$.

4.2 Attack Resilience Analysis

Let $\mathcal{M}^1$ be the set of true worm flows in $\mathcal{M}$ and let $\mathcal{M}^2 = \mathcal{M} \setminus \mathcal{M}^1$. Let the fraction of worm traffic flows in $\mathcal{M}$ be α, i.e. $\frac{|\mathcal{M}^1|}{|\mathcal{M}|} = \alpha\cdot$.

Theorem 2. *Under the adversary model $\Gamma(k^*, u(1), \ldots, u(k^*))$, if the invariant contains k^* tokens Algorithm 1 outputs a signature $\mathcal{S}_{OUT}$ such that $\frac{|\mathcal{M}^1_{\mathcal{S}_{OUT}}|}{|\mathcal{M}^1|} \geq 1 - \sigma_{k^*}\cdot\frac{(1-\alpha)}{\alpha}$ where $\sigma_i = \sum_{j=1}^{i} u(j)$.*

Proof. We prove the above by induction on the number of iterations for the loop in Algorithm Model-Based-Greedy-Set-Signature.

Let $H(j)$ denote the statement that after the j^{th} iteration $|\mathcal{M}^1_{\mathcal{S}}| \geq \alpha\cdot|\mathcal{M}| - \sum_{i=1}^{j} u(i)\cdot(1-\alpha)\cdot|\mathcal{M}|$.

Base Case: $j = 1$. Let the token selected in the first iteration be $\tilde{t_1}$. Since $\mathcal{M}_{\{\hat{t_1}\}} \geq \alpha\cdot|\mathcal{M}|$, $\mathcal{M}_{\{\tilde{t_1}\}} \geq \alpha\cdot|\mathcal{M}|$.

Since $\text{FP}_{\{\tilde{t_1}\}} \leq u(1)$ and the distribution of tokens in $\mathcal{M}^2$ is the same as that in $\mathcal{N}$, $|\mathcal{M}^2_{\{\tilde{t_1}\}}| \leq u(1)\cdot|\mathcal{M}^2| = u(1)\cdot(1-\alpha)\cdot|\mathcal{M}|$.

Therefore, $|\mathcal{M}^1_{\{\tilde{t_1}\}}| = |\mathcal{M}_{\{\tilde{t_1}\}} \setminus \mathcal{M}^2_{\{\tilde{t_1}\}}| = |\mathcal{M}_{\{\tilde{t_1}\}}| - |\mathcal{M}^2_{\{\tilde{t_1}\}}| \geq \alpha\cdot|\mathcal{M}| - u(1)\cdot(1-\alpha)\cdot|\mathcal{M}|$. Hence, $H(0)$ is true.

Induction Step: $H(j-1)$ holds for some j, where $0 \leq j-1 \leq k^*-1$. Let the signature at the end of $(j-1)^{\text{th}}$ iteration be $\mathcal{S}_{j-1}$. Let the token selected at the j^{th} iteration be $\tilde{t_j}$ and let $\mathcal{S}_j = \mathcal{S}_{j-1} \cup \{\tilde{t_j}\}$. Let $\mathcal{S}' = \mathcal{S}_{j-1} \cup \{\hat{t_j}\}$

By induction hypothesis, $|\mathcal{M}^1_{\mathcal{S}_{j-1}}| \geq \alpha\cdot|\mathcal{M}| - \sum_{i=1}^{j-1} u(i)\cdot(1-\alpha)\cdot|\mathcal{M}|$. Since $\mathcal{M}^1_{\{\hat{t_j}\}} = \mathcal{M}^1$, $|\mathcal{M}^1_{\mathcal{S}'}| = |\mathcal{M}^1_{\mathcal{S}_{j-1}}|$. Therefore, $|\mathcal{M}_{\mathcal{S}'}| \geq |\mathcal{M}^1_{\mathcal{S}_{j-1}}|$. Since $\tilde{t_j}$ has the maximum coverage at the j^{th} iteration, $|\mathcal{M}_{\mathcal{S}_j}| \geq |\mathcal{M}_{\mathcal{S}'}| \geq |\mathcal{M}^1_{\mathcal{S}_{j-1}}|$. Since $|\mathcal{M}^2_{\mathcal{S}_j}| \leq u(j)\cdot(1-\alpha)\cdot|\mathcal{M}|$, $|\mathcal{M}^1_{\mathcal{S}_j}| \geq |\mathcal{M}^1_{\mathcal{S}_{j-1}}| - u(j)\cdot(1-\alpha)\cdot|\mathcal{M}| \geq \alpha\cdot|\mathcal{M}| - \sum_{i=1}^{j-1} u(i)\cdot(1-\alpha)\cdot|\mathcal{M}|$.

Further, since $\frac{|\mathcal{M}^1|}{|\mathcal{M}|} = \alpha\cdot$, we get $\frac{|\mathcal{M}^1_{\mathcal{S}_{OUT}}|}{|\mathcal{M}^1|} \geq 1 - \sigma_{k^*}\cdot\frac{(1-\alpha)}{\alpha}$. □

Discussion of Theorem 2 Let the false negative of a signature $\mathcal{S}$ be the fraction of worm flows in $\mathcal{M}$ that are not covered by $\mathcal{S}$. Theorem 2 implies that the false negative rate of the output signature $\mathcal{S}_{\text{OUT}}$ is at most $\frac{\sigma_{k^*}\cdot(1-\alpha)}{\alpha}$ which is inversely proportional to α, the fraction of worm samples in the suspicious pool. So as this fraction decreases, the false negative increases. In other words, the signature has a higher false negative if there is more noise in the suspicious pool. However the false positive of the output signature is always low ($< \rho$).

For example, for $k^* = 5$, $u(1) = 0.2$, $u(2) = 0.08$, $u(3) = 0.04$, $u(4) = 0.02$ and $u(5) = 0.01$, if the noise in the suspicious pool is 5%, then the bound on the false negative is 1.84%. If the noise is 10%, then the bound becomes 3.89% and for noise of 20%, it is 8.75%. Hence, *the better the flow classifier, the lower are the false negatives*.

Note that Theorem 2 gives a lower bound on the coverage of the signature of the suspicious pool (and thereby an upper bound on false negative) that the algorithm generates in the worst case. However, in practice the signatures generated by the algorithm have a much lower false negative than this worst case bound. To create worms for the worst case scenario the attacker needs to include a precise amount of spurious tokens in the variant part of the worm flows. Including either more or less than that amount will result in better false negatives. This precise amount depends on α, the fraction of true worms in the suspicious traffic pool. It is unlikely that an attacker knows the exact value of α in advance.

Also note that while Theorem 2 assumes the number of tokens k in the invariant content to be equal to k^*, since k is generally not known in advance, k^* is chosen to be an upper bound on k. Therefore, in practice, the signature with k^* tokens may be too specific. Algorithm 1 deals with this issue crudely by breaking from the for loop as soon as a signature with low enough false positive rate is found. In Section 5 we address this issue in a greater detail and select the optimal number of tokens in the output signature to achieve both good sensitivity and specificity.

4.3 Attack Resilience Assumptions

In the previous section we give analytical guarantees on the coverage of the output signature of the suspicious traffic pool under the adversary model Γ. In this section we note that these attack resilience guarantees hold under certain assumptions on the system. Such assumptions provide potential avenues of attack and many of them have been discussed before. For each assumption we also discuss how they can be exploited by the attacker and how can the system be made further resilient to such attacks.

We first discuss the assumptions common to any automatic signature generation system using this model an is hence a potential vulnerability of the approach in general.

Common Assumption 1. *The attacker cannot control which worm samples are encountered by Hamsa.*

Note that if we allow the attacker to control which worm samples are encountered by the signature generation system, then it is not possible for any system to generate good signatures. Therefore, it is reasonable to assume that the attacker doesn't have such control and the worm samples in the suspicious pool are randomly selected from all the worm flows. One way to achieve this could be by collecting the worm samples from different locations on the internet.

Common Assumption 2. *The attacker cannot control which worm samples encountered by Hamsa are classified as worm samples by the worm flow classifier.*

An attack exploiting this is similar to the previous one and is an issue of the resilience of the worm flow classifier. To make such attacks difficult, we can use more than one worm flow classifier and aggregate their opinion to classify a flow as either worm or normal.

The following assumptions are unique to our system. Though they may also be required by other systems based on this model, they are not inherent to the model.

Unique Assumption 1. *The attacker cannot change the frequency of occurrence of tokens in normal traffic.*

If the attacker knows when the flows constituting the normal traffic pool are collected, she can attempt to send spurious normal traffic to corrupt the normal traffic pool. Since the byte frequencies/token frequencies in the normal traffic pool are relatively stable, we can collect the samples for the normal pool at random over a larger period of time to counter this attack. For instance, one can use tokens generated over a one hour period the previous day to serve as the normal pool for the same hour today. Using any deterministic strategy like this will still be vulnerable to normal pool poisoning. By including some randomness in the period for which the normal traces are chosen makes such attacks increasing difficult. The success of these measures depends on the relative stability of token frequencies in the normal pool.

Unique Assumption 2. *The attacker cannot control which normal samples encountered by Hamsa are classified as worm samples by the worm flow classifier.*

This assumption is required to ensure that the normal traffic misclassified as suspicious by the worm flow classifier has the same token distribution as the normal traffic. Note that this is an assumption on the worm classifier and not the signature generator. However, the recent work [20] propose an approach which could inject arbitrary amount of noise into the suspicious pool. This approach potentially could invalid the assumption we made here. It is an open question for us to develop better semantic based flow classifier which will not influence by it.

Note that the first two assumptions are generic for any signature generation algorithm while the last two are assumptions on the performance of the worm flow classifier. For the core generation algorithm we propose, the only assumptions are on the invariant part of the worm samples over which the attacker has no control on the first place. For the variant part, we allow the attacker to choose any byte sequences whatsoever.

If we make the assumption that the worm samples encountered by Hamsa are randomly chosen from the entire population of worm samples, then we can give high probability bounds on the true positives which will depend upon the size of the suspicious traffic pool or the number of worm samples in the pool. This is akin to sample complexity analysis.

5 Generalizing Signature Generation with Noise

The signature generation problem formulation as discussed in Section 4 contains a parameter ρ which is a bound on the false positive. The goal is to generate a signature that maximizes coverage of the suspicious pool while not causing a false positive of greater than ρ. In our experiments we found that for a fixed value of ρ, while some worms gave a "good" signature, others didn't. This indicates that Problem NTMSG does not accurately capture the notion of a "good" signature. In this section we generalize our problem formulation to do so.

5.1 Criterion for a "Good" Signature

To generate good signatures, it is imperative to formalize the notion of a good signature. For the single worm, the goodness of a signature will depend on two things, the coverage of the signature of the suspicious pool (coverage) and the coverage of the normal pool (false positive). Intuitively a good signature should have a high coverage and a low false positive.

NTMSG tries to capture this intuition by saying that given two signatures, if the false positive of both is above a certain threshold, both are bad signatures, if the false positive of only one is below the threshold, it is a good signature, and if the false positive of both is below the threshold, then the one with the higher coverage is better. Sometimes this criterion leads to counter intuitive goodness ranking. For example, if the threshold for false positive is say 2%, and signature A has a false positive of 1.5% and coverage of 71% while signature B has a false positive of 0.3% and coverage of 70%. According to our goodness function, signature A is better though conceivably one may prefer signature B over A for even though it has a slightly higher false negative, its false positive is considerably lower.

Arguably, there is a certain amount of subjectivity involved in making this trade-off between false positive and false negative. To capture this, we define a notion of scoring function which allows full flexibility in making this trade-off.

Scoring Function Given a signature $\mathcal{S}$ let $\mathrm{score}(\mathrm{COV}_{\mathcal{S}}, \mathrm{FP}_{\mathcal{S}})$ be the score of the signature where the scoring function $\mathrm{score}(\cdot,\cdot)$ captures the subjective notion of goodness of a signature. While there is room for subjectivity in the choice of this scoring function, any reasonable scoring function should satisfy the following two properties.

1. $\mathrm{score}(x, y)$ is monotonically non-decreasing in x.
2. $\mathrm{score}(x, y)$ is monotonically non-increasing in y.

5.2 Generalization of NTMSG

We capture this generalized notion of goodness in the following problem formulation.

Problem 2 (Generalized NTMSG(GNTMSG))**.**
INPUT*: Suspicious traffic pool* $\mathcal{M} = \{M_1, M_2, \ldots\}$ *and*

normal traffic pool $\mathcal{N} = \{N_1, N_2, \ldots\}$.

OUTPUT*: A set of tokens* $\mathcal{S} = \{t_1, \ldots, t_k\}$ *such that* $\mathrm{score}(\mathrm{COV}_\mathcal{S}, \mathrm{FP}_\mathcal{S})$ *is maximized.*

Theorem 3. *GNTMSG is NP-Hard.*

Algorithm 2 Generalized-NTMSG($\mathcal{M}$,$\mathcal{N}$,score$(\cdot,\cdot)$)

```
1. For $i = 1$ to $k^*$
    (a) Find the token $t$ such that $\mathrm{FP}_{\mathcal{S}_{i-1}\cup\{t\}} \leq u(i)$ and $|\mathcal{M}_{\mathcal{S}_{i-1}\cup\{t\}}|$ is maximized
    (b) $\mathcal{S}_i \leftarrow \mathcal{S}_{i-1} \cup \{t\}$
    (c) if $\mathrm{FP}_{\mathcal{S}_i} > u(i)$, then goto Step 2
2. Output $\mathcal{S}_i$ which maximizes $\mathrm{score}(\mathrm{COV}_{\mathcal{S}_i}, \mathrm{FP}_{\mathcal{S}_i})$.
```

5.3 Performance Guarantees for GNTMSG

Let α be the coverage of the true worm and let β be the false positive of its invariant content.

Theorem 4. *Under the adversary model* $\Gamma(k^*, u(1), \ldots, u(k^*))$*, if the fraction of worm traffic flows in* $\mathcal{M}$ *is* α*, then Algorithm 2 outputs a signature* $\mathcal{S}_{OUT}$ *such that for all* $i \leq k$, $\mathrm{score}(\alpha, \beta) \leq \mathrm{score}\left(\frac{\mathrm{COV}_{\mathcal{S}_i}+\sigma_i}{1+\sigma_i}, 0\right)$.

After executing Algorithm GNTMSG and finding all the $\mathcal{S}_i$'s, Theorem 4 can be used to get an upper bound on the score of the true worm. This way we can determine how far could the score of our signature be from that of the true worm.

Theorem 5. *Under the adversary model* $\Gamma(k^*, u(1), \ldots, u(k^*))$*, if the fraction of worm traffic flows in* $\mathcal{M}$ *is* α*, then Algorithm 2 outputs a signature* $\mathcal{S}_{OUT}$ *such that for all* $i \leq k$, $\mathrm{score}(\mathrm{COV}_{\mathcal{S}_{OUT}}, \mathrm{FP}_{\mathcal{S}_{OUT}}) \geq \mathrm{score}(\alpha - \sigma_i(1-\alpha), u(i))$.

Theorem 5 is a guarantee on the performance of the algorithm. That is independent of the run of the algorithm, we can lower bound the score of the signature that our algorithm is guaranteed to output.

6 Implementation Details

6.1 Scoring Function

As discussed in Section 5.1, to select a reasonable scoring function $\mathrm{score}(\mathrm{COV}_\mathcal{S}, \mathrm{FP}_\mathcal{S})$ is to make a subjective trade off between the coverage and false positive to catch the intuition of what is a good signature. [9] proposes an information theoretic approach to address this issue. However, for our implementation we use the following scoring function:

$$\begin{aligned}\mathrm{score}(\mathrm{COV}_\mathcal{S}, \mathrm{FP}_\mathcal{S}, \mathrm{LEN}_\mathcal{S}) = & -\log((\delta + \mathrm{FP}_\mathcal{S}), 10) \\ & + a * \mathrm{COV}_\mathcal{S} + b * \mathrm{LEN}_\mathcal{S} \\ & a >> b\end{aligned}$$

δ is used to avoid the log term becoming too large for $\mathrm{FP}_\mathcal{S}$ close to 0. We add some weight to the length of the token $\mathrm{LEN}_\mathcal{S}$ to break ties between signatures that have the same coverage and false positive rate. This is because even though two signatures may have the same false positive on our limited normal pool size, the longer signature is likely to have smaller false positive over the entire normal traffic and is therefore preferable.

For our experiments, we found $\delta = 10^{-6}$, $a = 20$ and $b = 0.01$ yields good results.

6.2 Token Extraction

Like Polygraph, we extract tokens with a minimum length ℓ_{min} and a minimum coverage λ in the suspicious pool. However, Polygraph's token extraction algorithm does not include a token if it is a substring of another token, unless its unique coverage (i.e. without counting the occurrences where it is a substring of other tokens) is larger than λ. This may potentially miss some invariant tokens, *e.g.*"%u" may occur only as either "%uc" and "%uk", which means that the unique coverage of "%u" is 0. However, it might be possible that "%u" covers all of the worm samples, but "%uc" and "%uk" do not, and so "%u" yields a better signature. Therefore, for our token extraction algorithm, every string with a coverage larger than λ is treated as a token.

Problem 3 (Token Extraction)**.**
INPUT*: Suspicious traffic pool* $\mathcal{M} = \{M_1, M_2 \ldots\}$*; the minimum token length* ℓ_{min} *and the minimum coverage* λ.

OUTPUT*: A set of tokens* $\mathcal{T} = \{t_1, t_2, \ldots\}$ *which meet the minimum length and coverage requirements and for each token the associated sample vector* $\mathcal{V}(t_i) = [a_{i1}, \ldots, a_{i|\mathcal{M}|}]$, $i \in [1, |\mathcal{M}|]$ *where* a_{ij} *denote the number of times token* t_i *occurs in flow* M_j.

Polygraph used a suffix tree based approach for token extraction. The basic idea is to do a bottom up traversal of the suffix tree to calculate a frequency vector of occurrences for each node (token candidate), and then via a top down traversal output the tokens and corresponding sample vectors which meet the minimum length and coverage requirement.

Although asymptotically linear, the space consumption of a suffix tree is quite large. Even recently improved implementations of linear time constructions require 20 bytes per input character in the worst case. [1] have proposed techniques that allows us to replace the suffix tree data structure with an enhanced suffix array for the token extraction algorithm. The suffix array based algorithm runs in linear time and requires a space of at most 8 bytes per input character. Another advantage of suffix array based approach is that it allows some pruning techniques to further speed up token extraction and improve memory consumption.

Though there are linear time suffix array creation algorithms, some lightweight algorithms with a worse bound on the worst case time complexity perform better for typical input sizes (such as less than 1000 samples). The reason as discussed in [23] is that the linear time algorithm makes too many random accesses to the main memory which makes the cache hit ratio low and result in poor performance. So for our implementation we choose a lightweight algorithm, `deepsort` [15], which is one of fastest suffix array construction algorithm in practice. Our experiments with one of the best known suffix tree libraries [11] show that we get around 100 times speedup for token extraction by using suffix arrays.

6.3 False Positive Calculation

For false positive estimation, we build a suffix array [15] of the normal traffic pool in a preprocessing step and store it on the disk. To calculate the false positive of a given token, we use binary search on the suffix array. We can employ a variety of different policies for maintaining the normal traffic pool in order to prevent an attacker from polluting it.

The normal traffic pool could be large, *e.g.*, 100MB and a suffix array for 100MB requires around 400MB of memory. Currently, we use **mmap** to map the suffix array to the memory space of our program. When we need to access some part of the array, a page fault happens and the relevant page (4KB) is loaded to the memory. In our experience, we found that we get good performance with only 50MB–200MB memory using this approach.

The large memory requirement due to suffix arrays can also be alleviated at the cost of accuracy, speed or expense as follows.

1. By dividing the normal pool randomly into a number of equal sized chunks and creating a suffix array over each of these chunks, the false positive can be approximated by the false positive over any one of these chunks kept in primary storage while the rest are in secondary storage. For tokens whose false positive is close to the threshold, a more accurate estimation can be performed by using chunks of normal traffic pool from secondary storage.

2. Each normal flow can be compressed using compression schemes such as LZ1. To compute the false positive for a token t, we can employ the string matching algorithms over compressed strings as discussed by Farach et al. [8]. This approach is more time consuming than suffix array based approach but doesn't sacrifice accuracy.

3. Since the false positive calculation is just a special case of string matching, hardware-based memory-efficient string matching algorithms can be employed. The ASIC/FPGA based implementation [27] can archive a matching speed of 6–8Gb/s. However, such specialized hardware makes the system expensive.

7 Evaluation

7.1 Methodology

Since there are no known polymorphic worms on the Internet, a real online evaluation is not possible. Instead, we test our approach offline on synthetically generated polymorphic worms. Since the flow classification is not the focus of this paper, we assume we have a flow classifier that can separate network traffic into two pools, a normal traffic pool and a suspicious pool (with polymorphic worms and possible noise). We take the two traffic pools as input and output a set of signatures and also their coverage of the suspicious pool and the false positives in the normal traffic pool. The input traffic pools can be treated as training datasets. After signature generation, we match the signature of each worm against 5000 samples generated by the same worm to evaluate false negatives and also against another 16GB of normal network traffic to evaluate false positives. Since most of the worm flow is usually binary code, we also create a binary evaluation dataset for testing false positives against the Linux binary distribution of /usr/bin in Fedora Core 4.

7.1.1 Polymorphic Worm Workload

In related work, Polygraph [16] generates several pseudo polymorphic worms based on real-world exploits for evaluation purposes. Polygraph's pseudo polymorphic worms are based on the following exploits: the Apache-Knacker exploit and the ATPhttpd exploit.

For our experiments, we use Polygraph's pseudo polymorphic worms and also develop a polymorphic version of Code-Red II. The polymorphic version of Code-Red II contains invariant content inherent to Code-Red II. We were able to detect and generate signatures for all of the polymorphic worms even in presence of normal traffic noise.

We also used two polymorphic engines found on the Internet, the CLET [7] and TAPiON [21] polymorphic engines to generate polymorphic worms. The CLET polymorphic engine is a sophisticated engine that is designed to generate polymorphic worms that fit closely to normal traffic distributions. For example, CLET can generate a NOP field for a polymorphic worm using English words. In addition, given a spectral file of byte frequencies, the CLET engine can give precedence to certain byte values when generating bytes for a polymorphic worm. We created a spectral frequency distribution from normal HTTP traffic to use as input to the CLET engine when creating our samples. With all the advanced features of the CLET engine enabled we were still able to detect and generate signatures for samples created by the CLET engine.

The TAPiON polymorphic engine is a very new and recent polymorphic engine. We used the TAPiON engine to generate 5000 samples of a known MS-DOS virus called MutaGen. Again we are able to apply our technique and generate signatures for samples created by the TAPiON engine.

7.1.2 Normal Traffic Data

We collected several normal network traffic traces for the normal traffic pool and evaluation datasets. Since most of our sample worms target web services, we use HTTP traces as normal traffic data. We collected two HTTP traces. The first HTTP trace is a 4-day web trace (12GB) collected from our departmental network gateway. The second HTTP trace (3.7GB) was collected by using web crawling tools that included many different file types: .mp3 .rm .pdf .ppt .doc .swf *etc.*.

7.1.3 Experiment Settings

Parameters for token extraction We set the minimum token length $\ell_{min} = 2$ and require each token to cover at least $\lambda = 15\%$ of the suspicious pool.

Signature generation We used the scoring function defined in Section 6.1 with $a = 20$ and $b = 0.01$. Moreover, we rejected any signature whose false positive rate is larger than 1% in the normal traffic pool. For u-parameters, we chose: $k^* = 15$, $u(1) = 0.15$, and $u_r = 0.5$. Based on u_r we can calculate $u(i) = u(1) * u_r^{(i-1)}$. In Section 7.3, we evaluate this choice of u-parameters.

All experiments were executed on a PC with a 3.2GHz Intel Pentium IV running Linux Fedora Core 4.

7.2 Signature Generation without Noise

We tested our five worms separately without noise. Comparing our approach with Polygraph, we found the signatures we generated were very close to conjunction signatures generated with Polygraph (single worm without noise). We found that our signatures are sometimes more specific than those of Polygraph while maintaining zero false negatives.

For a suspicious pool size of 100 samples and a normal traffic pool size of 300MB, the false negative and false positive measurements on training datasets are very close to those for much larger evaluation datasets. Moreover, we also tested on smaller normal traffic pool sizes: 30MB and 50MB. We found our approach to work well for both large and small pool sizes. Thus, we are not very sensitive to the size of the normal traffic pool. In Section 7.5, we discuss the effects of the number of worms in the suspicious pool on generating correct signatures.

7.3 u-parameter Evaluation

As mentioned before, we can use k^*, $u(1)$, and u_r to generate all the u-parameters. If we set $u(1)$ and u_r too high, it loosens our bound on attack resilience and may also result in signature with have high false positive. If we choose too low a value, we risk generating a signature altogether. Therefore, for all the worms we tested, we evaluated the minimum required value of $u(1)$ and u_r. We randomly injected 80 worm samples and 20 normal traffic noises into the suspicious pool (20% noise), and used the 300MB normal traffic pool. We tested our worms with various combinations of ($u(1)$,u_r) with $u(1)$ taking values from {0.02, 0.04, 0.06, 0.08, 0.10, 0.20, 0.30, 0.40, 0.50}, and u_r from {0.20, 0.40, 0.60, 0.80}. We found the minimum value of ($u(1)$,u_r) that works for all our test worms was (0.08,0.20).

We choose a far more conservative value of $u(1) = 0.15$ and $u_r = 0.5$ for our evaluation. Note that for $k^* = 15$, $u(k^*) = 9.16 * 10^{-6}$.

7.4 Signature Generation in Presence of Noise

The first experiment consists of randomly selecting 100 worm samples for each worm, and injecting different portions of noise, to create different noise ratios: 0%, 10%, 30%, 50%, and 70%. In our second experiment we fix the suspicious pool size to 100 and 200 samples, and evaluate for the noise ratios used in the first experiment.

As shown in Figure 3, Hamsa generates the signatures for the suspicious pool iteratively. So it can generate more than one signature if required and thus detect multiple worms. As shown in Table 2, we always generate worm signatures with zero false negative and low false positive. Since our algorithm generates signatures that have high coverage of the suspicious pool and low false positive of the normal traffic pool, if the noise ratio is larger than

Worm name	Training FN	Training FP	Evaluation FN	Evaluation FP	Binary eval FP	Signature
Code-Red II	0	0	0	0	0	{'.ida?': 1, '%u780': 1, ' HTTP/1.0\r\n': 1, 'GET /': 1, '%u': 2}
Apache-Knacker	0	0	0	0	0.038%	{'\xff\xbf': 1, 'GET ': 1, ': ': 4, '\r\n': 5, ' HTTP/1.1\r\n': 1, '\r\nHost: ': 2}
ATPhttpd	0	0	0	0	0	{'\x9e\xf8': 1, ' HTTP/1.1\r\n': 1, 'GET /': 1}
CLET	0	0.109%	0	0.06236%	0.268%	{ '0\x8b': 1, '\xff\xff\xff': 1, 't\x07\xeb': 1}
TAPiON	0	0.282%	0	0.1839%	0.115%	{'\x00\x00': 1, '\x9b\xdb\xe3': 1}

Table 2. Signatures for the five worms tested and accuracy of these signatures. {'\r\nHost: ': 2} means token '\r\nHost: ' occurs twice in each worm sample. FN stands for "False Negative" and FP stands for "False Positive".

50%, sometimes, we will generate two signatures. However, only one of them is the true signature for the worm in the suspicious pool; the other is due to normal traffic noise. We tested the noise signatures against the binary evaluation dataset and found they all have zero false positives. The average and maximum false positive rate for the 16GB normal traffic pool is 0.09% and 0.7% respectively. The following is an example of a noise signature.

```
'47 ': 1, 'en': 3, 'od': 3, 'ed': 1, 'b/': 1,
': ': 6, ' GMT\r\nServer: Apache/': 1, '0 m': 1
' mod_auth_': 1, '\r\n\r\n': 1, 'odi': 1,
'(Unix) mod_': 1, 'e: ': 2, 'ep': 1, 'er': 3,
'ec': 1, '00': 3, 'mod_ssl/2.': 1, ', ': 2,
'1 ': 2, '47': 2, ' mod_': 2, '4.': 1, '2': 1,
'rb': 1, 'pe': 2, '.1': 3, 'te': 3, '0.': 3,
'.6': 1, '\r\nCon': 2, ' 20': 3, '.3.': 1,
'7 ': 2, '10 ': 1, '13': 1, 'HTTP/1.1 ': 1,
'b D': 1, ' PHP': 1, 'ker': 1, 'on': 5,
'2.0.': 2, 'ma': 1, ' 200': 2, '/2': 3,
'\r\nDate: Mon, 11 Jul 2005 20:': 1, '.4': 1,
' OpenSSL/0.9.': 1, '\r\n': 9, 'e/2': 1,
```

Noise signatures can be identified as follows. If a signature has low coverage than some threshold for a different suspicious pool, then it is likely to be a noise signature. However, since noise signatures have low false positive rates, it is safe to include them as valid signatures.

7.5 Suspicious Pool Size Requirement

For worms obtained from Polygraph and the polymorphic Code-Red II worm, we only need a suspicious pool size of 10 samples (in presence of 20% noise) to obtain the exact same signature as shown in Table 2. However, for worms generated using CLET and TAPiON engines, a small suspicious pool size of 10–50 samples in presence of 20% noise could result in too specific a signature, such as { '0\x8b': 1, '\xff\xff\xff': 1, 't\x07\xeb': 1 'ER': 1 }. This is due to the polymorphic engines using common prefixes or suffixes in English words to pad the variant parts in the worm. This is similar to the *coincidental-pattern* attack mentioned in the Polygraph paper. In the above mentioned example, 95% of the worms have the token 'ER'. It is possible that when the suspicious pool is small, all the samples contain token 'ER', thus making 'ER' seem invariant and hence a part of the signature. This is why the signature above has 0% training false negative, but 5% false negative over the evaluation dataset. Therefore, for unknown worms it is best to use a large suspicious pool size, such as 100 samples.

7.6 Speed Comparison with Polygraph

Signature generation speeds are critical for containing worms in their early stages of infection. Both Polygraph and Hamsa have similar pre-processing requirements. In Section 4.1, we analyzed the time complexity of signature generation for Hamsa to be $O(T{\cdot}|\mathcal{N}|)$ where T is the number of tokens. The hierarchical clustering algorithm proposed by Polygraph needs $O(|\mathcal{M}|^2)$ comparison and for each comparison we need to compute its false positive which takes $O(|\mathcal{N}|)$ time. By making use of appropriate data structures, it is possible to merge the clusters and generate the signature for the new clusters so that the total runtime is $O(|\mathcal{M}|^2{\cdot}|\mathcal{N}|)$.

So the asymptotic runtime difference between the two approaches is $O(T)$ vs. $O(|\mathcal{M}|^2)$. In our experiments, we determine the average number of tokens T of 5 different runs for the same pool size $|\mathcal{M}|$. Table 3 summarizes our experimental observations. Note that the number of tokens T decreases as $|\mathcal{M}|$ increases. The larger the suspicious pool size $|M|$, the bigger the speed up ratio. Table 4 shows that Hamsa is analytically tens to hundreds of times faster than Polygraph. In our experiments over various parameter settings, Hamsa was found to be 64 to 361 times faster than Polygraph [2].

	Noise Ratio			
$\|M\|$	20%	30%	40%	50%
150	303	589	1582	2703
250	290	559	1327	2450
350	274	558	1172	2062

Table 3. The number of tokens for different pool sizes and noise ratios.

	Noise ratio			
$\|M\|$	20%	30%	40%	50%
150	74.26 (64.28)	38.20	14.22	8.32 (69.89)
250	215.52 (361.32)	111.81	47.10	25.51
350	447.08	219.53	104.52	59.41

Table 4. $|\mathcal{M}|^2/T$, the asymptotic speed up ratio with respect to Polygraph. The number in braces indicate the empirical speed up ratio.

7.7 Speed and Memory Consumption Results

We evaluate the speed and memory consumption of our optimized C++ implementation for different settings shown in Table 5. For each of the settings, we run our experiments for all the 5 different worms. The value reported in Table 5 is the maximum of the values obtained for the 5 worms. For the "pre-load" setting, we pre-load the normal traffic pool and its suffix array in the memory before running the code. Since the data is readily available in memory, we achieve very good speeds. However, the pre-load size is 5 times the normal pool size which could be too large for some cases. By using MMAP, we break the suffix array and the normal traffic pool into 4KB pages, and only load the parts which are required by the system. This saves a lot of memory but introduces some disk overheads. In all our experiments, we use a noise ratio of 20%.

[2]For a fair comparison, both systems are implemented in Python and use the same suffix tree based token extraction and the suffix array based false positive calculation techniques.

Number of samples in suspicious pool	Normal pool size (MB)	Memory usage MMAP (MB)	Speed MMAP (secs)	Speed pre-load (secs)
100	101	64.8	11.9	1.7
100	326	129.0	32.7	4.9
200	101	75.4	14.3	2.4
200	326	152.1	39.4	7.2
500	101	112.1	14.9	6.0
500	326	166.6	38.1	8.6

Table 5. Speed and memory consumption under different settings.

7.8 Attack Resilience

Here, we propose a new attack that is similar to the *coincidental-pattern* attack mentioned in Polygraph, but stronger. We call it the *token-fit* attack. It is possible that a hacker may obtain normal traffic with a similar token distribution as the normal noise in the suspicious pool. She can then extract tokens from the normal traffic and intentionally encode tokens into a worm. She may include different sets of tokens into different worm samples. This does not increases the similarity of worm samples in terms of shared tokens, but can increase the similarity of worm samples to normal traffic noise in the suspicious pool; thus, degrading the quality of the signature.

We evaluate both Hamsa and Polygraph for this attack by modifying the ATPhttpd exploit to inject different 40 tokens to the variant part of each worm sample. The tokens are extracted from the normal traffic noise in the same suspicious pool. We test both systems for suspicious pool with 50 samples using a noise ratio of 50%. We run two different trials, and find that Hamsa always output a correct signature as shown in Table 2. However, with the signature produced by Polygraph, no such polymorphic worms can be detected (100% false negative), although there is no false positive.

8 Related Work

Early automated worm signature generation efforts include Honeycomb [12], Autograph [10], and Early-Bird [24]. While these systems use different means to classify worm flows and normal traffic, all of them assume that a worm will have a long invariant substring. However, these techniques cannot be used for polymorphic worms since different instances of polymorphic worms do not contain a long enough common substring.

	Hamsa	Polygraph [16]	Similarity of CFG [13]	PADS [26]	Nemean [31]	COVERS [14]	Malware Detection [4]
Network or Host based	Network	Network	Network	Host	Host	Host	Host
Content or behavior based	Content based	Content based	Behavior based	Content based	Content based	Behavior based	Behavior based
Noise tolerance	Yes	Yes (slow)	Yes	No	No	Yes	Yes
On-line detection speed	Fast	Fast	Slow	Fast	Fast	Fast	Slow
General purpose or application specific	General purpose	General purpose	General purpose	General purpose	Protocol specific	Server specific	General purpose
Provable attack resilience	Yes	No	No	No	No	No	No
Information exploited	$\epsilon\gamma\pi$	$\epsilon\gamma\pi$	π	$\epsilon\gamma\pi$	ϵ	$\epsilon\gamma$	π

Table 6. Summary of relative strengths and weaknesses of different polymorphic worm detection and signature generation techniques proposed recently.

Recently, there has been active research on polymorphic worm signature generation and the related polymorphic worm and vulnerability study [4, 13, 14, 16, 26, 31]. In Table 8, we compare Hamsa with them in terms of the following seven metrics: 1) Network vs. host based: a network based system uses only the network traffic for detection and can be deployed on routers/gateways; 2) Content vs. behavior based detection approach; 3) Noise tolerance; 4) Online worm detection: this depends on the speed with which the signature generated can be compared with network traffic; 5) General purpose vs. application specific: some schemes like Nemean [31] and COVERS [14] require detailed protocol/application specification knowledge to detect the worms for each protocol/application (thus they are mostly host-based); 6) provable attack resilience; and 7) information exploited.

Polygraph [16] comes closest to our system. It considers three methods of generating signatures: (1) set of tokens (2) sequences of tokens, and (3) weighted set of tokens. As shown in Section 7, Hamsa is a significant improvement over Polygraph in terms of both speed and attack resilience.

Position-Aware Distribution Signatures [26] (PADS) bridge signature-based approaches with statistical anomaly-based approaches and are able to detect variants of the MSBlaster worms. However, in presence of noise the accuracy of PADS suffers.

There are also some semantic based approaches. Basically, there are two kinds of semantic information which can be exploited for containing polymorphic worms: protocol information and binary executable code information.

Nemean [31] uses protocol semantics to cluster the worm traffic of the same protocol to different clusters for different worms. It then uses automata learning to reconstruct the connection and session level signature (automata). However, it requires detailed protocol specifications for each and every application protocol. Also, Nemean may fail to produce effective signatures when the suspicious traffic pool contains noise.

Christopher et al., [13] propose an approach based on structural similarity of Control Flow Graphs (CFG) to generate a fingerprint for detecting different polymorphic worms. However, their approach can possibly be evaded by using SED as discussed in Section 2.1. Furthermore, matching fingerprints is computationally expensive and hence may not be useful for filtering worm traffic on high traffic links.

TaintCheck [17] and DACODA [5] dynamically traces and correlates the network input to control flow change to find the malicious input and infer the properties of worms. Although TaintCheck can help in understanding worms and vulnerabilities, it cannot automatically generate the signature of worms. Moreover their technique is very application specific: a certain version of a server must be deployed to monitor a vulnerability to discover how the worm interacts with the server.

COVERS [14], a system based on address-space randomization (ASR) [3] can detect and correlate the network input and generate signatures for server protection. However, although the signature generated can efficiently protect the servers, it cannot be used by NIDSes or firewalls since the hacker can potentially evade it[3]. Moreover, COVERS is application specific.

Mihai et al., [4] model the malicious program behavior and detect the code pieces similar to the abstract model.

[3]Their signature is based on a single worm sample, so the length threshold sometimes can cause false negatives.

However, the their approach is computationally expensive.

9 Conclusion

In this paper we propose Hamsa, a network-based signature generation system for zero-day polymorphic worms which generates multiset of tokens as signatures. Hamsa achieves significant improvements in speed, accuracy, and attack resilience over Polygraph, the previously proposed token-based approach. We prove that multiset signature generation problem is NP-Hard in presence of noise and design model based signature generation algorithms with analytical attack resilience guarantees. The signature generated by Hamsa can be easily deployed at IDSes such as Snort [22] or Bro [19].

10 Acknowledgement

We would like to thank Dawn Song and James Newsome for the Polygraph source code and their insightful discussions. We would also like to thank the anonymous reviewers and our shepherd, Sal Stolfo, for their constructive comments and suggestions. Support for this work was provided by a DOE Early CAREER award.

References

[1] M. I. Abouelhoda, S. Kurtz, et al. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2004.

[2] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 1975.

[3] S. Bhatkar, D. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proc. of USENIX Security*, 2003.

[4] M. Christodorescu, S. Jha, et al. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy*, 2005.

[5] J. R. Crandall, Z. Su, and S. F. Wu. On deriving unknown vulnerabilities from zeroday polymorphic and metamorphic worm exploits. In *Proc. of ACM CCS*, 2005.

[6] Critical Solutions Ltd. Critical TAPs: Ethernet splitters designed for IDS. `http://www.criticaltap.com`.

[7] T. Detristan, T. Ulenspiegel, et al. Polymorphic shellcode engine using spectrum analysis. `http://www.phrack.org/show.php?p=61&a=9`.

[8] M. Farach and M. Thorup. String matching in lempel-ziv compressed strings. *Symposium on the Theory of Computing (STOC)*, 1995.

[9] G. Gu, P. Fogla, et al. Measuring intrusion detection capability: An information-theoretic approach. In *Proc of ACM Symposium on InformAction, Computer and Communications Security (ASIACCS)*, 2006.

[10] H. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *USENIX Security Symposium*, 2004.

[11] C. Kreibich. libstree — generic suffix tree library. `http://www.cl.cam.ac.uk/~cpk25/libstree/`.

[12] C. Kreibich and J. Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *Proc. of the Workshop on Hot Topics in Networks (HotNets)*, 2003.

[13] C. Kruegel, E. Kirda, et al. Polymorphic worm detection using structural information of executables. In *Proc. of Recent Advances in Intrusion Detection (RAID)*, 2005.

[14] Z. Liang and R. Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proc. of ACM CCS*, 2005.

[15] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1), 2004.

[16] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Security and Privacy Symposium*, 2005.

[17] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of NDSS*, 2005.

[18] Packeteer. Solutions for Malicious Applications. `http://www.packeteer.com/prod-sol/solutions/dos.cfm`.

[19] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31, 1999.

[20] R. Perdisci, D. Dagon, W. Lee, et al. Misleading worm signature generators using deliberate noise injection. In *IEEE Security and Privacy Symposium*, 2006.

[21] Piotr Bania. TAPiON. `http://pb.specialised.info/all/tapion/`.

[22] M. Roesch. Snort: The lightweight network intrusion detection system, 2001. `http://www.snort.org/`.

[23] K.-B. Schurmann and J. Stoye. An incomplex algorithm for fast suffix array construction. In *Proceedings of ALENEX/ANALCO*, 2005.

[24] S. Singh, C. Estan, et al. Automated worm fingerprinting. In *Proc. of OSDI*, 2004.

[25] S. Staniford, V. Paxson, and N. Weaver. How to own the Internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium*, 2002.

[26] Y. Tang and S. Chen. Defending against internet worms: A signature-based approach. In *Proc. of Infocom*, 2003.

[27] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *Proc of IEEE Infocom*, 2004.

[28] R. Vargiya and P. Chan. Boundary detection in tokenizing network application payload for anomaly detection. In *ICDM Workshop on Data Mining for Computer Security (DMSEC)*, 2003.

[29] K. Wang, G. Cretu, and S. J. Stolfo. Anomalous payload-based worm detection and signature generation. In *Proc. of Recent Advances in Intrusion Detection (RAID)*, 2005.

[30] K. Wang and S. J. Stolfo. Anomalous payload-based network intrusion detection. In *Proc. of Recent Advances in Intrusion Detection (RAID)*, 2004.

[31] V. Yegneswaran, J. Giffin, P. Barford, and S. Jha. An architecture for generating semantic-aware signatures. In *USENIX Security Symposium*, 2005.